Synthèse des communications dans un environnement de génération de logiciel embarqué pour des plateformes multi-tuiles hétérogènes

A. Chagoya-Garzon

UNIVERSITE DE GRENOBLE
INSTITUT POLYTECHNIQUE DE GRENOBLE

N˚attribué par la bibliothèque
978-2-84813-161-0

THESE

pour obtenir le grade de

DOCTEUR de l'Université de Grenoble
délivré par l'Institut polytechnique de Grenoble

Spécialité : MNE

préparée au laboratoire TIMA

dans le cadre de **l'Ecole Doctorale** Electronique, Electrotechnique, Automatique et Traitement du Signal

présentée et soutenue publiquement

par

Alexandre Chagoya-Garzon

le 3 Décembre 2010

Synthèse des communications dans un environnement de génération de logiciel embarqué pour des plateformes multi-tuiles hétérogènes

Sous le direction de Pr. Frédéric ROUSSEAU et Pr. Frédéric PETROT

JURY

M. Tanguy RISSET , Président
M. Guy GOGNIAT , Rapporteur
M. Bernard TOURANCHEAU , Rapporteur
M. Frédéric ROUSSEAU , Directeur de thèse
M. Frédéric PETROT , Co-encadrant
M. Serdar MANAKLI , Examinateur

Remerciements

J'aimerais tout d'abord remercier chaleureusement Prof. Frédéric Rousseau et Prof Frédéric Pétrot d'avoir accepté d'encadrer ma thèse, et pour leurs précieux conseils et soutien tout au long de mes trois années de thèse.

J'aimerais également remercier tous les partenaires du projet européen SHAPES, en particuler Xavier Guérin, Stefan Kraemer, Davide Rossetti, Francesco Simula et Wolfgang Haid pour tous les échanges enrichissants à l'origine du succès de ce projet.

J'aimerais également remercier tous les doctorants et personnels du laboratoire TIMA pour tous les échanges que nous avons pu avoir à toute occasion : vie associative, conférences ou tout simplement thé/-café.

Et bien entendu, merci à mes parents, famille et amis pour leur soutien et patience durant les moments difficiles de ma thèse, et il y en a eu !

A mes très chers parents, à qui je dois tout

Table des matières

Introduction générale

La loi de Moore prédit de manière fiable, depuis 1965, l'évolution de la puissance des microprocesseurs. Si celle-ci continue bien de doubler tous les ans conformément à ce modèle, les moyens pour parvenir à ces fins sont en train d'évoluer. Jusqu'à la fin du dernier millénaire, l'augmentation de la fréquence était l'un des facteurs principaux qui permettait aux processeurs d'atteindre des puissances de calcul toujours plus élevées. Cependant cette course commence aujourd'hui à s'essouffler car des limites technologiques sont en passe d'être atteintes. La miniaturisation des composants a en effet permis d'intégrer de plus en plus de fonctionnalités dans les circuits, mais des problèmes de consommation et d'évacuation de la chaleur engendrées par de hautes fréquences rendent les avancées sur ce terrain plus difficiles.

C'est à présent un autre défi qui attend les concepteurs de circuits et les programmeurs de ces circuits. Les progrès en termes de puissance de calcul s'obtiennent aujourd'hui principalement par l'augmentation du nombre de cœurs à l'intérieur d'une même unité de calcul. En témoignent les choix architecturaux pour le processeur Cell d'IBM/Sony/Toshiba [KDH+05] (un PowerPC et 8 unités de calcul spécialisées), ou la famille de cartes graphiques GeForce de NVidia (avec plusieurs centaines de cœurs CUDA). La multiplication du nombre de cœurs de calcul ne permettrait cependant pas d'atteindre les objectifs de puissance de calcul sans une infrastructure de communication et un système de mémoires efficaces mais complexes à maîtriser d'un point de vue logiciel.

L'ITRS (voir Figure1) prévoit 250 éléments de calcul sur une puce en 2015 pour les appareils électroniques portables [ITR07]. Il faut donc s'attendre à ce que les systèmes sur puce (SoCs) intègrent un nombre considérable d'unités de calcul dans un futur proche. Dans ce contexte, nous sommes amenés à nous intéresser au monde du calcul de haute performance (HPC), dont les intérêts ne sont finalement plus si éloignés de ceux du monde des SoCs. En effet, une architecture performante du top500 [HPC], le *Roadrunner* d'IBM, a franchi officiellement en 2009 le seuil du petaflop (10^{15} opérations flottantes par seconde) avec plus de 20000 cœurs de calcul. Celle-ci est basée sur le processeur Cell, processeur de le Playstation3, preuve que nous arrivons à une convergence dans l'architecture des HPC et des systèmes sur puce multiprocesseurs.

La volonté nette de réduction de consommation d'énergie se manifeste aujourd'hui aussi bien dans le monde des SoCs (pour préserver la durée de vie de la batterie) que dans celui du HPC : en témoigne l'introduction du critère de consommation comme paramètre de la performance des supercalculateurs, où ce qui compte n'est plus le Flop/s (Opération flottante par seconde), mais le Flop/s/W (Opération flottante par seconde et par Watt consommé) [cFC07]. Dans les deux cas, les systèmes sur Puce multi-processeurs hétérogènes (HMPSoCs) ont bien souvent fourni une solution pour garantir une puissance de calcul élevée tout en offrant une consommation d'énergie réduite.

Les HMPSoCs, dont plusieurs modèles sont déjà proposés par les acteurs majeurs du marché des semiconducteurs, ont rencontré un succès incontestable dans le domaine de l'électronique portable du marché

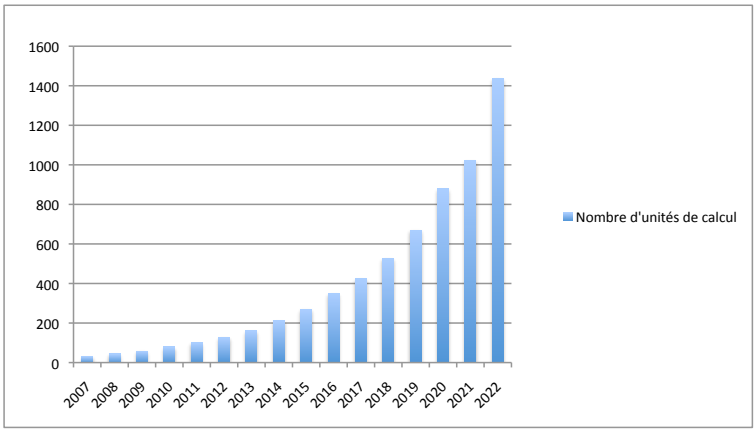

FIGURE 1 – Tendance des systèmes sur puce pour électronique portable grand public

grand public (applications audio, PDA, etc.). Nous pouvons citer quelques plateformes largement utilisées en industrie, comme OMAP [Tex], Nexperia [NXP], la plateforme Nomadik [ST] ou le Diopsis 940HF [Atm] d'Atmel. Elles combinent toutes des processeurs de natures différentes (RISC(s) et VLIW(s)), une infrastructure de communication efficace et une hiérarchie mémoire spécifique pour répondre aux besoins de puissance de calcul et de temps de mise sur le marché de ces applications. Cependant, ces systèmes sont très complexes à programmer et nécessitent un environnement de conception du logiciel [GPY+07].

Pour satisfaire les besoins en puissance de calcul des applications du monde du calcul de haute performance, il est nécessaire de connecter plusieurs de ces HMPSoCs hétérogènes (ou tuiles) par un réseau de connexion spécifique : nous obtenons alors un système multi-tuiles. Les environnements de conception du logiciel existants pour la tuile de base doivent alors être adaptés pour exploiter ces systèmes de manière optimale.

Nous pouvons à présent toucher au cœur de cette thèse : les applications (grand public et du HPC) et les méthodes de calculs sont aujourd'hui arrivées à maturité. Nous souhaitons donc proposer un flot qui se base sur une représentation abstraite de l'application, indépendante de toute plateforme d'exécution. Le flot repose également sur une représentation à haut niveau de l'architecture pour générer un binaire par processeur de la plateforme multi-tuiles cible, ce de manière automatique.

Nous verrons que l'intégration de systèmes complexes avec ce type de flot nécessite des méthodes d'automatisation, de génération et de synthèse de logiciel embarqué, et que les systèmes multi-tuiles introduisent des difficultés nouvelles par rapport à un système basé sur une seule tuile : la synthèse des communications en logiciel et la gestion automatique des mémoires du système sont les deux contributions principales de cette thèse.

Nous nous proposons donc d'étudier un environnement de conception de logiciel embarqué pour une architecture multi-tuiles, sans hypothèse sur le nombre de processeurs ou leur nature. L'étude se concentre sur les communications entre processeurs, et l'approche qui a été retenue pour masquer les détails de l'architecture au programmeur. Ceci impose quelques hypothèses sur le modèle en entrée, le matériel considéré et les outils de compilation associés, qui seront explicités et justifiés dans les premières parties de la thèse. Après ce tour d'horizon, nous présenterons nos contributions ainsi que les perspectives.

Cette thèse est organisée comme suit :

– Chapitre I : Nous passons en revue les différentes architectures parallèles existantes pour les MPSoCs. Nous verrons que chaque catégorie d'architecture peut être programmée selon différents paradigmes, et nous tenterons au fur et à mesure de dégager les raisons pour lesquelles nous ne pouvons nous contenter de ces approches pour nos fins de génération de logiciel ciblant des systèmes multi-tuiles hétérogènes. Cependant, nous nous pencherons sur les interfaces offertes au programmeur de l'application dans ces différents environnements, afin de souligner la manière dont les communications lui sont présentées.

– Chapitre II : Après une revue des paradigmes de programmation pour les différents types d'architectures parallèles, nous introduisons les environnement à base de génération de logiciel prenant en entrée un modèle de haut niveau, et les comparons aux environnements embarqués décrits au Chapitre I. Appliquer un environnement de génération de logiciel à des systèmes multi-tuiles introduit un certain nombre de problèmes, aussi bien pour définir le contenu du modèle que pour produire des binaires fonctionnels. Nos contributions répondent à quelques unes de ces problématiques, en apportant des réponses sur l'un des aspects les plus délicats introduits par les systèmes multi-tuiles hétérogènes : la gestion de communications entre processeurs.

– Chapitre III : Nous nous intéressons dans ce chapitre à tous les concepts qu'un flot de génération de logiciel doit manipuler. Ceci va du modèle de calcul adopté pour la description de haut niveau jusqu'à la structure des binaires générés, en passant par les technologies de compilateurs existantes pour les différents types de processeurs. Nous parviendrons ainsi à dégager les méthodes existantes que nous pouvons utiliser pour maîtriser l'hétérogénéité de la plateforme au niveau d'un processeur donné et de son proche environnement. Des réponses restent à apporter sur les communications entre processeurs a priori différents, qui introduisent une dépendance entre ceux-ci, dépendance qui doit se refléter dans leur binaire.

– Chapitre IV : Nous formalisons les trois composants de notre modèle de haut niveau, à savoir l'architecture, l'application et le déploiement des éléments applicatifs sur les éléments de l'architecture. Nous introduisons dans le fichier d'architecture des chemins de communication sur lesquels les canaux de communication du modèle d'application vont pouvoir être déployés. Nous décrivons ensuite notre modèle de pile logicielle, et la manière dont les communications entre tâches sont implantées de manière transparente pour le programmeur. L'introduction du modèle de haut niveau d'une part, de la pile logicielle d'autre part nous permettent d'introduire notre environnement de génération dont le but finalement est de combler le grand fossé qui sépare ces deux modèles. La phase de génération de logicielle n'étant pas suffisante pour y parvenir, notre flot comporte une phase de sélection de composant logiciels à puiser dans différents types de bibliothèques. Nous étudierons la structure des composants logiciels de communication, qui se présentent sous la forme de pilotes du système d'exploitation.

– Chapitre V : Le flot est mis en œuvre sous forme d'outils qui permettent effectivement la génération

d'un binaire par processeur de la plateforme depuis le modèle de haut niveau. Ces outils se basent sur une génération semi-automatique des configurations requises par le flot (par exemple la spécialisation des composants logiciels), et doivent manipuler les scripts d'édition de liens des compilateurs pour placer les éléments de la pile logicielle dans les mémoires de la plateforme.

– Chapitre VI : Nous présentons quelques expérimentations menées principalement sur une plateforme huit tuiles basée sur un Diopsis 940. Nous détaillons quelques caractéristiques de la pile logicielle pour l'ARM et le DSP d'une tuile, Nous donnons ensuite quelques métriques de performance pour les mécanismes de communication introduits en Chapitre V, avant de présenter trois applications industrielles et scientifiques non triviales qui ont été passées avec succès à travers notre flot de génération : le LQCD (Lattice Quantum Chromo-Dynamics), le WFS (Wave Field Synthesis) et une application d'ultrasons pour imagerie médicale.

Chapitre 1

Etat de l'art sur les communications dans les MPSoCs

Sommaire

Introduction

Depuis le début de l'informatique, les besoins en puissance de calcul ne cessent d'augmenter. De nombreuses applications de physique fondamentale, de traitement d'images, de la dynamique des fluides, nécessitent une puissance de calcul toujours plus élevée. Les machines séquentielles ne fournissent pas la puissance de calcul nécessaire à ces applications et le recours à une architecture parallèle est donc requis.

Il existe différentes classifications des machines parallèles. La première a été faite par Flynn en 1972 [Fly72]. Elle se base sur le flux de données et le flux d'instructions de la machine :

- SISD (Single Instruction, Single Data) : Il s'agit d'une machine qui exécute une instruction à la fois (sur un seul processeur), chaque instruction ne travaille que sur une portion précise du flot de donnée.
- SIMD (Single Instruction, Multiple Data) : Ce modèle consiste en un processeur dont les instructions sont vectorielles, c'est-à-dire que chaque instruction permet de traiter plusieurs portions du flot de données en même temps. Les processeurs vectoriels et les unités de calcul graphiques (GPU) rentrent dans cette catégorie.
- MISD (Multiple Instruction, Single Data) : Ce modèle d'architecture est plus rare en pratique, plusieurs processeurs travaillent en même temps sur la même portion du flux de données.
- MIMD (Multiple Instruction, Multiple Data, voir Figure1.1) : Il s'agit d'architectures avec plusieurs processeurs qui travaillent chacun sur une portion du flot de données qui leur est dévolu.

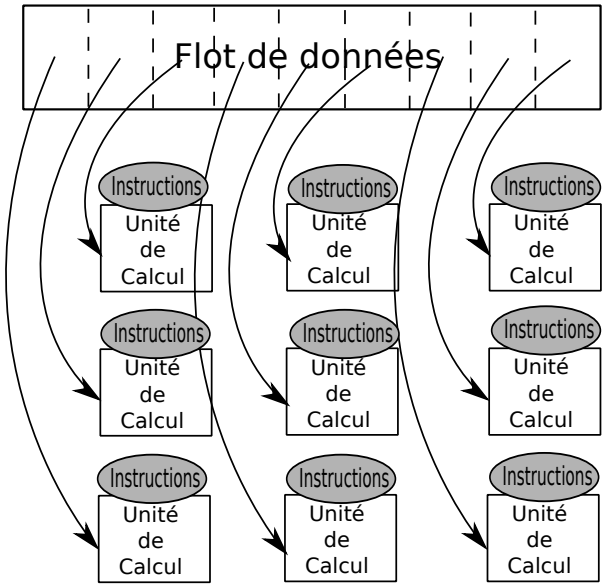

FIGURE 1.1 – Schéma de fonctionnement des machines parallèles de type MIMD

Si nous rentrons plus dans les détails de ce dernier type de machines, il existe deux types de MIMD :

- La MIMD à mémoire partagée : les processeurs peuvent communiquer via une mémoire globale et disposent en général d'un seul système d'exploitation (SE) qui gère la totalité des processeurs.
- La MIMD à mémoire distribuée : chaque processeur a son propre SE, les processeurs effectuent les calculs sur leurs données, puis communiquent leur résultat à la fin du calcul. Les communications s'effectuent via une infrastructure matérielle de synchronisation et d'acheminement des données, qui doit être explicitement gérée par le logiciel.

Nous nous intéressons à un nouveau type d'architecture parallèle rentrant dans la catégorie des MIMD à mémoire distribuée. Cette architecture, mono- ou multi-puces, est basée sur un MPSoC hétérogène (appelé tuile), répliqué et interconnecté par un réseau de communication inter-tuiles. Le niveau de granularité le plus fin est donc un MPSoC hétérogène. Nous nous attachons dans cette partie à faire un inventaire des architectures et méthodes de programmation existantes dans le monde des MPSoC, qui permettent de masquer les détails de l'architecture au programmeur. Ceci nous permettra de mieux comprendre les enjeux et difficultés de programmation de la tuile de base, et nous pourrons alors décrire les architectures multi-tuiles et présenter une architecture sur laquelle nous avons basé nos premiers travaux.

1.1 Architecture des MPSoCs

1.1.1 Vue schématique d'un MPSoC

La Figure1.2 donne une vue schématique de l'architecture d'un MPSoC. Celle-ci est composée de plusieurs sous-systèmes interconnectés par une structure de communication interne (par exemple, un bus, un commutateur ou un Réseau sur Puce).

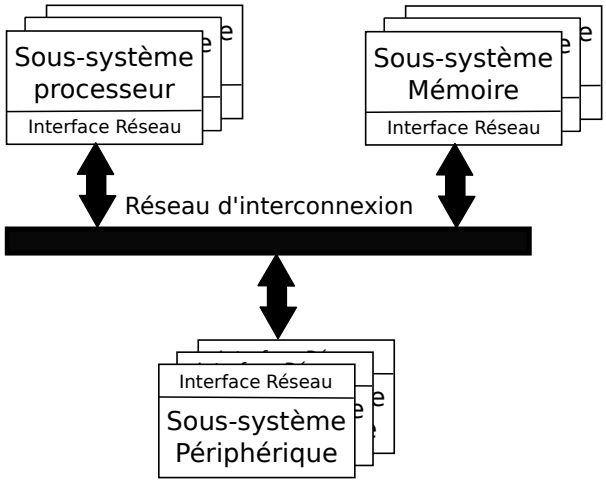

FIGURE 1.2 – Vue schématique d'un MPSoC

Il existe trois types de sous-systèmes :

- Les sous-systèmes dédiés au calcul, tout d'abord. Ceux-ci sont constitués d'une ou plusieurs unités de calculs, de même type ou de types différents, et d'une ou plusieurs mémoires locales, de taille réduite en général.
- Les sous-systèmes de type mémoire, qui fournissent un espace mémoire partagé au sein du MPSoC, sous forme d'une mémoire embarquée ou d'une interface vers une mémoire externe.
- Les sous-systèmes périphériques, qui permettent de sortir du MPSoC, pour communiquer avec le monde extérieur, avec d'autres MPSoCs ou un matériel de stockage par exemple.

Chacun de ces sous-systèmes peut accéder ou être accédé par la structure de communication du MPSoC par le biais d'une interface réseau, qui peut être un simple pont qui fait le lien avec le bus local au sous-système, ou alors une structure basée sur des accélérateurs matériels. Le DMA (Direct Memory Access), par exemple, est souvent utilisé pour décharger les unités de calcul du poids que représente la communication sur le réseau de connexion, des unités de calcul de type DSP par exemple pour lesquelles une telle opération pénalise fortement les performances.

1.1.2 Architectures et blocs matériels remarquables dans les MPSoCs

Lorsque le sous-système de calcul est composé de plusieurs processeurs de même type, nous parlons d'un système SMP (Symmetric Multi-Processor). Chaque processeur, comme le nom de l'architecture l'in-

dique, a accès aux mêmes infrastructures du MPSoC par le biais de la même infrastructure de communication matérielle. Lorsque le MPSoC comporte deux ou plusieurs sous-systèmes de structures internes différentes (unités de calcul et matériel associé aux communications) nous parlons d'un MPSoC hétérogène. L'hétérogénéité peut concerner aussi bien la nature des unités de calculs que le chemin matériel de communication, c'est-à-dire les blocs matériels à traverser pour atteindre un élément du SoC que nous introduirons en Chapitre III.

Parlons à présent de la hiérarchie mémoire d'un MPSoC. Une tendance nette dans ce domaine est d'avoir un système mémoire hiérarchisé, avec des tailles de mémoire et des temps d'accès qui varient inversement, la mémoire de plus grande taille étant en général une mémoire externe (SDRAM, ...) à coût d'accès élevé. Une telle architecture est appelée NUMA (Non Uniform Memory Access). Certaines infrastructures matérielles permettent de décharger les unités de calcul de la gestion de cette hiérarchie. L'exemple le plus répandu est un système de caches, qui permet de rapprocher les blocs d'instructions et de données utilisés dans des mémoires à faible temps d'accès pour le(s) processeur(s). Cependant ce système a un coût non négligeable en termes de silicium et de consommation [ZKKC03]. Un autre mécanisme présent dans certains systèmes sur puce est la TCM (Tightly Coupled Memory), une mémoire embarquée (instruction et/ou données) à disposition d'une et seulement une unité de calcul via un lien séparé du bus de communication. Cela assure un temps d'accès à la mémoire constant et performant, le système de caches est découplé du système TCM. Ces mémoires sont généralement utilisées pour des applications temps réel, pour traiter des interruptions en un temps déterminé.

Lorsque les architectures NUMA n'ont pas de système de gestion mémoire en matériel, nous parlons de EMM (Explicit Memory Management), où le logiciel doit se charger de gérer l'emplacement des données et des instructions du programme, afin d'assurer son bon fonctionnement et d'optimiser le placement de ces éléments [SYN09].

1.2 Environnements de programmation des MPSoCs

1.2.1 Programmation de zéro

Plusieurs approches de programmation existent aujourd'hui. Une approche qui consiste a simplement écrire du code applicatif et le logiciel de bas niveau pour chaque sous-système de calcul est envisageable pour une application très simple, mais devient vite irréalisable pour des systèmes multi-tuiles plus complexes, ou non souhaitables dans la mesure où la réutilisabilité du code applicatif est un critère très important dans le monde des SoCs. L'exemple suivant illustre les opérations à effectuer sur un DSP pourvu d'un DMA très simple avec seulement cinq registres, pour déposer le message *"Bons baisers de DSP land"* dans une mémoire partagée, en gardant à l'esprit que le RISC attend un signal dans cette mémoire et qu'il faut donc pourvoir le code d'une zone critique protégée par un verrou matériel (le même message est déposé à chaque itération d'une boucle infinie pour simplifier l'exemple).

Cet exemple simple atteint déjà un niveau de complexité considérable, nous pouvons voir apparaître un protocole pour la programmation du DMA, l'utilisation d'un verrou matériel pour une exécution en exclusion mutuelle entre le RISC et le DSP et un protocole de communication de type poignée de main au niveau logiciel entre les deux programmes (par un champ de synchronisation).

```
// Parametres de la plateforme materielle
definir ADRESSE_MEMOIRE_PARTAGEE=0x400100
definir ADRESSE_VERROU_MATERIEL =0x702000
definir ADRESSE_BASE_DMA        =0x10003400

// Structures de donnees
structure registres_dma {
    uint32_t adresse source;       //ecriture
    uint32_t adresse destination;  //ecriture
    uint32_t adresse taille;       //eriture
    uint32_t lancer_operation;     //ecriture/lecture
} à l'adresse ADRESSE_BASE_DMA;

structure vue_memoire  {
    uint8 synchronisation;    // 0: Zone mémoire appartient au DSP
                              // 1: Zone mémoire appartient à l'ARM
    string message;
} à l'adresse ADRESSE_MEMOIRE_PARTAGEE.


// Implantation du protocole avec le DMA
// Transfert de noctets octets de adresse_src a adresse_dst
fonction transfert_dma (adresse_src, adresse_dst, noctets)
debut
  //Premiere etape : preparation du DMA
   registres_dma->fini := 0;

  //Deuxieme etape : programmation du DMA
   registres_dma->source      := adresse_src;
   registres_dma->destination := adresse_dst;
   registres_dma->taille      := noctets;

   // Troisieme  etape : lancement de l'operation
   registres_dma->lancer_operation := 1;

   // Quatrieme etape : attente de la completion par scrutation
   tant que (registres_dma->lancer_operation=1)
        nil;
   fin tant que;
fin fonction

// Prise et liberation  d'un verrou materiel
fonction prendre_verrou ()
debut
    // Test and set en langage assembleur VLIW!
    assembleur(TSET ADRESSE_VERROU_MATERIEL);
fin fonction

fonction liberer_verrou ()
debut
    contenu(ADRESSE_VERROU_MATERIEL) := 0;
fin fonction



// Programme principal : deposer en boucle le message en memoire partagee
programme principal :
    string mes := "Bons baisers de DSP land";
    uint8 sync;
debut
    boucle infinie :
        // Debut de la zone critique
         prendre_verrou();
        // Attendre que la structure en memoire appartienne au DSP
        faire
            transfert_dma(vue_memoire->synchronisation,adr(sync),1);
        tant que (sync=1);

        // Lancer le transfert du message
        transfert_dma(adr(mes),vue_memoire->message,taille(mes));
```

```
        // Ceder la structure partagee vue_memoire
        sync=1;
        transfert_dma(vue_memoire->synchronisation, adr(sync), 1);

       // Attendre que la structure en memoire appartienne au DSP
        faire
            transfert_dma(vue_memoire->synchronisation,adr(sync),1);
        tant que (sync=1);

        // Fin de la zone critique
        liberer_verrou();

    fin boucle
fin programme
```

Cet exemple fait bien apparaître des dépendances par rapport à l'environnement du processeur (programmation de DMA, verrou matériel) et par rapport au protocole de communication choisi (poignée de main). Il ne fait pas ressortir les différences dans les calculs et dans le traitement pour les entiers ou les réels. Le RISC, dont le code n'est pas montré ici, est lié au programme du DSP par l'adresse qui a été choisie pour stocker la structure partagée, et par l'adresse du verrou utilisé pour l'exclusion mutuelle. Nous traitons ici des chaînes d'octets, mais une conversion d'endianness peut être nécessaire pour que le DSP et le RISC puissent interpréter de la même manière ces types. Enfin, nous pouvons faire remarquer qu'une scrutation a été utilisée pour détecter la fin du transfert du DMA, une interruption aurait pu être utilisée mais aurait considérablement compliqué l'exemple. En conclusion, le code montré ci-dessus est complexe en dépit de la simplicité de l'exemple, source d'erreur car il doit s'accorder avec les informations contenues dans le code du RISC, et hautement non portable.

1.2.2 Environnements pour les systèmes SMP

Plusieurs paradigmes de programmation existants permettent une écriture formalisée de l'application et masquent les aspects de bas niveau au programmeur. Nous commençons par la programmation des systèmes SMP. Tous les processeurs exécutent un unique système d'exploitation, qui se charge d'ordonnancer les processus de l'application sur chacun des processeurs. Ceci donne donc une illusion d'accélération, car les processus tournent bien en même temps sur des processeurs distincts. Cette approche présente deux inconvénients : tout d'abord, une application écrite séquentiellement ne sera pas accélérée par cette approche. L'application devra donc être parallélisée pour tirer bénéfice du système. De plus, cette approche présente le risque de saturation des accès mémoire si le nombre de tâches qui s'exécutent en même temps sur les différents processeurs est trop élevé. Le système d'exploitation et son ordonnanceur en particulier doivent mettre en œuvre cette fonctionnalité. La manière dont les tâches sont réparties dans les processeurs symétriques, et dont les accès aux ressources du système sont contrôlés (synchronisations ...) est généralement laissé à la discrétion du système d'exploitation. Le compilateur n'est pas mis à contribution dans cette approche, contrairement à celle que nous décrivons ci-après.

Une approche complémentaire est l'approche de openMP qui s'appuie sur des directives de compilation (des pragma pour le langage C par exemple) qui rendent possible le déroulement d'une boucle de calcul tout en gardant une approche de programmation séquentielle. Seules les portions de calcul répétitives sont donc parallélisées et des commandes de synchronisation entre les tâches sont nécessaires à l'intérieur de ces portions parallèles, comme indiqué ci-après.

```
/* La clause parallel indique que le code
  entre accolades doit être  effectué en parallèle
  Dans cet exemple, le compilateur répartit les éléments
  de la boucle comme il l'entend
 */
#pragma omp parallel {

/* La clause for nowait indique une boucle répartie
   sur plusieurs processeurs, sans synchronisation
  à la fin de la boucle
*/
#pragma omp for nowait
for(i = 1; i <= maxi; i + +)
    a[i] = b[i];

/* La clause for  indique une boucle répartie
   sur plusieurs processeurs avec un point d'attente
   entre tous les processeurs à la fin de leur boucle
*/
#pragma omp for
for(j = 1; j <= maxj; j + +)
    c[j] = d[j];
}
```

OpenMP nécessite donc à la fois le concours du compilateur pour interpréter les directives de compilation, et le concours du système d'exploitation pour assigner chaque tâche de calcul sur les différents processeurs et implanter les synchronisations nécessaires au bon fonctionnement du programme. Cette approche facilite l'écriture d'un programme séquentiel sur des systèmes à mémoire partagée en masquant les communications entre processeurs et les synchronisations.

Il est intéressant de noter que les communications entre processeurs ne sont pas explicites dans ces deux approches. En effet, le programmeur place ses processus (ou ses éléments de calcul pour openMP) dans les différents processeurs, le système d'exploitation embarqué met en œuvre les synchronisations et gère la mémoire partagée entre processeurs comme il l'entend.

1.2.3 Les interfaces de programmation à base de passage de messages

MPI

Les interfaces de programmation à base de passage de messages, comme leur nom l'indique, permettent un échange de messages entre deux processus par exemple. Ceci sous-entend que l'information est transférée d'un processus écrivain vers un processus récepteur, mécanisme qui nécessite des gestions de synchronisation explicites pour les communications, contrairement aux approches à mémoire partagée. Ces interfaces sont particulièrement adaptées aux systèmes à mémoire distribuée, mais peuvent aussi s'appliquer aux systèmes à mémoire partagée, même si dans la pratique les performances des bibliothèques à base de passage de messages sont moins intéressantes que les méthodes de programmation pour SMP.

Les environnements présentant une interface de type passage de message les plus évolués et complets sont à chercher dans le monde des supercalculateurs, avec l'incontournable interface de programmation MPI (Message Passing Interface). Cette interface dispose d'une grande variété de primitives pour l'envoi et la réception de messages. La Table 1.1 décrit les huit fonctions SEND (qui ont un RECEIVE équivalent) de

MPI, avec quatre modes de fonctionnement : standard (MPI choisit un mécanisme par défaut), synchrone (l'émetteur et le récepteur s'attendent pour échanger leurs données), ready (l'appel au processus émetteur est effectué avant l'appel au processus récepteur, sinon le résultat est indéfini) et *bufferisé* (l'utilisateur définit lui-même les caractéristiques du tampon intermédiaire). Ces quatre modes peuvent être soit bloquants, c'est-à-dire que les fonctions SEND et RECEIVE ne redonnent le contrôle au programme que lorsque l'opération est terminée, soit non bloquants, c'est alors le programmeur qui doit tester l'état de la communication pour vérifier sa complétion. Les mécanismes non bloquants ont l'avantage de permettre un recouvrement du temps de calcul par les communications.

Différents modes du SEND dans MPI		
Mode du SEND	**Bloquant**	**Non Bloquant**
Standard	MPI_SEND	MPI_ISEND
Synchrone	MPI_SSEND	MPI_ISSEND
Ready	MPI_RSEND	MPI_IRSEND
Bufferisé	MPI_BSEND	MPI_IBSEND

TABLE 1.1 – Huit fonctions SEND dans MPI

MPI fournit en outre une grande quantité de fonctionnalités, comme la définition de communicateurs pour les liaisons multicast, la gestion dynamiques des processus, et la communication directe de mémoire à mémoire (RDMA, pour Remote Direct Memory Access) dont nous reparlerons ultérieurement. MPI offre une interface complète (et donc très lourde) pour assurer la portabilité de l'application sur de nombreuses plateformes. Il existe plusieurs implantations de MPI, libres (LAM, MPICH) ou propriétaires (Fujitsu, Nec, IBM) pour diverses architectures de supercalculateurs. Tous les processeurs du système exécutent le même code, les processeurs ont cependant la possibilité de connaître leur identité dans le système (le rang) et peuvent ainsi exécuter du code différent. MPI est donc conçu pour des architectures massivement parallèles homogènes. L'une des différences notables avec les approches à mémoire partagée, est que les communications sont explicites pour le programmeur. Il définit un communicateur, et lance une primitive de communication de son choix.

```
programme simple_send_receive
    int myrank,ierr,status(MPI_STATUS_SIZE)
    float a(100)
debut

// Initialisation de MPI
    MPI_INIT(ierr)

// Recupere mon rang
    MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)

// Le processus 0 envoie un message de 100 mots avec le tag17, le processus 1 reçoit
    si (myrank = 0) alors
       MPI_SEND(a,100,MPI_REAL,1,17,MPI_COMM_WORLD,ierr)
    sinon si (myrank = 1) alors
       MPI_RECV(a,100,MPI_REAL,0,17, MPI_COMM_WORLD,status,ierr)
    fin si

// Termine MPI
    MPI_FINALIZE (ierr)

fin programme
```

Notons que MPI est de facto l'interface de programmation des applications HPC livrée avec les super-

calculateurs du marché. Une approche efficace pour les systèmes à nœud de calcul homogène consiste à employer localement openMP au niveau du nœud de calcul et MPI au niveau du système complet, cette approche hybride a longtemps été une solution pour atteindre des puissances de calcul élevées, mais devient difficile à appliquer avec l'hétérogénéité des nœuds de calcul (introduction du Cell pour les calculs, par exemple).

MCAPI

MCAPI [HAB+09] (Multicore Association Communication API) tire son héritage de MPI et des sockets BSD (Berkeley Software Distribution), la différence principale est que MCAPI se spécialise dans le traitement de flots de données et de communications entre nœuds de calculs appartenant à la même puce, et non à des stations de travails ou nœuds de calculs distribués. Pour cette raison, l'interface offerte par MCAPI est une interface de communication qui introduit des protocoles de communication efficaces basés sur des mémoires locales ou partagées, présentant une faible latence et une empreinte mémoire réduite. En contrepartie, MCAPI est moins flexible que MPI ou autres environnements plus complets. Notons également que MCAPI, contrairement à MPI et aux *pthreads*, n'offre pas d'interface pour paralléliser son application et gérer les synchronisations pour les communications entre les tâches. L'unité de base de MCAPI est un nœud, qui dispose d'un ensemble de points terminaux (*endpoints*) qui lui permettent de communiquer avec les autres nœuds. Ces points terminaux, à l'instar des sockets, peuvent être identifiés par un port et utilisent des FIFO unidirectionnelles point à point de trois types : non connecté (pas de contrôle sur le flot de données), connecté commuté par paquet et connecté scalaire (où le mot est l'unité d'envoi). La topologie des nœuds est fixée dans l'implantation de MCAPI pour une plateforme multi-coeurs donnée, le programmeur n'a donc pas à se soucier de l'unité d'exécution allouée au nœud lorsqu'il écrit son application.

Si nous résumons, MCAPI se concentre sur des architectures multi-coeurs homogènes ou hétérogènes, mais n'aborde pas le problème des communications extra- puce. Il n'utilise pas les services d'un environnement d'exécution comme un système d'exploitation pour paralléliser l'application de calcul au sein d'une unité de calcul, et le déploiement des éléments applicatifs dans les différents éléments de l'architecture n'est pas ouvert au programmeur.

1.3 L'approche coprocesseur

Un coprocesseur est un processeur complémentaire d'un processeur principal spécialisé dans l'exécution plus performante d'un jeu d'instructions. Cette approche nécessite donc au minimum deux processeurs sur la puce, de préférence de natures différentes, mais ceci n'est pas une obligation (le supercalculateur BlueGene d'IBM possède un nœud de calcul à base de deux PowerPC, dont l'un peut être utilisé en mode coprocesseur [Ma05]). D'un point de vue logiciel, le processeur principal (par exemple un ARM) exécute l'application de manière séquentielle et lance un programme ou une série d'instructions chez le coprocesseur. Cette approche nécessite un système d'exploitation tournant sur le processeur principal, et une bibliothèque proposant une interface de contrôle du coprocesseur au processeur principal. Le programme ci-dessous montre une approche de programmation possible pour le Diopsis940 d'Atmel, qui permet à l'ARM de lancer un programme préalablement compilé sur le DSP de la plateforme (le mAgicV).

```
int main()
{
   //ouvrir le coprocesseur mAgicV
   int fd = mAgicV_open();
   //activer l'horloge du  mAgicV
   mAgicV_enable_disable_clk(MAGICV_CLKEN,fd);
   //charger le programme du mAgicV
   mAgicV_intload_PM("progmem.bin",fd);
   //charger les données dans la mémoire donnée du  mAgicV
   mAgicV_load_DM("datamem.bin",fd);
   //réinitialiser le status du  mAgicV
   mAgicV_reset(fd);
   //lancer le programme du  mAgicV
   mAgicV_start(fd);
   //Attendre que le programme du mAgicV finisse
   mAgicV_wait_for_halt(fd);
   //désactiver l'horloge du  mAgicV
   mAgicV_enable_disable_clk(MAGICV_CLKDIS,fd);
   //fermer le coprocesseur du mAgicV
   mAgicV_close(fd);
   //print hello world
  printf("hello world from ARM\n");
   return 0;
}
```

Comme nous pouvons le voir, cet exemple implique qu'un programme et un ensemble de données soient fournit à l'avance au programme tournant sur le processeur principal (en l'occurence un ARM9), c'est ensuite la bibliothèque qui donne les instructions de bas niveau pour transférer et récupérer les données traitées par le coprocesseur (le mAgicV, dont nous ne fournissons pas le code). Ces instructions, comme nous pouvons le constater, sont très dépendantes de la plateforme visée et laissent peut de place à une généralisation à toutes les plateformes MPSoC. Le processeur principal nécessite un SE muni d'un système de fichiers pour pouvoir charger les différents éléments du programme du mAgicV.

Notons qu'une nouvelle fois, la communication entre le processeur principal et le coprocesseur est laissée à la discrétion de la bibliothèque de contrôle. Il est important de noter, même si cela semble évident, que cette approche dépend fortement des caractéristiques de la plateforme, car n'est pas coprocesseur qui veut. Un processeur doit accepter des commandes de la part de son processeur principal.

1.4 Les GPUs

Nous ouvrons une parenthèse ici pour traiter le cas des GPU , les unités de calcul graphiques qui gagnent en popularité dans la course à la puissance de calcul avec une faible fréquence de fonctionnement (la GeForce 9800 GTX, 420 GFlops, pour 675 MHz seulement). Il faut être conscients cependant que ce sont des unités de calculs spécialisées dans le traitement de larges vecteurs de données, toute autre opération présentera une vitesse d'exécution minime. Il nécessite un hôte (CPU) qui envoie une requête de calcul au GPU pour effectuer un calcul.

Les GPUs présentent une structure complexe et homogène dont les détails sont encore difficilement accessibles ce qui a longtemps rendu (et rend encore) ces architectures difficiles à programmer. Ils sont formés de plusieurs blocs de calculs, chacun composé d'un ensemble de microprocesseurs. Chaque bloc est destiné à l'exécution d'une tâche calculatoire. et peuvent effectuer ces tâche dépendamment ou indépendamment des autres blocs.

1.4.1 CUDA

CUDA (Compute Unified Device Architecture) fournit un ensemble de bibliothèques logicielles, un environnement d'exécution et un ensemble de pilotes pour plusieurs langages de programmation (C, C++, Fortran). C'est également un langage, dérivé du C, fourni avec son compilateur, supportant 9 nouveaux mots clés, 24 nouveaux types et 62 nouvelles fonctions. Un court exemple de code est fourni ci-dessous, il décrit l'allocation par l'hôte de la mémoire nécessaire et le lancement de la fonction de calcul qui sera assignée par le compilateur à un bloc de la GPU. La fonction *cudaMemcpy* constitue dans cet exemple la seule primitive qui induit un transfert de données entre la mémoire de l'hôte et une mémoire accessible par les blocs qui vont effectuer la tâche.

```
// Calcul matriciel C=A.B
void MatrixMulOnDevice(float * A, float * B, float * C, int Width)
{
    //calcul de la taille des matrices
    int size = Width * Width * sizeof(float);

    //allocation des matrices pour les blocs de la GPU  et  remplissage
    cudaMalloc(Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice) ;
    cudaMalloc(Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

    //allocation de la matrice de résultat
    cudaMalloc(Cd, size);

    //multiplication d'une seule matrice
    dim3 dimGrid(1, 1);
    //matrice carrée
    dim3 dimBlock(Width, Width);

    //produit matriciel proprement dit
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Ad, Bd, Cd, Width);

    //récupération du résultat du calcul
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

    //destruction des matrices, désormais inutilisées
    cudaFree(Md);
    cudaFree(Nd);
    cudaFree(Pd);
}
```

1.4.2 openCL

Nous pouvons également parler d'openCL, qui contrairement à Cuda n'a pas vocation à être propriétaire et est censé offrir un format portable et exécutable sur tout type de plateforme hétérogène. OpenCL reste tout de même axé sur les plateformes à base d'un processeur hôte et d'un GPU, Nous pouvons remarquer dans le court exemple qui suit que cet environnement permet également de compiler du code en cours d'exécution sur le processeur hôte, et de lancer le programme fraîchement compilé sur le GPU cible.

```
//Programme à exécuter sur le GPU sous forme d'une chaine de caracteres
          string vecSum = @"
                     __kernel void
                    floatVectorSum(__global       float * v1,
```

```
                    __global       float * v2)
        {
            // Vector element index
            int i = get_global_id(0);
            v1[i] = v1[i] + v2[i];
        }";

    //Initialisation des plateformes et des peripheriques
    OpenCLTemplate.CLCalc.InitCL();

    //Compilation du code source
    OpenCLTemplate.CLCalc.Program.Compile(new string[] { vecSum });

    //Accès au coeur de calcul pour l'hote
    OpenCLTemplate.CLCalc.Program.Kernel VectorSum = new OpenCLTemplate.CLCalc.Program.Kernel("
        floatVectorSum");


    //Créer les vecteurs d'entree v1 et v2 (leur initialisation n'est pas montree ici)
    //dans la mémoire du Device
    OpenCLTemplate.CLCalc.Program.Variable varV1 = new OpenCLTemplate.CLCalc.Program.Variable(v1);
    OpenCLTemplate.CLCalc.Program.Variable varV2 = new OpenCLTemplate.CLCalc.Program.Variable(v2);

    //Prepare les arguments qui son passes au coeur de calcul
    OpenCLTemplate.CLCalc.Program.Variable[] args = new OpenCLTemplate.CLCalc.Program.Variable[] {
         varV1, varV2 };

    //Nombre de travailleurs alloues au calcul
    int[] workers = new int[1] { 2000 };

    //Lancer le coeur de calcul
    VectorSum.Execute(args, workers);

    //Copier  varV1 (memoire du GPU) vers la memoire de l'hote v1
    varV1.ReadFromDeviceTo(v1);
```

Même si openCL est une interface de programmation ouverte, le logiciel de bas niveau pour piloter les GPUs et les détails architecturaux des cartes graphiques sont encore peu ou pas diffusés, ce qui rend une programmation à bas niveau difficilement envisageable, c'est pour cette raison que notre approche ne peut prétendre cibler des GPUs aujourd'hui.

1.5 Les systèmes multi-tuiles

1.5.1 Eléments architecturaux

Les systèmes multi-tuiles sont des architectures construites à partir de MPSoCs hétérogènes de base (ou tuiles), répliqués et interconnectés par une infrastructure de communication extensible. Comme illustré dans la Figure 1.3, un système multi-tuiles dispose de plusieurs niveaux de granularité : les tuiles (qui peuvent présenter des variantes), la puce qui peut contenir plusieurs de ces tuiles connectées par une infrastructure de communication interne (un réseau sur puce par exemple), et le système composé des différentes puces interconnectées par un réseau de communication hors de la puce (par exemple un torus 3D).

Les architectures multi-tuiles, comme indiqué en introduction de ce chapitre, sont globalement des MIMD à mémoire distribuée, chaque tuile traite une portion des données qui lui est assignée. Les communications entre tuiles permettent d'échanger des données, synchroniser les tâches ou encore effectuer des

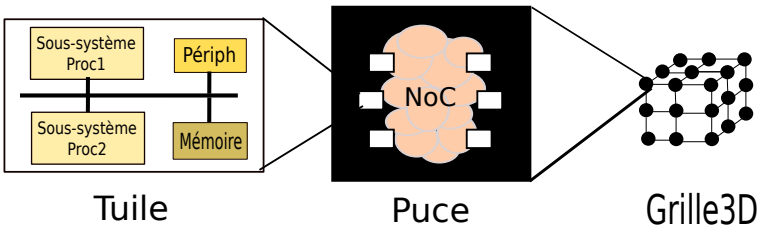

FIGURE 1.3 – Différents niveaux de granularité et exemples de réseaux utilisés dans les systèmes multi-tuiles

opérations de réduction sur un ensemble de tâches (opérations associatives comme l'addition). Il faut garder à l'esprit que les communications inter-tuiles introduisent une pénalité considérable, en particulier lorsque les communications concernent deux puces différentes. D'où l'importance qu'auront les communications dans l'environnement de programmation de ces plateformes, et la manière dont les communications seront mises en œuvre. Ajoutons à cela la structure de mémoire distribuée dans tout le système, pour lequel il n'y a pas de support matériel a priori, nous commençons à présent à comprendre les difficultés qui nous attendent.

1.5.2 Système multi-tuiles basé sur un Diopsis940 d'Atmel (SHAPES)

Tuile de base

Nous nous intéressons à une architecture multi-tuiles basée sur un Diopsis940. Le Diopsis940 (voir Figure 1.4) est un MPSoC hétérogène formé de deux sous-systèmes de calcul ARM et DSP, d'une interface vers une mémoire externe à la puce (en particulier une SDRAM de 64MO) et d'un ensemble de périphériques (USB, liens séries I2S et I2C, interface MAC pour l'Ethernet). Tous ces systèmes sont interconnectés par un bus de type AMBA hiérarchisé, qui permet de soutenir en parallèle les échanges de données entre les différents composants du Diopsis940. Nous ne rentrons pas plus dans les détails de cette architecture complexe, mais de nombreux autres blocs matériels (pour la gestion centralisée des interruptions, des périphériques et des horloges) qui rendent l'écriture du logiciel de bas niveau très complexe.

Le premier sous-système de calcul est le sous système ARM. Il est pourvu d'un ARM9, de 16kO de cache instructions et 16kO de cache données, d'une unité de gestion mémoire (MMU) et d'une interface TCM vers une mémoire embarquée de 48kO (la mémoire peut être configurée comme une mémoire TCM Instruction/Données, ou comme une mémoire normale accessible par le bus).

Le second sous-système de calcul est le sous-système DSP. Il est constitué d'un DSP (le mAgicV d'Atmel) capable d'effectuer jusqu'à dix opérations arithmétiques par cycle (quatre multiplications, trois additions, trois soustractions) le tout sur des entiers de 32 bits ou des flottants sur 40bits. En conséquence, le mAgicV peut opérer à 1 Gops à 100MHz, cela sous-entend toutefois un pipeline logiciel, l'exécution d'opérations de contrôle cassent le pipeline et font diminuer les performances de manière drastique, Le DSP dispose donc d'une taille de donnée de 40 bits et d'une taille d'instruction de 128 bits (d'où la catégorisation du mAgicV comme un VLIW, Very Long Instruction Word). Chaque instruction contrôle cinq éléments de l'architecture : 2 entrées contrôlent un générateur d'adresses permettant de générer quatre

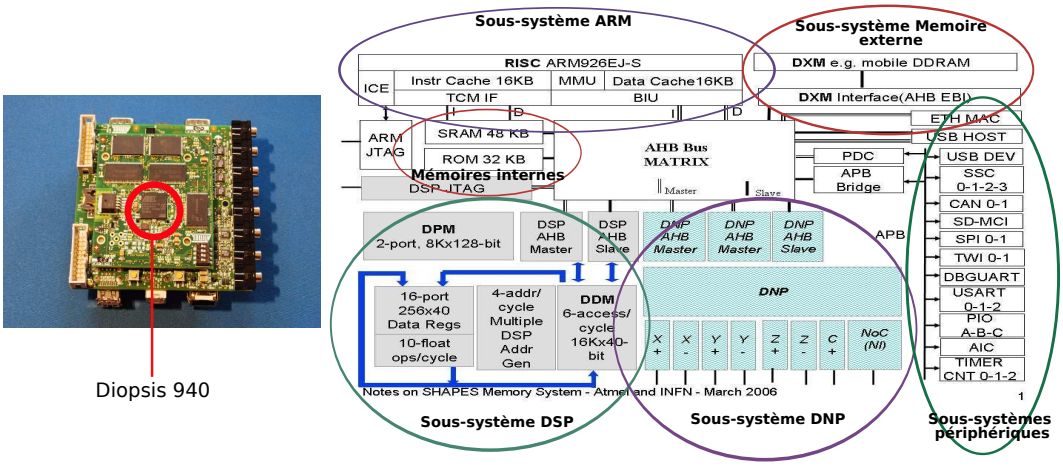

FIGURE 1.4 – Diopsis 940 d'Atmel (à gauche), détails d'une tuile basée sur un Diopsis 940 avec le processeur réseau (à droite)

adresses par cycle. La mémoire données fait 16K * 40 bits, la mémoire instructions seulement 8k * 128 bits. Une unité de gestion du programme (PMU pour Program Management Unit) permet de gérer en matériel un programme situé dans une mémoire externe au sous-système en rapprochant les blocs utilisés dans la mémoire programme. Il est important de souligner l'interface réseau : le DSP ne peut sortir de son sous-système qu'au travers un DMA, qui peut également servir d'interface esclave et est donc programmable de l'extérieur. Une interface esclave permet aussi d'accéder directement aux mémoires internes du DSP. Point intéressant sur ces deux interfaces réseau : le passage des données à 40 bits du sous-système DSP aux 32 bits de la tuile. Un traitement différent est fait suivant le type de la donnée, un décalage à gauche de 8bits pour les entiers, un décalage à droite de 8bits pour les flottants.

Le Diopsis940HF est à l'origine conçu pour le domaine de l'audio, sa plateforme d'évaluation comporte d'ailleurs plusieurs entrées/sorties audio pour l'utilisation dans des synthétiseurs ou autres.

Réseau d'interconnection et processeur réseau

L'architecture multi-tuiles est formée de trois types de réseaux, tous trois commutés par paquets : un réseau sur puce pour interconnecter les tuiles appartenant à la même puce, un réseau de fils courts qui forme un réseau Torus 3D. Un arbre collectif pour les opérations de réduction est aussi prévu. Le réseau de type Torus3D permet de s'affranchir de l'utilisation explicite de routeurs car les messages passent de proche en proche selon un routage en (X,Y,Z) dépendant des interfaces réseau de chaque tuile. Cette topologie est de plus bien adaptée aux applications de physique nécessitant une décomposition spatiale (ou spatio-temporelle). Une interface réseau faisant le lien entre la tuile et ce réseau de communication (composé de NoC, d'un réseau Torus3D et d'un arbre collectif) est donc requise. Le DNP (Decentralized Network Processor) a été conçu par l'INFN (institut national de physique nucléaire de Rome) et profite du savoir-faire de cet institut dans la conception de plusieurs générations de supercalculateurs (APE Array Processor Experiment). Les différentes générations de APE sont reconnues comme des plateformes de référence pour l'exécution du LQCD (Lattice Quantic Chromodynamics). Ces systèmes étaient basés soit sur des VLIWs [AGM+04] soit sur des processeurs généralistes fonctionnant à une fréquence réduite (200MHz pour la dernière génération, le APEmille).

La principale tâche du DNP est de fournir un service de communication inter-tuiles, mais le DNP peut également faire office de DMA pour des communications intra-tuile. Attaché au bus de chaque tuile (de la même manière que les autres périphériques de la tuile), le DNP offre aux unités de calcul (ARM, DSP) une interface Esclave par laquelle il reçoit des commandes, et présente deux interfaces Maître peuvent accéder les mémoires et périphériques de la tuile. Il présente trois types d'interfaces avec le monde extérieur : Torus 3D, Arbre collectif et NoC. Le DNP dégage les unités de calcul de la gestion du réseau commuté par paquets. En particulier, il se charge de la fragmentation, de l'arbitrage, du routage et de la mise en tampon des paquets. Le DNP présente un contrôleur RDMA, capable d'effectuer des opérations mémoire à mémoire à travers une interface qui nécessite un logiciel de bas niveau complexe pour le gérer.

Le système multi-tuiles est composé de différents types de tuiles : RDT (ARM+DSP), DET (DSP) et MET (mémoire seulement). Un simulateur matériel d'un système à huit tuiles RDT connectées par un torus3D 2x2x2, précis au niveau cycle pour les unités de calcul et intégrant un modèle SystemC TLM du DNP, est à disposition et sert de base aux travaux présentés dans cette thèse.

Chapitre 2

Problématiques et contributions de la thèse

Sommaire

Introduction

Les applications et les méthodes de calcul pour les systèmes MPSoC sont aujourd'hui arrivées à maturité. Selon l'ITRS, ce ne sont plus les applications qui doivent s'adapter aux architectures existantes, mais les architectures qui doivent s'adapter aux exigences des applications ; celles-ci demandent toujours plus de puissance de calcul tout en limitant leur consommation d'énergie pour préserver la durée de vie de la batterie. Les circuits intégrés spécifiques à une application (ou ASIC) permettent, pour une application donnée, d'accélérer certaines parties critiques pour remplir le contrat de performance, de consommation ou de temps d'exécution. Cependant ces solutions à base d'ASIC offrent une flexibilité très limitée aux programmeurs de l'application, et une grande pénalité dans le temps de mise sur le marché si l'ASIC n'est pas disponible au début des développements.

Nous nous basons sur des systèmes multiprocesseurs hétérogènes, qui offrent plus de flexibilité tout en garantissant une forte puissance de calcul avec une consommation d'énergie réduite. Cependant ces architectures s'avèrent très complexes à programmer : en effet, l'hétérogénéité de leurs unités de calcul, de leur infrastructure de communication et de leur hiérarchie mémoire rendent la programmation "de zéro" quasiment impossible (en tous cas en un temps raisonnable). Certains environnements de programmation, basés sur des outils de génération de code, ont été proposés pour pallier la complexité de programmation de ces systèmes, et permettre aux programmeurs des applications grand public de respecter la contrainte de temps de mise sur le marché. Ces approches, que nous décrirons en détails dans le Chapitre III se basent sur une description à un haut niveau d'abstraction de l'application.

2.1 Pourquoi un flot de génération ?

Le but d'un environnement de programmation est avant tout de faciliter la conception d'une application parallèle sur une architecture. Comme nous l'avons vu au Chapitre I, de nombreux environnements de programmation (MCAPI, openMP) utilisant des paradigmes de programmation efficaces existent déjà, pourquoi donc vouloir proposer une approche différente ? La première réponse tient au type de plateforme visé par ces flots : les MPSoCs hétérogènes. Ceux-ci embarquent des DSPs qui ne peuvent supporter le poids d'un environnement logiciel complet (mémoires programme et données limitées, baisse dramatique des performances lors de l'exécution de code de contrôle). Cependant, le logiciel bas niveau présent dans ces environnements doit tout de même être mis à disposition. Les outils de génération de logiciel permettent d'embarquer un logiciel de bas niveau plus léger. Le logiciel de bas niveau se présente sous forme de composants logiciels, qui sont sélectionnés en fonction du langage de haut niveau et de la plateforme cible. Ces bibliothèques de composants doivent s'adapter aux exigences des DSP et être aussi légères que possible. Pour un RISC, les bibliothèques logicielles peuvent être plus riches et mettre en œuvre plus de fonctionnalités.

Les approches basées sur des outils de génération de logiciel présentent divers avantages par rapport à un environnement de programmation embarqué comme MPI, sachant que d'autres plateformes homogènes peuvent en bénéficier.

- La représentation haut niveau de l'application est portable sur toute plateforme, pourvu que le flot la supporte.
- La complexité de la plateforme (hétérogénéité, hiérarchie mémoire, endianness) est masquée au programmeur grâce à l'utilisation d'une interface de programmation qui donne accès aux couches logicielles inférieures.
- Le programmeur parallélise son application sans se soucier dans un premier temps de leur répartition sur la plateforme cible. Une fois l'application fonctionnelle, plusieurs essais de déploiement pourront être testés quasiment sans effort grâce à l'utilisation d'une interface au niveau de l'application pour abstraire les éléments de plus bas niveau (les communications par exemple). C'est même une phase d'exploration au niveau système qui pourra proposer en fonction de certains paramètres du modèle de haut niveau quelques solutions pertinentes de déploiement.
- Le code de bas niveau embarqué est allégé par rapport à un environnement de programmation embarqué complet (comme MPI), en partie parce que les outils se basent sur des paramètres fixés par la représentation de haut niveau et non sur des structures ou fonctions embarquées qui gèrent dynamiquement ces paramètres (par exemple, les communicateurs).

– Le modèle de haut niveau peut être soumis à des outils d'aide à la conception de l'application (vérifications formelles, validation au niveau système) pour aider à la conception de l'application.

En revanche, ces approches supposent certaines exigences auxquelles les programmeurs de l'application doivent se plier.

– Le code applicatif doit être portable, c'est-à-dire qu'il ne doit pas par exemple comporter de pointeur sur une adresse en dur, et utiliser une interface spécifique pour les opérations du système d'exploitation et des opérations de communication.
– Le programmeur doit se conformer à une approche de programmation imposée par le langage de haut niveau choisi. Celle-ci nécessite une approche parallèle (comme MCAPI ou MPI), car le programmeur ne sait pas à l'avance si la plateforme sera de type SMP pour utiliser des approches inspirées de openMP où une représentation séquentielle est possible.
– Masquer les détails architecturaux au programmeur est à double tranchant, car écrire un code portable pour toute plateforme et tout type de processeur peut donner un résultat correct mais non optimal. Le travail d'optimisation, une fois le déploiement de l'application sur la plateforme MPSoC effectuée, reste d'ailleurs à la charge du programmeur. Il est important de noter que ce point ne concerne que les tâches applicatives, les couches de bas niveaux mises à dispositions par le flot de génération de logiciel sont disponibles en plusieurs versions pour tout type de processeur et/ou plateforme, chacune étant optimale pour la plateforme considérée.

2.2 Application d'un flot de génération à un système multi-tuiles

Nous souhaiterions garder une approche de programmation basée sur une description de l'application à un haut niveau d'abstraction et cibler des systèmes multi-tuiles, sans hypothèse sur le nombre de tuiles ou le réseau de connexion utilisé. Le principal bénéfice de ce flot dans ce contexte n'est plus uniquement le temps réduit de conception comme pour le cas d'un MPSoC hétérogène simple, mais plus simplement la maîtrise de la complexité de ces plateformes. Nous dégageons les principales difficultés introduites par ces plateformes multi-tuiles dans les points suivants.

2.2.1 Problématique 1 : rendre la mise en œuvre des communications entre tâches transparente pour l'application

Le langage de haut niveau de description de l'application, comme nous l'avons dit, doit en proposer un modèle parallèle. Nous verrons au Chapitre III quelques représentations parallèles pour les applications, mais sommairement, au niveau du code généré, nous aurons à faire à des tâches qui sont gérées par un système d'exploitation. Ces tâches communiquent entre elles pour obtenir la fonctionnalité désirée. Cependant, le programmeur ne connaît pas à l'avance les éléments matériels de la plateforme qui permettent à plusieurs tâches de s'échanger des données. Une des principales difficultés introduites par les systèmes multi-tuiles, par rapport à un MPSoC hétérogène simple, est le nombre de possibilités offertes pour faire communiquer deux unités de calcul, appartenant à la même tuile, à la même puce, ou à des puces différentes. Nous introduisons au niveau du formalisme de l'architecture la notion de chemin matériel, c'est à dire la liste ordonnée des blocs matériels traversés pour réaliser une communication. Ceux-ci seront introduits formellement au

Chapitre IV, la Figure 2.1 donne une illustration de ces chemins matériels pour une tuile, nous pouvons voir que des chemins matériels de communication en écriture (d'une unité de calcul à une mémoire) et en lecture (d'une mémoire à une unité de calcul) illustrés ne sont pas forcément symétriques.

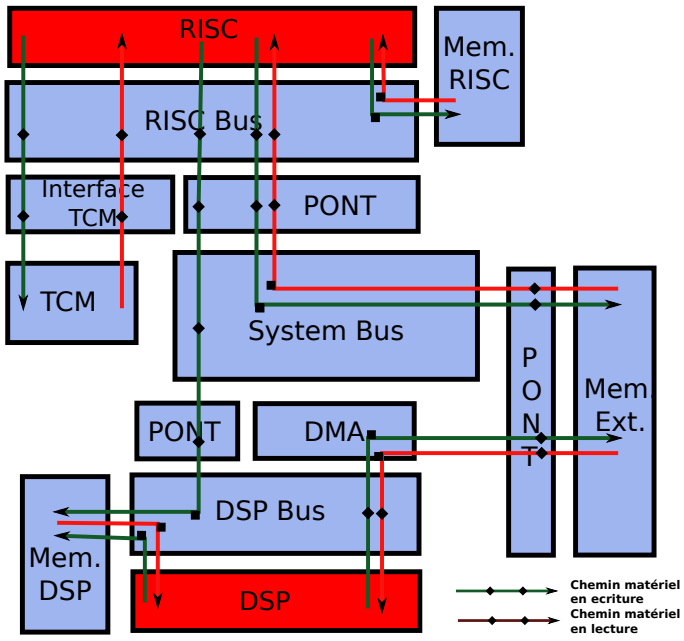

FIGURE 2.1 – Illustration de quelques chemins matériels sur un MPSoC hétérogène

Les chemins matériels peuvent être classifiés en trois catégories, pourvu que nous fassions l'hypothèse d'une interface homogène établissant le lien entre le monde de la tuile et le réseau de communication extra-tuile.

- Les chemins intra-processeur, qui permettent à deux tâches sur le même processeur de communiquer. Les chemins matériels les plus efficaces sont ceux qui ne sortent pas du système local au processeur.
- Les chemins intra-tuile inter-processeurs, qui connectent des processeurs de la même tuile. Les chemins les plus efficaces utilisent une mémoire partagée.
- Les chemins inter-tuiles, qui connectent des processeurs appartenant à des tuiles différentes. Les chemins matériels incluent nécessairement une ou plusieurs interfaces réseaux qui permettent de sortir de la tuile.

Il est important, dans le modèle de l'application de haut niveau, de découpler le réseau de communication logique qui connecte les tâches du modèle de l'application du chemin de communication matériel qui permettra de le réaliser sur la plateforme cible. Ceci nécessite donc une phase de déploiement des éléments applicatifs sur les éléments matériels. Ceci est d'autant plus vrai qu'en raison de l'hétérogénéité de l'accès à la structure de communication, les chemins de communication autorisés ne sont pas les mêmes en fonction de l'unité de calcul exécutant une tâche donnée (par exemple, sur la plateforme multi-tuiles présentée en Chapitre I, le DSP doit sortir de son sous-système exclusivement par un DMA).

Il est donc important de proposer une gestion des communications transparente. Ceci nécessite l'emploi d'une interface de communication qui permet au programmeur de s'affranchir des détails de plus bas niveau pour faire communiquer ses tâches. Ces détails de bas niveau concernent la mise en œuvre du protocole de communication, la programmation des éléments du chemin de communication matériel (par exemple les

périphériques) et des synchronisations inhérentes au schéma de communication par passage de message.

2.2.2 Problématique 2 : pouvoir gérer indifféremment des plateformes avec des processeurs de natures quelconques

Si le programmeur de l'application écrit son modèle de haut niveau et les fonctionnalités associées à ses tâches sans avoir à se soucier des unités de calcul qui exécuteront son application, il n'en va pas de même du logiciel de plus bas niveau qui sera utilisé sur les processeurs.

Deux processeurs différents nécessitent des traitements spécifiques, qui sont normalement du ressort d'un système d'exploitation :

- L'initialisation du système, qui est spécifique à chaque processeur, mais aussi à la plateforme.
- La gestion du contexte d'exécution, qui dépend des registres du processeur (registres généraux, registres de pile, compteur programme).
- La gestion des interruptions. L'activation, le masquage, et le traitement lorsqu'une interruption se produit.

Cependant, d'autres différences interviennent lorsque l'on considère la communication entre deux unités de calculs de natures différentes :

- L'*endianness*, d'abord, qui concerne l'ordre dans lequel les octets sont stockés dans un mot de plus de 1 octet. Si deux processeurs qui essaient de communiquer ont une endianness différente, une conversion du message est requise (du côté de l'émetteur ou du récepteur).
- La taille des mots. Ceci concerne plus particulièrement les DSPs, le mAgicV d'Atmel fournit une mémoire donnée de largeur 40 bits. Echanger des données peut donc supposer une conversion, qui peut même varier en fonction de leur type (entier ou réel).
- Les opérations autorisées par le compilateur, en particulier les types de données supportés et les opérations sur les champs de bits.

Ce dernier point est important à noter et nous y reviendrons au Chapitre III. Si les DSP sont aujourd'hui fournis avec des compilateurs efficaces qui affranchissent une programmation dans leur langage machine, il est difficile à ces compilateurs d'offrir le même niveau de "confort" et de fonctionnalité que ceux qui visent des processeurs de type RISC. Le confort dont nous parlons ici concerne la manipulation de types de données différents et la définition de structures de données complexes. De plus, ils offrent des options de configuration et d'optimisation qui diffèrent des compilateurs plus "classiques", qui sont étroitement liés à l'architecture interne du DSP.

2.2.3 Problématique 3 : pouvoir gérer indifféremment un nombre de processeurs quelconque et les faire communiquer

Le supercalculateur le plus performant au moment de l'écriture de cette thèse, le Cray XT-5 installé au laboratoire national de Oak Ridge aux Etats-Unis, possède 200000 nœuds de calcul, (des Opteron six coeurs à 2,6GHz pour la petite histoire). Si assembler un système à 200000 tuiles n'est pas encore à l'ordre

du jour, nous ne nous fixons pas de limite quant au nombre de tuiles de notre système, il devient difficile dans ce contexte de laisser dans le flot de génération des fichiers à éditer à la main.

Mais l'automatisation du flot n'est pas une tâche aisée, en particulier en raison du processus de compilation croisée, qui introduit beaucoup d'inconnues. Ces inconnues ne peuvent rester à l'état d'inconnues dans l'étape du lien final, et nécessitent donc une formalisation pour permettre au programmeur ou à l'outil de résoudre leur valeur. Nous pouvons citer deux de ces inconnues : d'une part, les adresses physiques des mémoires utilisées pour les communications, ceci concernant aussi bien la zone mémoire dédiée aux échanges de données que les espaces mémoires nécessaires à la communication entre un processeur et un périphérique. D'autre part, la répartition des éléments logiciels (applicatifs ou du système d'exploitation) sur les mémoires de la tuile.

Une autre difficulté dans l'automatisation provient des sources diverses qui permettent de configurer un élément du logiciel. Un bon exemple est la FIFO logicielle en mémoire partagée qui sera utilisée ultérieurement pour faire communiquer deux tâches sur deux processeurs distincts. Si on peut laisser au programmeur le choix de la taille de la FIFO, on ne peut pas lui demander des détails de plus bas niveau comme l'adresse physique de la mémoire utilisée pour la FIFO ou alors le verrou matériel qui permet un accès en mutuelle exclusion aux données.

Une autre difficulté introduite par l'hypothèse d'un nombre quelconque de processeurs, est le maintien d'une cohérence dans la configuration des communications entre tâches s'exécutant sur des processeurs différents. Nous verrons en effet qu'une dépendance est créée au niveau des binaires générés pour des processeurs qui communiquent entre eux. Cette cohérence doit être maintenue et adaptée lorsqu'un changement dans le modèle de haut niveau intervient.

2.3 Contributions

2.3.1 Contribution 1 : un environnement de génération de logiciel qui abstrait la gestion des communications

Nous proposons une approche de programmation qui découple la description parallèle de l'application et la fonctionnalité de ses tâches des détails de plus bas niveau. Pour cela, nous nous appuyons sur une pile logicielle, dont la couche la plus haute représente l'application et son point d'entrée, la deuxième couche représente le système d'exploitation et une bibliothèque de communication que les tâches applicatives doivent utiliser pour communiquer entre elles, et enfin la couche la plus basse qui met en œuvre un ensemble de services offerts par le matériel. Cette couche d'abstraction du matériel, appelée HAL, permet un portage plus aisé de la pile logicielle sur de nouvelles plateformes. Chaque couche repose sur la couche logicielle inférieure par une interface fixée qui sera décrite dans les chapitres suivants.

Le modèle de l'application offert au programmeur se décompose en deux parties : d'une part, la description des tâches et de leurs interconnexions, d'autre part la description du comportement de chaque tâche utilisant exclusivement l'interface de programmation proposée par la couche logicielle de plus bas niveau. La phase de génération de logiciel génère, dans le langage de programmation des tâches, le code nécessaire pour initialiser et créer les tâches elles-mêmes mais aussi leurs canaux de communication.

Nous prenons le parti dans cette thèse d'arrêter à la première couche de la pile logicielle l'étape de génération de logiciel. Une conséquence de ce choix est que le flot doit disposer d'une bibliothèque de composants logiciels. Il y a plusieurs catégories de composants logiciels : le système d'exploitation, le composant qui gère les communications et la couche d'abstraction du matériel. Il ne faut pas confondre à ce stade le composant qui gère les communications, qui permet au programmeur de l'application de s'abstraire des détails d'implémentation des communications, et les composants qui mettent en œuvre une communication du modèle de haut niveau. Ces derniers, comme nous le verrons en détail, sont synthétisés sous forme de pilotes du système d'exploitation choisi. Lorsque ces pilotes implantent une communication inter-tuiles et qu'un périphérique est mis en jeu, le pilote devient très difficile à écrire. Nous proposons alors un modèle pour ces pilotes afin de faciliter l'écriture de nouveaux mécanismes de communication ou l'utilisation d'un périphérique différent.

Le rôle du flot est donc de sélectionner les composants logiciels qui constitueront la pile logicielle finale pour chaque unité de calcul de la plateforme cible : le système d'exploitation, l'interface de communication et le HAL, mais aussi les pilotes nécessaires pour implanter les communications du modèle de l'application et les modules du système d'exploitation. La sélection des pilotes nécessite un déploiement des éléments de communication de l'application sur les chemins matériels offerts par la plateforme afin de choisir le pilote adéquat.

2.3.2 Contribution 2 : des composants logiciels de communication efficaces et une gestion automatique de leur spécialisation

Le flot nécessite des mécanismes de communication efficaces afin de ne pas dégrader les performances globales de l'application. Ceci peut être atteint par un recouvrement du temps de la communication par le temps de calcul (dépendant de chaque application). Il faut donc offrir des mécanismes optimaux pour un chemin matériel donné. Nous proposons quelques protocoles de communication pour les chemins matériels les plus pertinents, chacun avec leurs forces et leurs faiblesses. Les mécanismes inter-tuiles, s'appuyant sur un périphérique de communication, s'avèrent très difficiles à concevoir, nous proposons un modèle de programmation pour faciliter leur écriture. Ces composants de communication ont été intégrés à notre flot de génération, et chaque communication du modèle de l'application déployé sur un chemin matériel donne lieu à la sélection de l'un de ces composants.

La phase de sélection des composants logiciels n'est malheureusement pas suffisante pour obtenir des binaires qui reflètent le comportement de l'application spécifiée à un haut niveau. Il faut passer par une phase de spécialisation des communications. Si nous reprenons l'exemple de la FIFO logicielle, mécanisme utilisé pour faire communiquer deux processeurs appartenant à une même tuile, spécialiser le mécanisme de FIFO consiste à lui assigner une taille, une adresse physique, et un mécanisme de verrou matériel offert normalement par la plateforme.

L'automatisation de cette gestion est rendue difficile par la variété de schémas de communication qu'offre le système multi-tuiles, chacun d'eux nécessitant une configuration différente, en fonction du chemin de communication matériel choisi, du protocole et du périphérique mis en jeu. En plus de cela, comme nous l'avons dit auparavant, les pilotes de communication puisent leur configuration soit dans le langage de haut niveau, soit dans des caractéristiques détaillées de la plateforme (des adresses physiques principalement). Nous avons donc été menés, lors de la mise en œuvre du flot sous forme d'outils, à considérer un

système basé sur des variables classifiées en fonction de leur nature (haut niveau, valeur par défaut ou entrée manuellement, ou alors à chercher dans un gestionnaire de ressources de la plateforme). Chaque mécanisme de communication synthétisé sous la forme d'un pilote, récupère donc chaque valeur de configuration soit par un appel à une fonction fournie avec le modèle, soit par un appel à un gestionnaire matériel qui fournira une adresse physique libre pour l'usage des pilotes.

2.3.3 Contribution 3 : Vers une génération du déploiement des sections logicielles en mémoire

Le déploiement des éléments du logiciel sur une plateforme dont la hiérarchie mémoire n'est pas connue à l'avance est une tâche extrêmement difficile à réaliser, Cette étape est cependant indispensable pour obtenir un binaire qui fonctionne sur une plateforme avec une hiérarchie mémoire complexe (NUMA). La difficulté provient principalement du fait que cette phase de l'édition de liens manipule des concepts de logiciel très bas niveau, du compilateur et de la plateforme cible, et que le présenter de manière intuitive pour le programmeur de l'application est ambitieux. Pourtant, cela est un sujet important à traiter dans la mesure où l'on se dirige vers des systèmes à mémoires explicitement gérées (EMM, voir l'exemple du Cell d'IBM), où la gestion de la mémoire n'est plus assurée par le matériel mais explicitement par le logiciel. Ceci signifie que le programmeur de l'application aura peut-être un jour à gérer l'emplacement mémoire des données et du programme de son application dans une phase d'optimisation.

Notre approche de génération garantissant une résultat fonctionnel mais non forcément optimal au niveau de l'application, nous proposons une génération automatique du script d'édition de liens, reprenant le même principe que la génération de la configuration des pilotes de communication. Nous nous basons donc sur un modèle du script d'édition de lien (pour le RISC), où beaucoup de valeurs sont codées en durs et optimales pour le système d'exploitation et la plateforme considérés. Cependant, proposer une représentation de haut niveau permettant de générer le script d'édition de liens, qui est une tâche source d'erreurs même pour les spécialistes, est l'une des possibles continuations de ce travail.

Chapitre 3

Tour d'horizon

Sommaire

Introduction

Nous avons introduit une approche de programmation alternative aux environnements de programmation embarqués dits "classiques". La caractéristique principale de cette approche est une spécification de l'application à un haut niveau d'abstraction, dans un formalisme où le parallélisme est explicite. Le modèle de l'application doit être accompagné d'une description du déploiement des éléments de l'application (par exemple les connexions entre tâches) sur les éléments de l'architecture (par exemple les chemins de communication matériels). Pour obtenir les binaires à partir de ce modèle de haut niveau, un grand fossé doit être comblé. Nous introduisons une pile logicielle pour y parvenir, chaque couche logicielle ayant besoin d'être composée par le flot de génération de logiciel. Cette composition nécessite une phase de génération de logiciel vers un langage de programmation standard (le C pour fixer les idées), mais aussi une sélection de composants logiciels puisés dans diverses bibliothèques. L'une des bibliothèques importantes lorsque nous voulons cibler des systèmes multi-tuiles hétérogènes, est la bibliothèque qui permet de mettre en œuvre les mécanismes de communication nécessaires pour faire communiquer les tâches applicatives. Les systèmes multi-tuiles offrent en effet une grande quantité de chemins matériels : les chemins intra-processeur, inter-processeurs intra-tuile et les chemins inter-tuiles. Les pilotes de communication utilisés pour les chemins

inter-tuiles, parcequ'ils introduisent des périphériques, sont très complexes à programmer, une génération de ces composants est d'ailleurs un sujet de recherche important [OOJ98]. Une fois les différentes couches logicielles composées pour une unité de calcul donnée, le binaire final est obtenu par une phase de compilation et d'édition de lien, cette dernière étape permettant de passer d'un language standard (C, C++) vers le langage machine et d'attribuer une adresse physique à tous les éléments de la pile logicielle. Dans le meilleur des mondes, toutes ces étapes sont effectuées sans aucune intervention du programmeur de l'application, cependant comme nous nous basons sur un modèle de l'architecture à un haut niveau, beaucoup de données intermédiaires spécifiques à chaque couche logicielle doivent être injectées dans le flot, c'est ce que nous essayons de faire sentir dans ce tour d'horizon.

Nous donnons donc ici un état de l'art sur les modèles et les outils existants basés sur ces modèles. Nous commençons par une revue des modèles de calcul pour l'application et de l'usage qui est fait de ces modèles par quelques outils existants. Nous continuons par une revue des modèles de binaires dans le monde des MPSoCs, en insistant par la même occasion sur la manière dont nous traitons les communications dans ce contexte. Enfin, nous nous intéressons aux outils de compilation qui permettent d'assembler le binaire et de spécifier la répartition des éléments logiciels sur les mémoires de la plateforme.

3.1 Modèles de haut niveau et outils associés

3.1.1 Modèles de Calcul

Un modèle de calcul (MoC) est une vue abstraite et formelle d'un système (mécanique, physique, électronique), qui ignore un certain nombre de détails d'implantation pour en faciliter l'analyse. Par exemple, pour les systèmes électroniques comme les ordinateurs, on abstrait toujours les lois physiques qui régissent la circulation des électrons dans les fils qui interconnectent les éléments de calcul. Il existe de nombreux modèles utilisés pour la description de systèmes de calcul (Réseaux de Petri [HV87], Dataflow Graphs [LM87], Concurrent Sequential Processes [BHR84], Réseaux de Processus [LP95]), Certains modèles sont plus adaptés que d'autres selon le cadre qui est mis en place pour les exploiter.

Nous nous intéressons dans la suite aux Réseaux de Processus (ou Process Networks), qui ont souvent été utilisés pour modéliser des applications ou du matériel mettant en jeu des flots de données.

Réseaux de Processus

Dans les Réseaux de Processus (PN en anglais), des processus communiquent au travers de canaux qui peuvent mettre en tampon les données qui les traversent. Nous décrivons ci-après les Réseaux de Processus de Kahn, qui sont l'un des modèles les plus souvent supportés dans le outils à base de PN existants.

Réseaux de Processus de Kahn

Les réseaux de processus de Kahn (KPNs) sont un modèle de calcul proposé par Gilles Kahn en 1979. C'est un modèle très simple qui a été introduit aussi bien intuitivement que mathématiquement par son

auteur. Si nous gardons l'approche intuitive, il s'agit de processus (qu'il appelle informellement des stations de travail autonomes) et d'un réseau de communication constitué de lignes reliant deux et seulement deux stations. Les lignes de communication sont les seuls moyens pour les stations de communiquer, les lignes transmettent l'information en un temps imprédictible mais fini, et les stations, qui effectuent leurs opérations de manière séquentielle, sont soit en phase de calcul soit en phase d'attente de données sur l'une de leurs lignes à un moment donné.

L'approche mathématique introduit un graphe orienté, où les arcs et les nœuds sont étiquetés. Les arcs portent des données d'un type D ; Kahn introduit D_w comme l'ensemble des séquences dénombrables (finies ou infinies) de D, et associe une telle séquence à un arc du KPN pour représenter l'ensemble des valeurs qu'un observateur placé sur l'arc a pu voir passer à un instant donné de l'exécution du KPN. Il construit ainsi une relation d'ordre partiel dans cet ensemble, et peut définir des fonctions continues (les processus) qui prennent un ensemble de séquences en entrée pour produire un ensemble de séquences en sortie. Cette caractérisation mathématique fait rentrer les KPN dans la catégorie des graphes de flot de données (Dataflow Graphs ou DFG), mais diffèrent d'autres DFG existants à base de jetons qui consistent en une série d'acteurs qui une fois mis à feu par un passage de jeton effectuent une suite d'opérations sans blocage. L'une des caractéristiques importantes des KPN, comme d'autres DFG, est qu'il n'y a pas de perte de données au niveau des nœuds.

Les arcs du KPN sont souvent assimilés à des FIFOs de taille infinie. Ce modèle a beaucoup été utilisé dans les programmes à contrôle statique [Fea95], c'est à dire les programmes ou le flot de données mis en jeu ne conditionne pas le traitement à effectuer sur celui-ci. L'écart entre le modèle et la réalité, en l'occurrence la taille limitée des mémoires des ordinateurs, a été parfois palliée par un mécanimse qui peut détecter et résoudre les situations d'interblocage (deadlocks) [Par95]. En effet, sur FIFO pleine, une opération d'écriture peut devenir bloquante, et mener à une situation ou plus aucun processus ne peut continuer son exécution.

CSP

Nous touchons ici un petit mot sur les CSP (Communicating Sequential Processes) [Hoa78], qui est une algèbre de processus mais aussi un langage de programmation qui modélise les interactions entre divers processus séquentiels concurrents. Un point intéressant à noter ; les communications entre processeurs sont à base de rendez-vous ou de passages de message synchrones. Si deux processus sont amenés à communiquer, et l'un deux atteint en premier le moment où il doit communiquer, alors il se bloque jusqu'à ce que l'autre processus soit prêt à communiquer.

3.1.2 Outils basés sur des PN

De nombreux outils se basent sur des Réseaux de Processus, nous pouvons les classer en quatre catégories :

Outils de co-synthèse logiciel/matériel

Nous commençons par cette catégorie d'outils car c'est celle qui pourrait paraître la plus proche de ce que nous souhaitons réaliser. Le but de ces outils est de concevoir un système embarqué complet en spécifiant à partir d'un modèle d'entrée les blocs qui doivent être réalisés en matériel, ceux qui doivent être réalisés en logiciel et les interfaces de communication entre ces deux mondes. Chinook [COB95], de l'université de Washington, est un environnent de conception complet, qui supporte d'ailleurs plusieurs MoCs. Chinook aborde la notion de synthèse d'interfaces, qui utilise des éléments logiciels et matériels pour réaliser une communication entre deux processeurs ou alors entre un processeur et un périphérique. Les pilotes de périphériques sont synthétisés à partir de diagrammes de timing (timing diagrams), qui représentent la suite de signaux à échanger, les délais à respecter pour les opérations d'E/S, ainsi que le contrat de réponse par le périphérique si ces contraintes sont respectées. Ceci permet à l'outil de faire un choix sur le partitionnement matériel/logiciel de la communication. Du logiciel de bas niveau est généré pour le processeur cible et pour la partie de la communication à réaliser en logiciel.

L'environnement de co-synthèse MoPCoM [KAV$^+$], lui, se base sur un modèle UML pour représenter un système embarqué complet en utilisant le profil UML Marte. Cette méthodologie définit trois niveaux d'abstraction (*abstract*, *execution* et *detailed*), chacun étant composé de descriptions de l'application et de la plateforme. Le niveau le plus détaillé permet une génération de la plateforme sous forme de fichiers VHDL synthétisables [VdLG$^+$09], l'accent dans ces travaux n'est pas mis sur la génération de logiciel mais plutôt sur l'utilisation du profile Marte d'UML pour la modélisation de systèmes matériels.

La principale différence entre co-synthèse et génération de logiciel tient à la plateforme matérielle cible, qui est figée (processeurs, périphériques, mémoires, plages d'adresses) pour cette dernière catégorie. Les outils de co-synthèse nécessitent en général une spécification basée sur un langage de description de matériel (Chinook) ou alors sur un formalisme très pointu (MoPCoM), et nécessitent donc des connaissances poussées de la part de concepteur du système. Nous souhaitons dans notre approche spécifier l'architecture cible à plus haut niveau et affranchir le programmeur d'application de ces problématiques.

Outils d'exploration des solutions de conception (design space exploration)

Les outils d'exploration des solutions de conception ont pour but principal de prendre une description de l'application, et d'en répartir les éléments de manière optimale sur une plateforme, à partir d'un ensemble de caractéristiques de la plateforme cible. Ces outils se basent en général sur un modèle en "Y-chart", composé de trois fichiers : le modèle d'application, le modèle d'architecture et le modèle de déploiement des éléments logiciels sur les éléments matériels de la plateforme. DOL [TBHH07] (Distributed Operation Layer) se base sur ces trois fichiers, décrits dans le format XML. Le modèle de calcul choisi pour l'application est un réseau de processus, avec un ensemble de tâches interconnectées par des canaux points à points de type FIFO de taille finie. Le modèle d'application définit un ensemble de tâches et de canaux de communication logiciels, interconnectés par le biais de ports d'entrée et de sortie. Le modèle d'architecture, lui, définit des blocs matériels pour les processeurs, mémoires et périphériques sous la forme de boîtes noires. Le déploiement des éléments applicatifs sur la plateforme consiste donc principalement à assigner un processeur à chaque tâche. Chaque processus du modèle d'application est associé à un fichier de description de sa fonctionnalité en langage C/C++, chacun définissant une primitive *init* pour initialiser les données internes au processus et *fire* pour décrire le comportement du processus. Ces deux fonctions utilisent une interface de programmation

liée au modèle pour communiquer (*DOL_read*, *DOL_write*) et indiquer que le processus courant peut être tué (*DOL_detach*). Un modèle SystemC est généré à partir de ces fichiers pour valider le comportement de l'application au niveau système et générer automatiquement un déploiement optimal (Voir Annexe1 pour un exemple complet d'un modèle DOL).

De nombreux autres outils d'exploration des solutions d'implémentation existent. Nous pouvons citer Artemis [PEP06], Metropolis [BWH+03] et Ptolemy [BLL+08]. Tous suivent le principe énoncé précédemment, avec quelques différences sur les modèles pour l'application et l'architecture pris en entrée. Metropolis par exemple accepte plusieurs MoCs pour l'application, et adjoint au modèle d'architecture un fichier de description pour chacun de ses éléments, incluant par exemple des informations de délais pour les opérations d'écriture/lecture en mémoire ou pour l'exécution d'un cycle du processeur. Ptolemy, lui, diffère un peu des autres approches d'exploration en s'intéressant plus à la composition de MoCs hétérogènes pour caractériser un système complet quelconque. Ptolemy se présente sous la forme d'un ensemble de classes Java pour modéliser un large spectre de systèmes, donc il reste à un niveau d'abstraction assez élevé.

Nous désirons baser notre flot de génération sur un modèle de l'architecture et de l'application à un haut niveau d'abstraction, mais souhaitons utiliser ces modèles pour générer des binaires destinés à une plateforme matérielle réelle. Il est sûr que pour atteindre ce but, nous avons besoin de plus d'informations pour la génération que pour les outils d'exploration des solutions de conception, cependant les travaux basés sur un modèle en Y-chart, comme DOL, constituent une bonne base pour nos travaux.

Vérification formelle

Un domaine de recherche actuel consiste à transformer un modèle en Y-Chart tel que décrit dans la partie précédente, dans un langage intermédiaire qui permet d'en exploiter les propriétés à des fins d'optimisation et de vérification. Passer par un tel langage intermédiaire permet donc d'effectuer des vérifications formelles avant une phase de génération de logiciel. Par exemple, BiP [BSS09] est un environnement dont le principe est d'assembler incrémentalement des composants hétérogènes, tout en vérifiant des propriétés essentielles au système comme l'exclusion mutuelle ou l'absence d'interblocages.

La génération de logiciel à partir d'un tel modèle intermédiaire a l'avantage d'assurer certaines propriétés qui n'auront pas à être validées par exécution, et rejoint par certains aspects des problématiques soulevées dans cette thèse. Mais notons que nos travaux ne s'intéressent pas aujourd'hui aux aspects temps réels, contrairement au BiP qui a explicitement vocation à cibler les systèmes temps réel.

Génération de logiciel

De nombreux outils de génération de logiciel pour des MPSoCs hétérogènes ont été proposés. Nous pouvons citer Compaan/Laura [SZT+04] qui génère du logiciel pour une plateforme fixée basée sur un microcontrôleur et un FPGA. Il se base sur une description des tâches applicatives écrite sous forme d'un fichier Matlab. Un outil intermédiaire transforme ce modèle en code avec une syntaxe particulière (YAPI), et un déploiement est automatiquement généré pour déterminer quelle partie de l'application doit être portée sur le FPGA. ESPAM [NSD08] est également un outil qui génère du logiciel qui cible des plateformes P2P, c'est à dire des plateformes qui ont le même nombre de processeurs que de processus et autant de FIFOs matérielles que de canaux de communications. Ils embarquent donc un SE mono tâche, et synthétisent leur

plateforme sous forme de VHDL. Il est intéressant de noter que dans cette approche les adresses matérielles des FIFOs sont imposées par la plateforme, le générateur logiciel tient donc compte de ces adresses pour générer le pilote de chaque canal de communication sans intervention extérieure.

Notre approche est très différente des approches existantes pour plusieurs raisons :
- Nous ne cherchons pas à générer du matériel, car nous avons une vraie plateforme matérielle
- Nous ciblons des plateformes avec un nombre de tuiles, et donc de processeur quelconques, chaque processeur pouvant gérer plusieurs tâches
- Nous générons du code également pour les DSPs de notre plateforme.
- Nous ne pouvons nous contenter de la FIFO comme modèle pour les canaux de communication, en particulier lorsque nous souhaitons sortir de la tuile pour aller sur un réseau commuté par paquets.

3.1.3 Modèles pour l'architecture

Après cette revue de MoCs et des outils existants associés, nous avons montré que l'une des différences profondes des approches existantes avec la nôtre, est que l'architecture doit être spécifiée à un haut niveau afin de faciliter la tâche du concepteur de l'application, contrairement à d'autres approches où l'architecture est fixée à l'avance ou spécifiée dans un langage de description de matériel.

Dans ce contexte, nous nous intéressons donc au niveau de détail que nous pouvons attendre du langage de haut niveau. Notre flot ne doit pas être dépendant d'un et d'un seul langage de haut niveau en entrée, il est donc important de garder à l'esprit que nous n'aurons pas les mêmes informations en entrée en fonction de celui que l'on choisit.

Nous avons vu l'approche DOL pour le fichier d'architecture qui est une approche par interconnexion de boîtes noires avec un minimum de détails, qui a le mérite d'être simple à mettre en place. Cependant, certains détails importants manquent, en particulier les plages d'adresses physiques et des caractéristiques de plus bas niveau. IP-XACT est un standard du consortium SPIRIT [Spi], utilisé pour la description d'IPs (Intellectual Properties). Ce modèle, basé sur un ensemble de fichiers XML, fournit un grand niveau de détails pour chaque composant de la plateforme. Nous présentons dans le listing ci-dessous un extrait de fichier IP-XACT pour la spécification d'un DMA, qui décrit la plage d'adresse du DMA, avec une description détaillée de chacun de ses registres. Cette instanciation est utilisée pour une plateforme complète (utilisant un processeur Leon) décrite en IP-XACT.

Le fait qu'IP-XACT soit conçu pour décrire de manière complète un bloc matériel avec une sémantique bien définie le rend peut-être un peu lourd pour une génération de logiciel, cependant c'est le seul langage de description d'architecture à haut niveau à notre connaissance qui comporte autant de détails utiles pour notre propos.

```
<spirit:memoryMaps>
   <spirit:memoryMap>
     <spirit:name>ambaAPB</spirit:name>
     <spirit:addressBlock>
       <spirit:register>
         <spirit:name>sourceAddress</spirit:name>
         <spirit:addressOffset>0x0</spirit:addressOffset>
         <spirit:size>32</spirit:size>
         <spirit:access>read-write</spirit:access>
       </spirit:register>
```

```
        <spirit:register>
          <spirit:name>destinationAddress</spirit:name>
          <spirit:addressOffset>0x4</spirit:addressOffset>
          <spirit:size>32</spirit:size>
          <spirit:access>read-write</spirit:access>
        </spirit:register>
          <spirit:field>
            <spirit:name>length</spirit:name>
            <spirit:description>Size of data transfer</spirit:description>
            <spirit:bitOffset>0</spirit:bitOffset>
            <spirit:bitWidth>8</spirit:bitWidth>
            <spirit:access>read-write</spirit:access>
          </spirit:field>
          <spirit:field>
            <spirit:name>sourceIncrement</spirit:name>
            <spirit:description>00=no incrment, 01=Byte, 01=half word, 11=Word incrementing of address</
                spirit:description>
            <spirit:bitOffset>8</spirit:bitOffset>
            <spirit:bitWidth>2</spirit:bitWidth>
            <spirit:access>read-write</spirit:access>
          </spirit:field>
...
```

3.2 Structure du binaire

3.2.1 Systèmes d'exploitation "monolithiques" dans les MPSoCs

Après ce bref aperçu des MoCs et des modèles d'architecture qui constituent l'entrée de notre flot, nous nous intéressons à la structure du logiciel qui doit être généré. La structure typique des binaires dans le monde des MPSoCs est une pile logicielle, composée d'un système d'exploitation et d'une couche d'abstraction du matériel (HAL). L'application repose sur le système d'exploitation pour des opérations telles que la création de tâches ou les synchronisations. Le SE, lui, repose exclusivement sur le HAL pour accéder aux ressources matérielles offertes par la plateforme cible.

Linux ne possède pas à proprement parler de HAL, et c'est l'une des raisons pour laquelle cibler une nouvelle plateforme avec ce système d'exploitation reste une tâche ardue. Il possède d'ailleurs plusieurs versions de son noyau et autres modules, une pour chaque plateforme et processeurs associés. Au contraire, eCos [Mas02], qui n'est pas un système d'exploitation généraliste mais un SE ciblant exclusivement des plateformes à base de MPSoCs, définit un HAL clair qui exporte un ensemble de fonctions qui restent les mêmes pour les différents processeurs (ARM9, MIPS32, PowerPC, x86, etc.) et plateformes. Le HAL dans eCos est minimal et comprend des opération vitales du processeur (gestion de contexte, interruptions, opérations d'E/S basiques) et un nombre réduit d'opérations liées à la plateforme (conversion d'endianness, verrous matériels).

Cette structure autorise un portage facile du système d'exploitation sur la nouvelle plateforme, le travail de portage consistant théoriquement uniquement à implanter les primitives du HAL. Cette vue idéale des choses n'est malheureusement pas totalement exacte, car lors d'un changement de plateforme, d'autres éléments que le processeur changent, comme les périphériques (timers, ports séries, etc.). Ceux-ci nécessitent des pilotes qui ne sont pas, dans l'approche d'eCos, inclus dans le HAL, ce qui est somme toute censé car lorsque une application est portée sur une plateforme différente, il n'est pas dit qu'elle nécessite tous les périphériques de la plateforme. L'écriture des pilotes de la plateforme fait donc partie du travail de portage.

3.2.2 Systèmes d'exploitation par composants

En raison des contraintes en termes de mémoire et de temps d'exécution dans le monde des MPSoCs, de nombreux environnements de construction du système d'exploitation par composants ont vu le jour. Le principe de ces environnements est de définir les composants qui constituent les couches de bas niveau du système, incluant le SE et le HAL, mais aussi les éléments du SE (gestion de MMU, systèmes de fichiers, pilotes, ordonnanceurs) et les composants de bas niveau pour gérer les périphériques. Cet aspect de sélection des composants du système d'exploitation est particulièrement important dans notre contexte, car il ne faut pas oublier que notre flot génère une pile logicielle également pour des unités de calcul spécialisées comme le DSP. Il doit donc être aisé de composer un système d'exploitation léger pour ces unités de calcul.

APES (APlication Elements for Embedded Systems) [GP09] est un tel environnement. Il est basé sur le système d'exploitation DNA-OS et des HAL supportant une large gamme de processeurs, y compris des DSPs. Les éléments du système d'exploitation sont composés statiquement avant la phase de compilation et de lien, ce qui garantit une empreinte mémoire minimale. La sélection des composants logiciels se fait à l'aide d'un fichier de description des composants requis par l'application, qui est invoqué lors de la compilation des éléments constitutifs du système complet.

Il existe bien d'autres environnements à base de composants logiciels, comme la bibliothèque de composants KORTEX et son système d'exploitation THINK [FSLM02]. Celui-ci offre également la possibilité de charger en cours d'exécution de nouveaux composants, fonctionnalité intéressante mais superflue dans notre cas car notre flot de génération est statique et concerne des applications dont le contenu n'est pas amené à changer en cours du cycle de vie du système embarqué. Cet environnement ne supporte pour l'instant qu'un seul processeur, le PowerPC. D'autres systèmes d'exploitation avec une approche par composants peuvent être trouvés dans la littérature, comme OSKit [FBB+97] ou TinyOS [LMG+04] qui est plutôt conçu pour des systèmes de réseaux de capteurs. Ceux-ci ne proposent pas de HAL, ce qui ne sied pas à notre but de composition de pile logicielle. Nous pouvons ouvrir une parenthèse ici sur les réseaux de capteurs, dont les travaux de recherche ont des problématiques communes à notre propos (besoin de pile logicielles légères, importance des communications, facilité de programmation). Quelques travaux dans ce domaine se sont d'ailleurs également rapprochés du monde du calcul de haute performance pour simplifier la tâche des programmeurs en proposant un portage de l'interface de MPI dans leur pile logicielle [MT08]. Cependant, nous n'avons pas trouvé dans la littérature de travaux s'intéressant à la génération de logiciel pour ces systèmes.

3.3 Outils de compilation et gestion de la mémoire

Qui dit production de binaires sous-entend obligatoirement des outils de compilations qui permettent de passer d'un langage standard (comme le C) au langage machine adéquat pour le processeur cible. L'hétérogénéité des plateformes visées introduit alors une difficulté supplémentaire, dans la mesure où les compilateurs pour les RISC et les processeurs VLIWs ne se présentent pas de la même manière.

3.3.1 Compilateurs classiques

Les RISCs, comme l'ARM ou le MIPS disposent de nombreux compilateurs, industriels (Realview), ou libres (GCC, LLVM). Comme les RISCs sont utilisés dans une grande variété de plateformes, les compilateurs offrent la possibilité d'affecter une adresse mémoire à chaque élément du logiciel. La chaîne de compilation GNU (GCC), ld, accepte un ensemble de directives sous forme d'un script d'édition de liens. Une illustration de ce script est donnée ci-dessous, nous pouvons voir que ce script nécessite plusieurs informations liées à la plateforme matérielle : la liste des mémoires et leur plage d'adresses, la liste des sections (ou en-têtes) physiques, et enfin la partie la plus importante, la répartition des sections trouvées dans les objets compilés sur les mémoires déclarées au début du fichier.

```
MEMORY /*Definition des memoires du systeme*/
  {
    ram : ORIGIN = 0, LENGTH = 256K
    rom : ORIGIN = 0x30000000, LENGTH = 4M
  }
PHDRS   /* En-tetes physiques */
{
     text PT_LOAD;
     data PT_LOAD;
     dynamic PT_DYNAMIC;
}
SECTIONS /*Déploiement des sections logicielles dans les memoires a des adresses donnees*/
{
  . = 0x000000;
  .text : { *(.text) }: text > ram
  . = 0x3000000;
  .data : { *(.data) }: data > rom
  .os_heap .+ 0x20000: {}: dynamic > rom

  .=0x30040000   /*Declaration de deux  variables globales initialisees  à partir de cette adresse*/
   GLOBAL1 = .;LONG(0xDEADBEEF)
   GLOBAL2 = .;LONG(0xBAADFACE)
}
```

Ecrire un script d'édition de lien est une tâche ardue et source d'erreurs, et est une étape peu ou pas connue de beaucoup de programmeurs de logiciel qui écrivent des programmes destinés à s'exécuter sur la machine hôte. Cependant, notre flot de génération fait intervenir un processus de compilation croisée. Celui-ci implique une définition explicite des caractéristiques du système cible et en particulier de ses mémoires. Il faut bien entendu noter que les compilateurs et éditeurs de lien offrent également des options qui concernent des niveaux d'optimisation, des options qui renseignent l'environnement du processeur (unité matérielle pour les flottants...) et des options sur les formats à générer.

3.3.2 Compilateurs pour les DSPs

Les VLIWs, de par leur architecture spécifique, ont des compilateurs qui offrent moins de souplesse que les RISCs. Ceux-ci sont en général fixés pour une architecture spécifique, et les configurations qu'ils offrent concernent plus des problèmes d'optimisation (prédiction, déroulage de boucle) qui sont très importantes à régler. Certains VLIWs disposent d'outils de haut niveau pour générer leur compilateur, comme la suite d'outils de compilation Chess [Tar] ou Lisatek [WHLM03], ceux-ci sont basés sur des langages de description de processeurs qui décrivent l'architecture interne du DSP et les instructions (sous forme d'une grammaire en général) disponibles pour ce DSP. Le compilateur connait donc à l'avance les mémoires disponibles et y range les éléments logiciels comme il l'entend.

Chapitre 4

Méthodologie de synthèse des communications

Sommaire

Introduction

Un certain nombre de concepts manipulés par notre flot de génération de logiciel ont été explicités au Chapitre précédent : le modèle d'application en réseau de processus, la structure en pile du logiciel, l'approche par composants du système d'exploitation. Le contenu du fichier d'architecture est en revanche resté vague, nous allons tenter de construire un formalisme pour ce modèle tout au long du manuscrit. Dans un premier temps, nous nous intéressons aux paramètres nécessaires dans la description de l'architecture

pour un déploiement des éléments du modèle de l'application sur les éléments de celui-ci. Nous introduisons pour ce faire la notion de chemins matériels de communication de manière formelle.

Le flot de génération prend ces modèles pour produire un binaire par processeur de la plateforme cible, qui se présente sous la forme d'une pile logicielle. Comme nous l'avons vu, nous prenons le parti d'arrêter la phase de génération à la couche haute de la pile logicielle (la couche applicative). Nous nous basons ainsi sur des bibliothèques de composants pour en former les couches plus basses. L'aspect composant dans un système d'exploitation a déjà été largement traité dans la littérature, nous ne nous attarderons donc pas dans ce chapitre à expliciter le cadre de sélection des composants du système d'exploitation en fonction du type d'unité de calcul visé. Nous rappelons cependant que la pile logicielle doit être légère pour les DSPs, car ces unités de calcul sont conçues pour exécuter des tâches de calcul répétitives, et non du code de contrôle qui diminue les performances de manière dramatique. Les composants de communication logiciels seront en revanche étudiés en détails.

Nous présentons donc ici notre modèle de flot de génération, et insistons sur la manière dont les communications sont *synthétisées* depuis le déploiement d'un canal de communication sur un chemin matériel jusqu'aux binaires finaux contenant les tâches aux deux extrémités du canal. Le terme de synthèse est bien adapté à ce que nous essayons de faire ici : dans le domaine de la conception de circuits numériques, la phase de synthèse consiste à assembler un ensemble de blocs matériels d'une ou plusieurs bibliothèques données et à les interconnecter par des fils pour refléter le comportement décrit par le programmeur dans le langage de description de matériel. Dans notre cas, nous ne disposons pas de bibliothèques matérielles, mais de bibliothèques logicielles que nous ne pouvons bien sûr connecter par des fils. Les composants logiciels contiennent des interfaces (avec le SE et le matériel) définies, mais nécessitent une spécialisation pour refléter le contexte d'exécution par rapport au modèle de haut niveau.

L'étude de cette phase de spécialisation pour quelques mécanismes de communication nous permettra de montrer qu'une vue formelle des fichiers d'architecture n'est pas suffisante pour arriver jusqu'aux binaires destinés à s'exécuter sur la plateforme matérielle.

4.1 Formalisme adopté pour les trois composantes du modèle de haut niveau

Nous formalisons dans cette partie tous les éléments nécessaires à la composition des piles logicielles pour un système multi-tuiles. Nous introduisons en particulier les chemins de communication intra-tuile et inter-tuiles, avec en point de mire la sélection des composants de communication qui permettront aux tâches déployées sur les différents processeurs de la plateforme de s'échanger des données. Nous définissons deux types de chemins matériels de communication qui n'utilisent pas le même medium pour s'échanger des données : les chemins de communication intra-tuile d'une part qui reposent sur une mémoire partagée, les chemins de communication inter-tuiles d'autre part qui reposent sur un réseau inter-tuiles et ses périphériques attachés.

4.1.1 Formalisme pour l'architecture

Tuile de base

Nous introduisons l'ensemble des tuiles $\mathcal{T}$, et définissons une tuile T de cet ensemble comme suit :

$$T = \{\mathcal{B}_T, \mathcal{I}_T, \mathcal{L}_T\}$$

$$\text{oú } \mathcal{B}_T = \{\mathcal{U}_T, \mathcal{M}_T\}$$

$$\text{et } \mathcal{I}_T = \{\mathcal{R}_T, \mathcal{P}_T, \mathcal{L}_I\}$$

$\mathcal{B}_T$ est l'ensemble des unités de calcul ($\mathcal{U}_T$) et des mémoires ($\mathcal{M}_T$) de la tuile T, $\mathcal{I}_T$ est l'infrastructure de communication de la tuile composée de réseaux de communication (ensemble $\mathcal{R}_T$) et de différentes interfaces réseaux que nous appellerons abusivement les périphériques de la tuile (ensemble $\mathcal{P}_T$).

$\mathcal{L}_I$ représente l'ensemble des liaisons dans l'infrastructure de communication, sous forme de paires orientées $(Peripherique, Reseau)$ ou $(Reseau, Peripherique)$. De même, $\mathcal{L}_T$ représente l'ensemble des liaisons entre un élément de $\mathcal{B}_T$ et un réseau de communication de $\mathcal{R}_T$. sous forme de paires orientées $(Bloc, Reseau)$ ou $(Reseau, Bloc)$.

Nous définissons à présent deux ensembles pour les chemins matériels de communication intra-tuile entre une unité de calcul UC et une mémoire M : l'ensemble des chemins en écriture et l'ensemble des chemins en lecture. Nous notons "$\cdot$" l'opérateur de concaténation mathématique [Qui46] entre deux éléments de l'architecture, et $(E)*$ la concaténation mathématique de 0 ou n éléments E.

Pour l'ensemble des chemins de communication matériel en écriture intra-tuile, nous avons :

$$WritePaths_T(UC, Mem) := \{Mem_0 \cdot Res_0 \cdot (Periph_i \cdot Res_{i+1})^* \cdot Mem\}$$

$$\text{où } \begin{cases} (Mem_0, Res_0), (Res_{last}, Mem) \in \mathcal{L}_T \\ (Res_i, Periph_i), (Periph_i, Res_{i+1}) \in \mathcal{L}_I \end{cases}$$

Pour le chemin de communication matériel en lecture, nous avons :

$$ReadPaths_T(UC, Mem) := \{Mem_0 \cdot Res_0 \cdot (Periph_i \cdot Res_{i+1})^* \cdot Mem\}$$

$$\text{où } \begin{cases} (Res_0, Mem_0), (Mem, Res_{last}) \in \mathcal{L}_T \\ (Res_i, Periph_i), (Res_{i+1}, Periph_i) \in \mathcal{L}_I \end{cases}$$

Autrement dit, un chemin de communication en écriture (respectivement en lecture) intra-tuile entre une unité de calcul et une mémoire connecte cette unité à une mémoire Mem_0 sur le même réseau. Celle-ci représente l'emplacement mémoire des données à émettre (respectivement à recevoir) par le passage de messages. Le chemin matériel utilise l'infrastructure de communication de la tuile, constituée de périphériques

et de réseaux, pour connecter cette mémoire à la mémoire Mem pour un accès en écriture (respectivement en lecture). Nous pouvons alors introduire l'ensemble des chemins matériels de communication internes à une tuile T, $\mathcal{I}ntraPaths_T$:

$$\mathcal{I}ntrapaths_T = \{\mathcal{W}riteIntraPaths_T, \mathcal{R}eadIntraPaths_T\}$$

$$\text{où } \begin{cases} \mathcal{W}riteIntraPaths_T = \{WritePaths_T(UC, Mem)/(UC, Mem) \in \mathcal{U}_T \times \mathcal{M}_T\} \\ \mathcal{R}eadIntraPaths_T = \{ReadPaths_T(UC, Mem)/(UC, Mem) \in \mathcal{U}_T \times \mathcal{M}_T\} \end{cases}$$

Système multi-tuiles

Nous introduisons l'ensemble des systèmes multi-tuiles $\mathcal{MT}$, et définissons un système à n tuiles de la manière suivante :

$$MT(n) = \{\mathcal{T}^n, \mathcal{I}_{MT(n)}\}$$

$$\text{où } \mathcal{I}_{MT(n)} = \{\mathcal{P}_{MT(n)}, \mathcal{R}_{MT(n)}, \mathcal{L}_{MT(n)}\}$$

Un système multi-tuiles est donc constitué de plusieurs tuiles $\{T_1, T_2, \cdots T_n\}$, et donc d'un ensemble d'unités de calcul et de mémoires que nous notons $\mathcal{U}_{MT}$ et $\mathcal{M}_{MT}$.

$$\mathcal{U}_{\mathcal{MT}} = \bigcup_{i=1}^{n}(\mathcal{U}_{T_i}) \text{ et } \mathcal{M}_{MT} = \bigcup_{i=1}^{n}(\mathcal{M}_{T_i})$$

connectées par une infrastructure de communication inter-tuiles $\mathcal{I}_{MT(n)}$. Cet ensemble est lui-même constitué d'un ensemble de réseaux ($\mathcal{R}_{MT(n)}$), de périphériques ($\mathcal{P}_{MT(n)}$) et d'un ensemble $\mathcal{L}_{MT(n)}$ de paires orientées $(Peripherique, Reseau)$, $(Reseau, Peripherique)$ représentant les liaison entre ceux-ci. Les périphériques du système multi-tuiles sont inclus dans l'ensemble des interfaces réseau de toutes les tuiles :

$$\mathcal{P}_{MT(n)} \subset \bigcup_1^n \mathcal{P}_{T_n}$$

De la même manière que pour la tuile, nous définissons des chemins matériels de communication inter-tuiles, mais cette fois-ci entre une unité de calcul $UC_{T_i} \in T_i$ et un réseau inter-tuile $Res_{MT} \in R_{MT}$.

L'ensemble des chemins de communication en écriture inter-tuiles est défini de la manière suivante :

$$WritePaths_{MT}(UC_i, Res_{MT}) := \{Mem_0 \cdot Res_0 \cdot (Periph_i \cdot Res_{i+1})^* \cdot Periph_{MT} \cdot Res_{MT}\}$$

$$\text{où } \begin{cases} (Mem_O, Res_0) \in \mathcal{L}_{T_i} \\ (Res_i, Periph_i), (Periph_i, Res_{i+1}) \in \mathcal{L}_{I_{Ti}} \\ \text{et } (Periph_{MT}, Res_{MT}) \in \mathcal{L}_{I_{MTn}} \end{cases}$$

et l'ensemble des chemins de communication de lecture inter-tuiles sont définis par

$$ReadPaths_{MT}(UC_i, Res_{MT}) := \{Mem_0 \cdot Res_0 \cdot (Periph_i \cdot Res_{i+1})^* \cdot Periph_{MT} \cdot Res_{MT}\}$$

$$\text{où } \begin{cases} ((Mem_0, Res_0) \in \mathcal{L}_{T_i} \\ (Periph_i, Res_i), (Res_{i+1}, Periph_i) \in \mathcal{L}_{I_{Ti}} \\ \text{et } (Periph_{MT}, Res_{MT}) \in \mathcal{L}_{I_{MT}} \end{cases}$$

Autrement dit, un chemin de communication en écriture (respectivement en lecture) inter-tuiles représente l'infrastructure de communication de la tuile pour connecter, par le biais d'une mémoire locale, l'unité de calcul à un périphérique faisant le lien avec un réseau inter-tuiles Res_{MT} pour une opération d'écriture (respectivement de lecture).

Un système multi-tuiles dispose ainsi d'un ensemble de chemins matériels de communication inter-tuiles que nous notons $\mathcal{I}nterPaths$, composé d'un ensemble de chemins matériels intra- et inter-tuiles en écriture et en lecture.

$$\mathcal{I}nterPaths_{MT} = \{\mathcal{W}riteInterPaths_{MT}, \mathcal{R}eadInterPathsMT\}$$

$$\text{où } \begin{cases} \mathcal{W}riteInterPaths_{MT} = \{WritePaths_T(UC, Res)/(UC, Res) \in \mathcal{U}_{MT} \times \mathcal{R}_{MT}\} \\ \mathcal{R}eadInterPaths_{MT} = \{ReadPaths_T(UC, Res)/(UC, Res) \in \mathcal{U}_{MT} \times \mathcal{R}_{MT}\} \end{cases}$$

Il nous reste tout simplement à introduire l'ensemble des chemins matériels du système multi-tuiles complet, composé des chemins matériels de communication intra-tuile et inter-tuiles.

$$\mathcal{H}wComPaths_{MT} = \mathcal{H}wComWritePaths_{MT}, \mathcal{H}wComReadPathsMT$$

$$\text{où } \begin{cases} \mathcal{H}wComWritePaths_{MT} = \bigcup_{i=1}^{n} \mathcal{W}riteIntraPaths_{T_i} \cup \mathcal{W}riteInterPaths_{MT} \\ \mathcal{H}wComReadPaths_{MT} = \bigcup_{i=1}^{n} \mathcal{R}eadIntraPaths_{T_i} \cup \mathcal{R}eadInterPaths_{MT} \end{cases}$$

4.1.2 Formalisme pour l'application

Le modèle pour une application A est un réseau de Processus que nous représentons ainsi :

$$A = \{Processus_A, Canaux_A, \mathcal{L}_A\}$$

Les connexions se présentent sous forme de paires orientées, la relation suivante est vérifiée dans l'ensemble $\mathcal{L}_A$:

$$\forall C \in Canaux_A, \exists!(P0, P1) \in Processus_A^2/((T0, C) \in \mathcal{L}_A) \wedge ((C, T1) \in \mathcal{L}_\mathcal{A})$$

Autrement dit, les canaux du modèle de haut niveau sont unidirectionnels point à point. Nous notons $Canaux_A(P_i, Pj)$ le sous-ensemble de $Canaux_A$ qui représente l'ensemble de canaux de communication qui relient le processus P_i au processus P_j.

$$\forall (P_i, P_j) \in Processus_A^2, Canaux_A(P_i, P_j) = \{C \in Canaux_A / (P_i, C), (C, P_j) \in \mathcal{L}_A)\}$$

4.1.3 Déploiement des éléments applicatifs

Enfin, le déploiement des éléments applicatifs sur l'architecture multi-tuiles se présente sous la forme de deux fonctions :

$mapping\_tache : Processus_A \longmapsto \mathcal{U}_{\mathcal{MT}}$

qui à chaque processus du modèle d'application fait correspondre une unité de calcul du système multi-tuiles, et

$mapping\_canaux : Canaux_A \longmapsto (\mathcal{H}wComReadPaths \times \mathcal{H}wComWritePaths)$

qui à chaque canal de l'application associe deux chemins matériels de communication, un en lecture, un en écriture. Un canal C donné connecte deux processus que nous notons (P_i, P_j). Nous notons $UC_i = mapping\_tache(P_i)$ et $UC_j = mapping\_tache(P_j)$, et notons T_i, T_j les tuiles auxquelles ces unités de calcul appartiennent. La fonction $mapping\_canaux$ appliquée à ce canal doit vérifier la relation suivante :

$mapping\_canaux(C) = (WP, RP)$

$$\text{où} \begin{cases} (WP, RP) \in (WritePaths_{T_i}(UC_i, Mem) \times ReadPaths_{T_i}(UC_j, Mem)) & si \quad T_i = T_j \\ (WP, RP) \in (WritePaths_{MT}(UC_i, Res) \times ReadPaths_{MT}(UC_j, Res)) & si \quad T_i \neq T_j \end{cases}$$

$$(Mem \in \mathcal{M}_{T_i}, Res \in \mathcal{R}_{MT(n)})$$

Le déploiement des canaux de communication de l'application utilise donc des chemins matériels intra-tuile pour un canal connectant deux processus mis sur la même tuile (en spécifiant une mémoire commune aux deux chemins), soit des chemins matériels inter-tuiles pour un canal connectant deux processus situés sur deux tuiles differentes (en spécifiant un réseau extra-tuile commune aux deux chemins).

4.2 Mise en œuvre d'un canal de communication dans la pile logicielle

4.2.1 Structure de pile adoptée

Nous avons introduit au Chapitre III le concept de pile logicielle largement utilisé dans le monde des MPSoCs. La Figure 4.1 décrit la structure de pile proposée. Nous retrouvons bien les tâches applicatives et leur module d'initialisation dans la couche la plus haute, le système d'exploitation, qui possède une approche par composants et qui repose exclusivement sur le HAL (Hardware Abstraction Layer) pour accéder aux services offerts par la plateforme matérielle. Au niveau de la deuxième couche, conjointement avec le SE, nous trouvons un module dont nous n'avions pas parlé dans le Chapitre III, qui est la couche de COM. Elle propose aux tâches applicatives (écrites par le programmeur de l'application) une interface unique pour mettre en œuvre les opérations d'initialisation, d'écriture, de lecture et de contrôle des canaux de communication du réseau de processus. Elle implémente une coquille qui permet de faire le lien entre le monde des réseaux de processus et le monde du SE en ce qui concerne les communications du modèle.

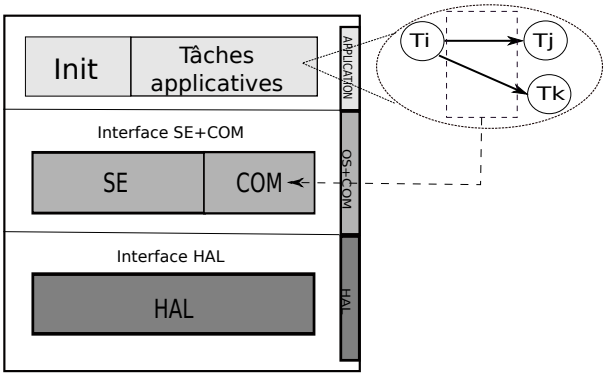

FIGURE 4.1 – Structure de pile logicielle

Nous pouvons noter un point intéressant : nous disposons d'un modèle en réseau de processus, ou le terme processus est pris au sens mathématique du terme (une fonction continue, comme dans le formalisme de Kahn). Dans un système d'exploitation, le processus est un concept bien défini qui diffère du sens mathématique du terme. Il s'agit d'un ensemble d'instructions, auquel est alloué un espace mémoire pour la pile, pour les descripteurs de fichiers utilisés et autres. Dans les systèmes d'exploitation "classiques" pour ordinateurs de bureau (Linux, Windows), le processus est l'unité permettant l'exécution des programmes utilisateurs. Cependant, le processus informatique possède un contexte lourd à manipuler (en particulier durant la phase de changement de contexte), c'est pour cette raison que le concept de fil d'exécution (thread en anglais) a été introduit. Un fil d'exécution est lancé par un processus, et partage l'état du processus père avec les autres fils lancés par ce processus. Ceci implique un partage de certaines ressources (comme les fichiers ou les ports réseaux ouverts) par des sémaphores au niveau de l'implémentation du fil et rend son écriture plus délicate pour les personnes non rompues à l'art de la programmation système. Nous continuerons dans la suite de ce manuscrit de parler de tâche, ce mot indiquant la mise en œuvre dans le SE d'un processus (mathématique) du modèle de haut niveau.

En ce qui concerne le HAL, nous choisissons une approche où le HAL ne renferme qu'un minimum de code assembleur bas niveau concernant le processeur cible (interruptions, changement de contexte, E/S...)

et son proche environnement (endianness, verrous matériel...), comme dans eCos en fait. Les fonctions d'E/S (*HAL_READ*, *HAL_WRITE*) permettent d'accéder à l'infrastructure de communication de la tuile pour atteindre un autre élément, elle permet donc, au niveau d'une unité de calcul donnée, d'homogénéiser les opérations à effectuer pour atteindre le proche environnement de celle-ci. Ceci peut impliquer la programmation de blocs matériels (que nous avons appelés peut-être un peu abusivement périphériques dans la partie de formalisation) qui font le lien entre deux réseaux de la tuile comme le DMA, ou alors peuvent être effectuées de manière transparente pour le programmeur grâce à un pont matériel.

Cela dit, le HAL ne renferme pas de fonction pour programmer les pilotes de périphériques pour les chemins inter-tuiles, qui possèdent une structure bien plus complexe : les pilotes et leur partie dépendante de la plateforme (de leur périphérique en fait) ne sont pas dans le HAL, mais reposent sur le HAL pour accéder au matériel. Le support par le flot d'une plateforme ne tient donc pas seulement à la présence des HAL pour les processeurs de le plateforme, mais aussi sur la présence de pilotes pour ses périphériques. Ceci est vrai également pour les pilotes autres que les pilotes réseaux, comme les pilotes pour matériel de stockage.

4.2.2 Système de fichiers virtuels et pilotes de communication

Un système de fichier virtuel (VFS) est un accès uniforme fourni par un système d'exploitation pour accéder à un système de fichiers mais aussi à des périphériques ou à un réseau. Pour cela, un ensemble de systèmes de gestion de fichiers (SGF) sont montés à l'initialisation du SE, ceux-ci pouvant être de type ext2 ou FAT32 par exemple. Un fichier est donc stocké dans le format imposé par son SGF en mémoire (dans le disque dur pour les ordinateurs de bureau), et une requête au VFS (écriture, lecture, contrôle) sur ce fichier va donner lieu à des opérations sur ses descripteurs. En revanche, un système de gestion de périphériques, également attaché au VFS du système d'exploitation à l'initialisation, ne fournit pas une description de ses fichiers sous forme de descripteurs stockés en mémoire, mais associe à chacun de ses fichiers un pilote qui permettra d'effectuer les opérations requises à plus haut niveau. Par exemple, si le système de gestion de périphériques est monté à l'initialisation dans le répertoire "/devices", le fichier "/devices/swfifo.2" représentera un fichier de ce système, dont le nom permettra d'associer le pilote de communication de type FIFO logicielle qui sera étudié ci-après.

4.2.3 Illustration d'un écriture sur un canal de communication

Nous illustrons les intéractions entre composants de la pile logicielle, par l'étude d'une écriture sur un canal de communication. La Figure 4.2 montre la succession d'appels de fonctions effectuée lors d'une opération d'écriture sur un canal de communication dans le code d'implantation d'une tâche fournie avec le modèle de haut niveau. Nous pouvons voir que chaque pilote de communication exporte une série de fonctions comme indiqué dans la Figure 4.2), cette interface étant dépendante du SE utilisé bien entendu. En l'occurrence, la primitive *write* est utilisée et donne lieu, à un moment de son exécution, à un appel à *HAL_WRITE*, qui permet d'effectuer une écriture sur le réseau d'interconnexion de la tuile. Comme indiqué précédemment, la primitive *HAL_WRITE* peut être plus ou moins complexe selon l'environnement du processeur visé, il pourra s'agir d'une simple écriture à l'adresse matérielle donnée, ou au contraire d'une programmation de bloc matériel.

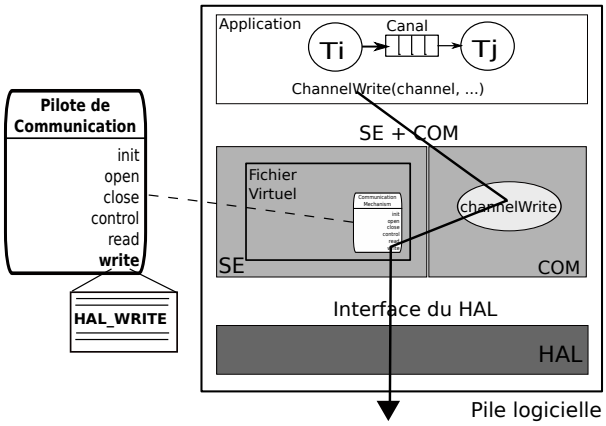

FIGURE 4.2 – Suite d'appels de fonction mis en œuvre pour une écriture sur un canal de communication

4.3 Flot de génération

4.3.1 Vue générale

Notre flot de génération de logiciel [GPY+07] (voir Figure 4.3) est composé d'une partie *front-end* qui parcourt le modèle de l'application, de l'architecture et du déploiement pour composer les différentes couches logiciels, et d'une partie *back-end* qui compile les différents éléments logiciels (cette étape n'étant pas nécessaire pour les éléments fournis sous forme de bibliothèques précompilées) et procède à une édition de liens pour produire un binaire exécutable pour chaque processeur de la plateforme.

La partie *front-end* du flot de génération est composée de deux phases principales. Tout d'abord, une phase de génération génère le point d'entrée de l'application. Celui-ci utilise l'interface de programmation du SE pour créer et lancer les tâches applicatives (fournies avec le modèle sous forme de code spécifiant leur comportement) déployées sur le processeur considéré, et l'interface de programmation de la COM pour configurer les canaux de communication utilisés par ces tâches. Le code ci-dessous donne un exemple de fichier généré, en utilisant une interface *pthread* pour la gestion des fils d'exécution et une interface de communication simple pour la gestion des canaux. Ce fichier généré lance deux fils d'exécution fournis par ailleurs (fil1 et fil2) connectés par deux canaux de communication liés aux fichiers virtuels *"/devices/d940_swfifo.0"* et *"/devices/d940_rendez-vous.0"* par la primitive *channelInit*. Le nom des deux fichiers permet de reconnaître le pilote attaché à chacun des deux canaux : une FIFO logicielle pour le premier, un Rendez-vous pour le second.

```
void main()
{
    // Variables locales
    Channel canaux[2];
    pthread_t fil1, fil2;


    // Initialisation des canaux de communication
    canaux[0] = channelInit("/dev/d940_swfifo.0");
    canaux[1] = channelInit("/dev/d940_rendez-vous.0");
```

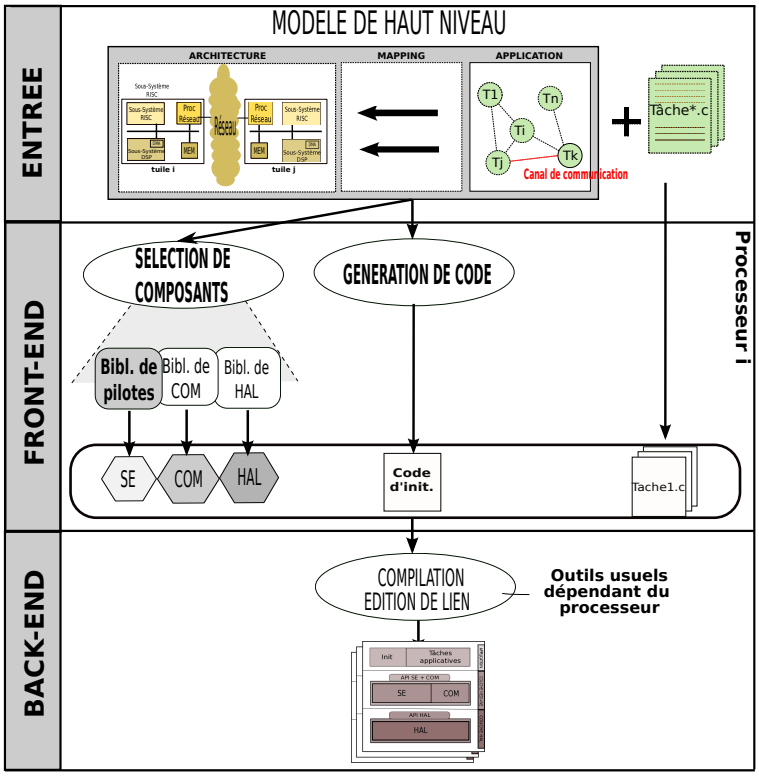

FIGURE 4.3 – Description du flot de génération de logiciel

```
    // Initialisation des fils, le point d'entrée est une fonction behaviour
    // fournie par ailleurs
    pthread_create(&fil1, NULL, fil1_behaviour, canaux);
    pthread_create(&fil2, NULL, fil2_behaviour , canaux);

    // Suspension du fil courant jusqu a la fin de l execution
    pthread_join(fil2, NULL);
}
```

Nous pouvons voir dans cet exemple de code généré quelques contraintes au niveau de la description des fonctionnalité des tâches. Outre la sémantique de communication imposée, celle-ci doit fournir une fonction *behaviour* avec un nom imposé par le nom du processus du modèle. Ces règles peuvent varier en fonction du modèle de haut niveau qui peut imposer d'autres contraintes. (Voir Annexe 1 pour un exemple de génération de code avec le formalisme de DOL).

La partie *front_end* du flot de génération est également composée d'une phase de sélection de composants qui permet au flot de puiser des éléments dans des bibliothèques logicielles existantes pour sélectionner le SE, la COM et le HAL souhaités. Le travail de sélection consiste également à sélectionner les pilotes nécessaires pour mettre en œuvre les canaux de communication du modèle de haut niveau. La sélection des pilotes de communication, contrairement au choix des autres modules du SE, dépend du déploiement des éléments applicatifs sur l'architecture.

La partie *back-end* du flot de génération, quant à elle, se base sur des outils de compilation et d'édition de liens existants pour compiler les différents éléments sélectionnés et produire une binaire exécutable par le processeur cible. Nous verrons tout ce que cela implique plus en détail dans le Chapitre V.

4.4 Composants de communication

4.4.1 Structure interne d'un pilote de communication

Les composants de communication se présentent sour la forme de pilotes associés à un système d'exploitation. Nous en présentons une vue schématique en Figure 4.4, celle-ci devant être appliquée principalement aux fonctions de lecture et d'écriture du pilote (les primitives d'initialisation et de fermeture rentrent également dans ce schéma).

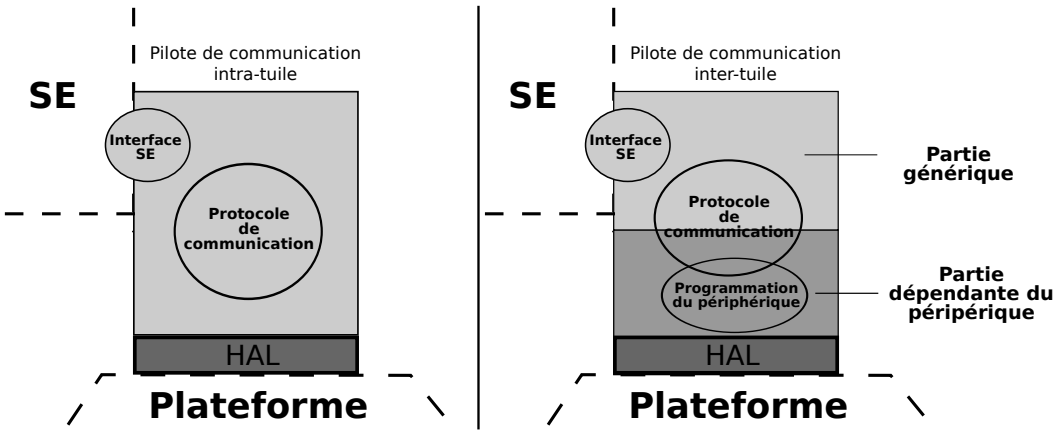

FIGURE 4.4 – Structure interne d'un pilote de communication

Nous voyons apparaître une différence entre les pilotes de communication mettant en œuvre un chemin de communication intra-tuile d'une part et un chemin de communication inter-tuiles d'autre part. Une partie commune à ces deux types de pilotes est l'interface avec le système d'exploitation (les primitives exportées par le pilote pour un SE donné, mais aussi les fonctions de synchronisation, de traitement des interruptions et de gestion du fil d'exécution offertes par le SE), et le HAL qui permet d'accéder au proche environnement du processeur. Outre ces deux interfaces, ils offrent tous deux un protocole de communication qui définit la manière dont les deux extrémités du canal s'échangent les données. Ces protocoles ne sont pas les mêmes suivant que la communication est basée sur une mémoire partagée (chemin intra-tuile) ou sur un réseau de communication (chemin inter-tuiles) comme nous le verrons dans cette partie. Dans le premier cas, le périphérique peut atteindre la mémoire de communication par le biais du HAL, et ne nécessite aucun autre support matériel pour mettre en œuvre son protocole. En revanche, atteindre un périphérique de communication inter-tuiles ne suffit pas à atteindre la tuile cible par le biais du réseau de communication, il faut programmer le périphérique pour atteindre la tuile cible, d'où la présence pour les pilotes de communication inter-tuiles d'une composante dépendante du périphérique utilisé.

Plusieurs remarques intéressantes peuvent être faites à ce stade ; le HAL peut contenir une part de programmation d'interfaces réseau internes (comme un DMA) mais cela est transparent pour les pilotes de communication grâce au choix fait pour la pile logicielle et le HAL. Dans tous les cas, une fois le HAL fixé pour un binaire, le chemin pour aller d'une UC à une mémoire ou un périphérique est fixé. Nous rappelons la définition d'un chemin matériel inter-tuiles en écriture par exemple :

$$WritePaths_{MT}(UC_i, Res_{MT}) := \{Mem_0 \cdot Res_0 \cdot (Periph_i \cdot Res_{i+1})^* \cdot Periph_{MT} \cdot Res_{MT}\}$$

Nous notons ici un premier écart important entre le modèle et le binaire : le chemin situé entre la mémoire locale au processeur (Mem_0) et le medium de communication ($Periph_{MT}$ pour le chemin inter-tuiles ci-dessus) est imposé par le HAL. Ce choix permet de simplifier le processus de sélection des pilotes, car le critère de sélection principal n'est finalement plus le moyen pour atteindre le medium de communication à l'intérieur de la tuile, mais les mémoires et les périphériques mis en jeu, le HAL assurant un chemin matériel optimal. Un moyen de coller plus au modèle sera décrit dans les travaux futurs, et consiste à générer une partie du pilote de communication dans le flot.

Nous étudions dans la suite quelques mécanismes de communication intra-tuile et inter-tuiles et présentons la manière dont le flot va choisir le protocole parmi la liste des protocoles disponibles pour un chemin matériel donné. Sommairement, nous pouvons dire que nous avons eu à choisir entre une approche à la MPI où l'utilisateur choisit lui-même la nature de ses communications (bufferisées, synchrones, ready et bloquantes ou non bloquantes), et l'approche "boîte noire" ou le programmeur de l'application ignore tout du protocole utilisé pour mettre en œuvre les communications de son modèle. Comme nous le verrons en pratique, ce choix a souvent été contraint pour le DSP en raison des ressources limitées dont il dispose.

4.4.2 Protocoles de communication intra-tuile

Nous nous intéressons ici aux canaux de communications de modèle d'application déployés sur un chemin matériel intra-tuile pour faire communiquer deux tâches situées sur deux unités de calcul $UC1$ et $UC2$. Comme nous l'avons vu dans la première partie de ce chapitre, un chemin matériel en écriture WP appartenant à l'ensemble $WritePaths(UC1, Mem)$ et un chemin matériel en lecture RP appartenant à l'ensemble $ReadPaths(UC2, Mem)$ sont associés au canal, une mémoire partagée Mem est donc utilisée comme tampon intermédiaire entre les tampons d'écriture et de lecture.

Nous avons illustré en Chapitre I l'implantation d'un échange de messages entre un DSP et un ARM par un protocole simple (pour ne pas dire simpliste) basé sur un bit de synchronisation entre les deux unités de calcul. Nous introduisons ci-après deux protocoles de communication plus évolués.

FIFO logicielle

La FIFO logicielle (voir Figure 4.5) permet un stockage des données dans la mémoire partagée. Un verrou matériel est utilisé pour protéger l'accès en lecture et en écriture de l'en-tête et de la FIFO. Nous pouvons noter qu'une opération devient bloquante lorsque la FIFO est pleine (pour une opération d'écriture) ou vide (pour une opération de lecture). Les données étant stockées dans un tampon intermédiaire, plusieurs copies intermédiaires sont nécessaires pour effectuer la transaction complète : une copie du tampon d'émission au tampon intermédiaire du côté de l'écrivain, une copie du tampon intermédiaire au tampon de réception du côté du lecteur. Ces copies intermédiaires pénalisent bien entendu les performances, d'autant plus que la quantité de données à transférer est importante.

La Figure 4.6 montre la programmation des primitives read/write du pilote mettant en œuvre une FIFO logicielle.

Nous pouvons constater que plusieurs primitives du HAL sont utilisées : *HAL_READ* et *HAL_WRITE* pour lire/écrire une adresse donnée de la plateforme, *HAL_PLATFORM_LOCK* et

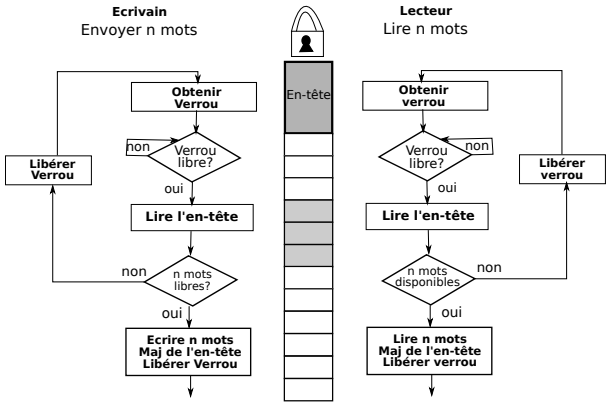

FIGURE 4.5 – Principe de la FIFO logicielle

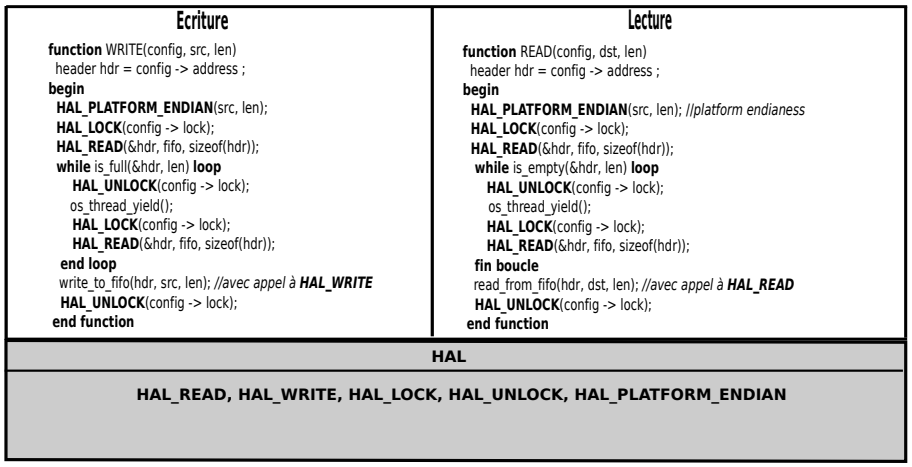

FIGURE 4.6 – Détails de programmation des primitives read/write d'un pilote implantant une FIFO logicielle

HAL_PLATFORM_UNLOCK pour prendre/libérer un verrou matériel et *HAL_PLATFORM _ENDIAN* pour convertir les données reçues et envoyées vers une endianness fixée pour la plateforme. L'interface du pilote avec le système d'exploitation, outre les primitives exportées par le pilote, consiste uniquement en une fonction *os_thread_yield* qui permet de céder la main aux autres fils d'exécution du processeur. Nous pouvons observer cependant que lorsque $UC1 = UC2$, nous pouvons aussi utiliser des primitives de gestion des sémaphores en lieu et place des fonction de prise et libération de verrou du HAL. En d'autres termes, lorsque le chemin de communication permet de faire communiquer deux fils situés sur le même processeur, un verrou logiciel peut être utilisé à la place d'un verrou matériel. Dans l'implantation présentée en Figure 4.6, nous pouvons noter une structure *config* et la présence de deux champs : *lock* (pour identifier le verrou matériel) et *address* pour disposer de l'adresse de début de la structure partagée pour le canal. La structure n'a pas explicitement un champ *taille*, qui permet de connaître la taille de la FIFO, mais nous supposons que cette information, ainsi que les pointeurs de lecture et d'écriture courants, sont contenus dans l'en-tête qui a été initialisé par ailleurs et qu'ils sont utilisés dans les fonctions *is_empty* et *is_full* qui permettent de déterminer si l'opération de lecture/écriture en FIFO peut s'effectuer.

Si nous résumons, trois attributs doivent être associés à un canal de type FIFO logicielle inter-

processeurs (l'identificateur de verrou est superflu pour les canaux intra-processeur) :

$$Attributs(FIFO) := \text{Adresse FIFO}, \text{Taille FIFO}, \text{Identificateur Verrou}$$

Rendez-vous

Le Rendez-vous quant à lui est directement inspiré de l'implantation des communications utilisée dans [Hoa78], il introduit une synchronisation entre processus lecteur et écrivain. Le premier processus qui cherche à communiquer sur son canal signale sa présence (en indiquant également l'adresse de son tampon) au processus pair avant de se mettre en attente. Celui-ci doit alors effectuer directement le transfert de données, et indiquer la complétion de l'opération. Nous pouvons noter que la mémoire partagée dans ce protocole ne stocke pas les données mais uniquement des informations de contrôle sur l'état de la transaction. Le processus chargé d'effectuer le transfert, l'écrivain par exemple, doit utiliser un autre chemin matériel non spécifié pour la transaction, mais ceci ne concerne pas le mécanisme de sélection, mais la manière dont la primitive du HAL (*HAL_READ*/*HAL_WRITE*) met en œuvre l'accès à une certaine adresse de la plateforme.

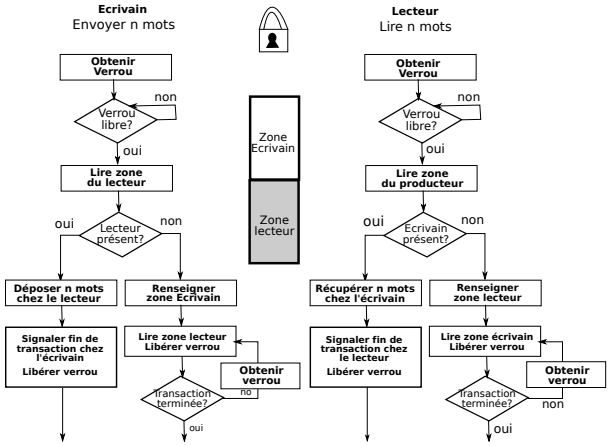

FIGURE 4.7 – Principe du Rendez-vous

La Figure 4.8 donne l'implantation des primitives de lecture et d'écriture pour un pilote mettant en œuvre un protocole Rendez-vous.

Nous pouvons constater l'emploi de cinq primitives du HAL comme dans le cas de la FIFO logicielle : *HAL_READ*, *HAL_WRITE*, *PLATFORM_LOCK*, *PLATFORM_UNLOCK*, *PLATFORM_ENDIAN*. Chaque canal dispose également d'attributs qui doivent être configurés pour chaque canal :

$$Attributs(RENDEZ-VOUS) := \text{Adresse structure controle}, \text{Verrou Logiciel}$$

Le Rendez-vous ne nécessite pas la spécification de la taille d'un tampon intermédiaire, puisqu'il n'en dispose pas. Ceci est d'ailleurs un grand atout du protocole Rendez-vous, car celui-ci ne nécessite pas de copies intermédiaires pour effectuer un transfert.

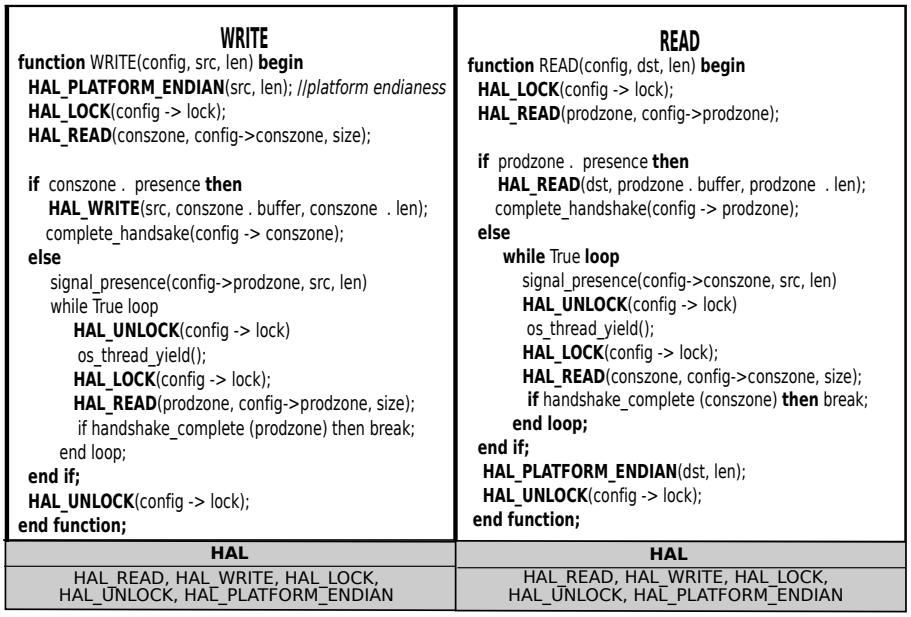

FIGURE 4.8 – Détails de programmation des primitives read/write d'un pilote implantant un Rendez-vous

Rendez-vous en mode ready

Si nous dressons une comparaison avec les primitives de communication disponibles dans MPI, nous disposons avec les mécanismes de FIFO logicielle et de Rendez-vous des modes *Bufferisé* et *Synchrone* pour un canal de communication déployé sur un chemin matériel de communication intra-tuile. MPI propose également un mode *Ready*, ou l'écrivain doit toujours arriver avant le lecteur. Ce mode est utile lorsque l'ordonnancement dans un processeur est de type statique, et où les appels aux primitives de lecture et d'écriture du pilote doivent assurer une complétion du mécanisme de communication dans tous les cas. C'est au programmeur d'application dans ce cas de faire attention à l'ordre d'exécution de ses tâches dans l'ordonnancement statique. Nous proposons dans la Figure 4.9 un protocole inspiré du Rendez-vous décrit précédemment, mais simplifié pour assurer une complétion du mécanisme quoi qu'il arrive. Les tests de présence du processus pair ont donc disparu dans cet algorithme, qui s'en trouve grandement simplifié.

Le mode ready possède les mêmes attributs que le mode Rendez-vous cité précédemment :

$$Attributs(\text{RENDEZ} - \text{VOUS} - \text{READY}) := \text{Adresse structure controle}, \text{Verrou Logiciel}$$

Il est important de distinguer ici les notions de communication non bloquante d'une part et en mode Ready d'autre part. Tous les algorithmes que nous avons proposés jusqu'ici sont tous bloquants dans la sémantique de MPI, car la primitive de communication appelée ne redonne la main au fil d'exécution que lorsque l'opération de lecture/écriture est achevée (le fil pouvant céder la main à l'ordonnanceur du SE). Nous ne proposons pas de variante non bloquante des trois mécanismes présentés ici, car le programmeur d'application n'est censé choisir ses primitives de communication que lors de la phase de déploiement, c'est-à-dire après avoir écrit la fonctionnalité de ses tâches applicatives. L'utilisation d'un mode non bloquant pour un canal de communication imposerait une modification de celles-ci, afin d'inclure un test de complétion de la communication.

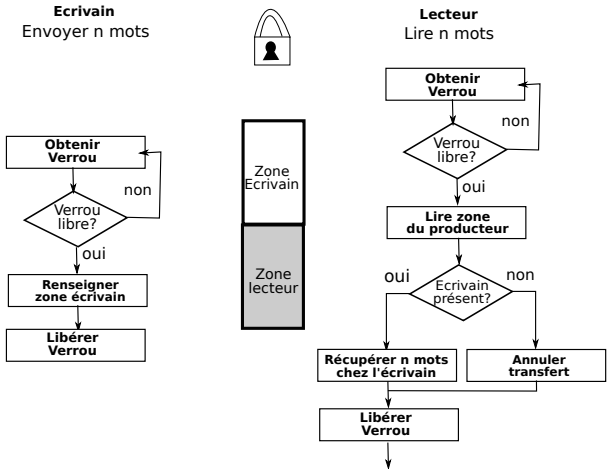

FIGURE 4.9 – Mécanisme de communication en mode Ready

4.4.3 Protocoles de communication inter-tuiles

Nous nous intéressons ici aux canaux de communication déployés sur un chemin matériel inter-tuiles pour faire communiquer deux tâches déployées sur deux unités de calcul $UC1$ et $UC2$ (appartenant à deux tuiles différentes). Un chemin matériel en écriture WP appartenant à l'ensemble $WritePaths(UC1, Res)$ et un chemin matériel en lecture RP appartenant à l'ensemble $ReadPaths(UC2, Res)$ sont associés au canal, un réseau Res connectant différentes tuiles par le biais de périphériques de communication est donc utilisé, et contrairement au cas intra-tuile, aucun tampon intermédiaire n'est explicitement utilisé.

Les réseaux inter-tuiles utilisés pour l'infrastructure de communication sont des réseaux extensifs commutés par paquet. Le principe d'un envoi de message par le biais d'un tel réseau est de le fragmenter en blocs élémentaires qui voyagent dans le réseau indépendamment les uns des autres. Le réseau est formé d'un ensemble de blocs matériels qui permettent le routage de ces paquets vers la destination finale. Nous faisons abstraction de l'infrastructure matérielle du réseau dans le formalisme adopté, nous supposons que les opérations effectuées dans le pilote sont suffisantes pour mener à bien le transfert. Nous étudions deux types de réseaux, Ethernet et RDMA, et quelques protocoles associés que nous avons intégrés sous forme de composants de communication logiciels.

Ethernet

L'Ethernet représente les deux couches basses (physique et liaison) de la suite de protocoles IP [RFC]. Celle-ci est principalement utilisée pour connecter des ordinateurs par un réseau devenu quasiment universel (Internet bien sûr). Ethernet n'en reste pas moins utilisé dans certains superordinateurs (le CrayXT5 et le QPace, premiers aux classements du top500 et du green500), avec un support matériel et des connexions spécifiques, comme le Gigabit Ethernet [GEA] qui permet d'envoyer des trames à un Gigabit par seconde sur tout type d'infrastructure (fils de cuivre ou fibre optique). Il faut tout de même souligner que ce réseau tend à être utilisé plutôt pour accéder un serveur à des fins de stockage de grandes quantités de données (dans un système de fichiers par exemple), et pas forcément pour faire communiquer les nœuds de calcul.

La pile IP se décompose en différentes couches : la couche physique (chargée de la conversion entre bits et signaux électriques ou optiques), la couche liaison de données (avec un système d'adressage matériel MAC), la couche réseau (IP, ARP), la couche transport (TCP, UDP, L2TP) et la couche applicative (SMTP, LDAP). Les interfaces réseau disponibles implantent la couche physique et une bonne partie de la couche liaison de données en règle générale, certains travaux de recherche ont cependant tenté d'intégrer les protocoles IP et TCP en matériel [OK06].

Nous présentons en Figure 4.10 deux protocoles de la couche transport, TCP et UDP. Le protocole TCP (Transmission Control Protocol) débute par une phase de connexion entre un client et un serveur que nous assimilons pour notre propos à un écrivain et à un lecteur. Ceux-ci s'échangent des données par séquences, chacune de ces séquences devant être validée par le récepteur qui peut alors ordonner les séquences reçues par ordre croissant si besoin. TCP assure donc l'intégrité des données de l'écrivain au lecteur. Au contraire, UDP (User Datagram Protocol) propose un mode non connecté qui n'offre aucune assurance sur l'intégrité des données du côté du lecteur. UDP peut détecter que le message est corrompu grâce à une vérification de somme de contrôle, mais ne dispose d'aucun mécanisme pour détecter les messages perdus ou faire une requête de réémission.

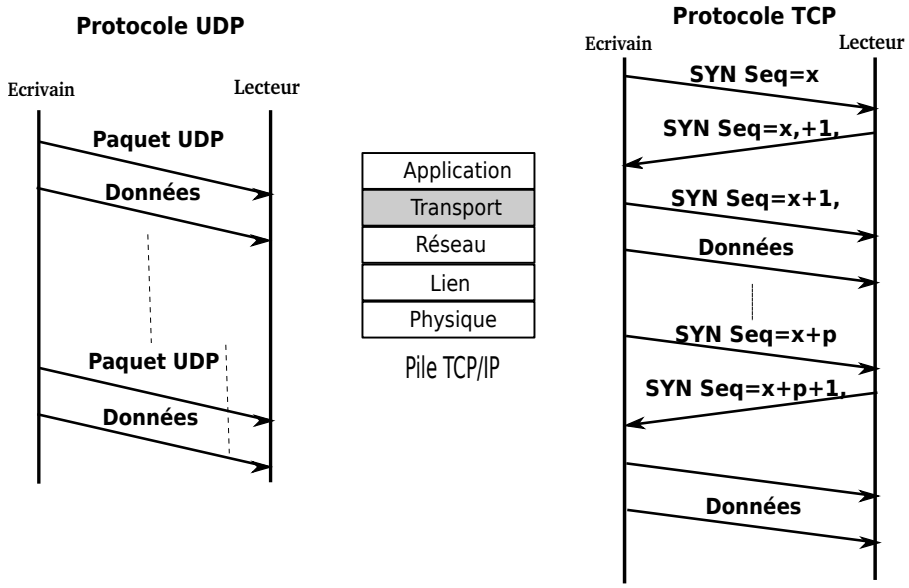

FIGURE 4.10 – Deux protocoles de la suite de protocoles IP : TCP et UDP

Une interface de programmation, les sockets, permet à la couche applicative d'émettre ou recevoir des données par le biais du protocole TCP (socket client et socket serveur) ou UDP (socket datagramme). Nous pouvons également parler de la socket *Raw*, qui occulte la couche transport pour transférer les données. Notons que l'utilisation des protocoles de la couche IP est très lourde à manipuler, car elle implique de nombreuses copies de données intermédiaires et des calculs de somme de contrôle qui prennent une quantité non négligeable de ressources de calcul [FHN+03]. Son utilisation dans le domaine du calcul de haute performance est d'ailleurs mise en doute dans [PBP04], qui préconise justement l'utilisation des protocoles RDMA plutôt qu'une utilisation de sockets par une comparaison du traffic mémoire engendré par les différents protocoles.

Nous proposons trois pilotes pour les trois types de sockets à utiliser pour les réseaux Ethernet : connecté, non connecté, raw. Chacun de ceux-ci met donc en œuvre son protocole de la couche transport, et

les protocoles de plus bas niveau de la pile non pris en charge par l'interface réseau ciblée. Concrètement, le pilote doit former les en-têtes de la suite de protocoles (Ethernet, IP, TCP/UDP) avec récupération de l'adresse MAC du distant par une requête ARP (Address Resolution Protocol), et transférer les données suivant les spécifications de son protocole de transport. Nous ne présentons pas ici l'interface avec la partie dépendante du périphérique, cependant la structure générale indiquée en Figure 4.14 est respectée.

Le partie protocole de communication des trois pilotes nécessitent plusieurs types d'informations annexes. Tout d'abord, une configuration concernant l'adressage de la tuile et du processeur cible pour chaque canal :

$$Attributs(\text{canal socket}) := \\ \text{AdresseIP destination}, \text{Port destination}, \text{PorT source}, \text{Processeur destination}$$

Elle nécessite également des informations sur l'hôte, partie commune à tous les canaux utilisant le réseau Ethernet.

$$Attributs(\text{hote socket}) := \text{Adresse MAC}, \text{AdresseIP source}, \text{Masque reseau}, \text{Passerelle}$$

RDMA

Le RDMA (Remote Direct Memory Access) n'est pas à proprement parler un réseau. Il s'agit plutôt, pour dresser une comparaison avec le réseau Ethernet, d'une série de protocoles au niveau Transport et Réseau qui nécessite le support du périphérique pour effectuer certaines opérations. Nous proposons en Figure 4.11 deux protocoles : le Eager et le Rendez-vous (initié par le lecteur). Nous pouvons voir que le protocole Rendez-vous fait appel à une primitive GET, qui indique à l'interface réseau d'effectuer une opération de récupération de données chez le distant, sans faire intervenir son système d'exploitation. Le protocole Eager, comme son nom l'indique, permet à l'écrivain d'envoyer immédiatement ses données. Comme dans le cas de la FIFO logicielle en mémoire partagée, il fait intervenir plusieurs copies de données intermédiaires, cependant il est en général plus avantageux pour les échanges d'une faible quantité de données que le protocole Rendez-vous en raison de la pénalité induite par les message d'initialisation et de fin de ce protocole. En ce qui concerne la couche physique du réseau RDMA, elle est non standard et dépendante de la manière dont le périphérique concerné forme ses en-têtes. Il existe plusieurs vendeurs sur le marché qui proposent un service RDMA, comme le bus haute performance Infiniband [Inf].

Nous proposons deux composants de communication supportant une infrastructure RDMA, un pour le protocole Eager, l'autre pour le protocole Rendez-vous initié par le lecteur. La Figure 4.12 montre l'implantation du protocole Eager sur une interface dépendante du périphérique sous-jacent. Dans cette implantation, l'écrivain doit vérifier que la taille des paquets qu'il forme et qu'il envoie par la fonction *RDMA_SEND_PKT* ne dépasse pas la taille maximale autorisée pour le paquet, une fragmentation des données à émettre peut donc être nécessaire. Les fonctions de réception du paquet offertes par la partie dépendante du périphérique sont non bloquantes, ce qui permet au fil d'exécution concerné de céder la main lorsque la requête ne peut être satisfaite.

La Figure 4.13 montre l'implantation du protocole Rendez-vous initié par le lecteur. Ce protocole repose sur les mêmes fonctions que le protocole Eager pour émettre et recevoir les paquets d'initialisation et de

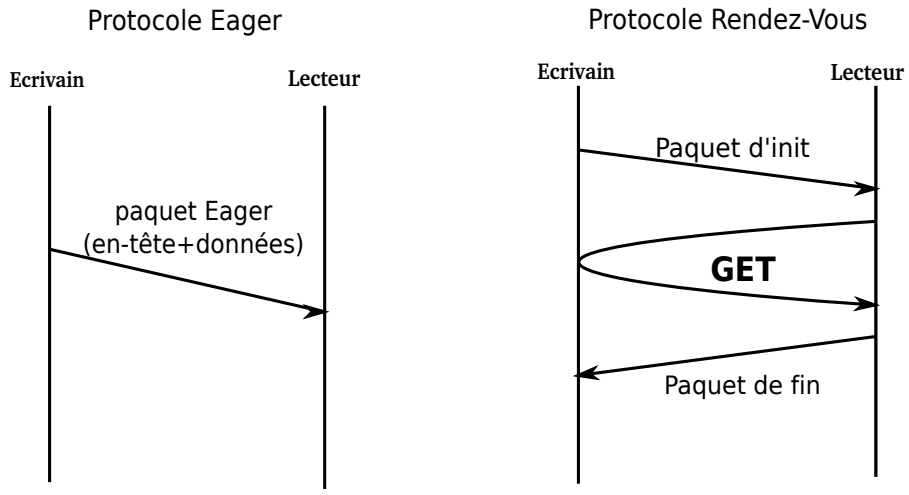

FIGURE 4.11 – Deux protocoles RDMA : Eager et Rendez-vous

fin, nous pouvons voir qu'il fait cependant appel à une fonction *RDMA_GET* qui utilise les capacités du périphérique pour récupérer directement les données en mémoire distante. La fonction *RDMA_TEST* permet de tester l'état de la transaction *GET* et rend la main au fil pour qu'il cède éventuellement la main.

Dans les deux protocoles, nous avons masqué la structure des en-têtes, ce choix spécifique au pilote (car non interprété par le matériel) est présenté en Annexe II.

Les attributs suivants sont associés aux protocoles Eager et Rendez-vous, nous pouvons voir que nous nous inspirons du système d'adressage de MPI en associant à chaque tuile un rang qui correspond à son identification dans le réseau considéré.

$$Attributs(\text{canal rdma}) := \text{ID du canal}, \text{Rang de la tuile cible}, \text{Processeur cible}$$

Comme pour le cas de l'Ethernet, les pilotes RDMA ont besoin d'une partie de configuration commune pour l'hôte afin de mener à bien leur protocole.

$$Attributs(\text{hote rdma}) := \text{Rang de la tuile hote}, \text{Processeur hote}$$

4.4.4 Programmation d'un périphérique de communication

Après avoir passé en revue quelques protocoles de communication basés sur une mémoire partagée ou sur un réseau, nous nous intéressons à la partie de programmation des périphériques de communication pour les communications inter-tuiles (voir Figure 4.4). La Figure 4.14 donne un aperçu de cette partie du pilote, dont le but est de mettre en place un protocole avec le périphérique pour réaliser un transfert de données (émission/réception) sur le réseau de communication vers la tuile cible. Cette partie du pilote est au service du protocole de communication.

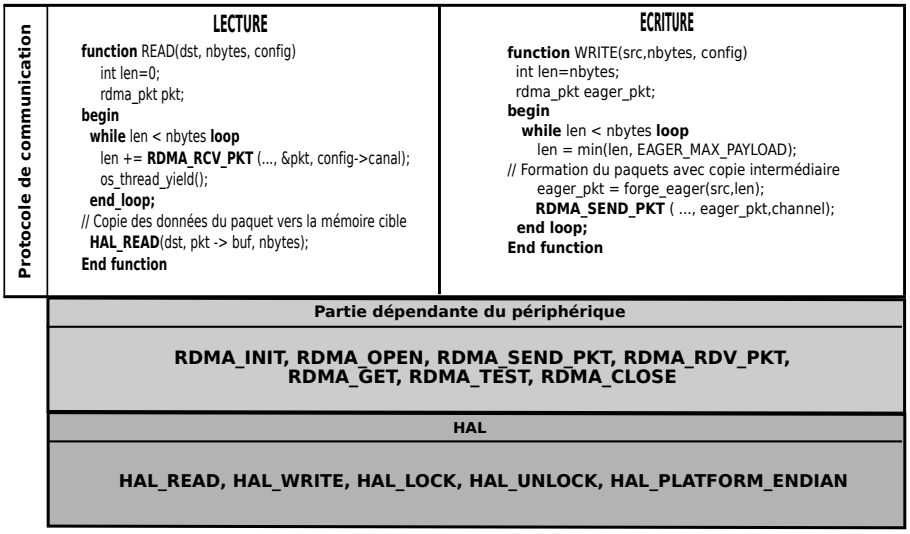

FIGURE 4.12 – Implantation du protocole Eager

Comme nous pouvons le voir, la partie dépendante du périphérique est composée de trois blocs principaux :

- La partie "Traitement des acquittements et signalisations du périphérique", qui traite tous les retours du périphérique et forme des messages qu'elle range dans différentes boîtes aux lettres. Le pilote fournit deux boîtes aux lettres par canal, une en entrée et une en sortie, chaque message dans une boîte aux lettres correspondant à une requête, une réponse ou une signalisation du périphérique concernant respectivement une opération d'émission ou de réception de données sur le réseau.
- La partie de "Programmation du périphérique" qui exporte un ensemble de fonctions (un ordre de transfert par exemple) au protocole de communication. Ces fonctions mettent en œuvre une machine à états qui doit atteindre un état final avant de rendre la main au protocole.
- La partie "Etat des transferts", qui permet d'interpréter les acquittements dans les boîtes aux lettres d'un canal, pour l'usage du protocole de communication afin de vérifier l'état d'un transfert. Ce sont également des fonctions exportées au protocole de communication mettant en œuvre une machine à états.

La partie "Traitement des acquittements et signalisations du périphérique" est commune à tous les canaux de communication utilisant le périphérique et peut être soit effectuée par un fil d'exécution séparé, soit par un traitant d'interruption lorsque le périphérique base sa signalisation sur ce mécanisme, soit régulièrement par chaque tâche au moment de faire appel aux services du périphérique. En ce qui concerne les fonctions d'interprétation des messages reçus, il peuvent mettre en œuvre une machine à états suivant les signalisations reçues, qui ne rendent la main au protocole de communication que lorsque l'un des états finaux est atteint. Les parties "Programmation du périphérique" et "Etat des transferts" se basent sur les messages d'une boîte aux lettres donnée pour faire évoluer leur machine à états.

Il est difficile de généraliser l'interface de programmation et de signalisation offerte par un périphérique de communication, en raison du nombre de concepteurs présents sur le marché. Chacun a sa propre conception de l'interface de programmation de ses propriétés intellectuelles. Nous pouvons tout de même noter en Figure 4.14 la présence de registres accessibles en lecture ou en écriture pour l'usage du pilote, nous faisons

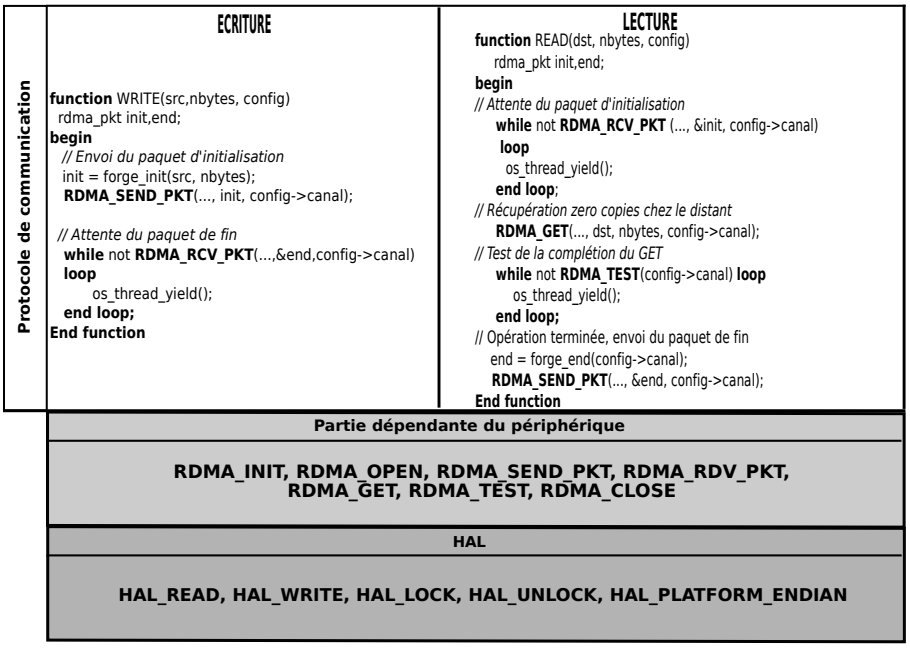

FIGURE 4.13 – Implantation du protocole Rendez-vous

également apparaître une zone de mémoire partagée qui permet également d'échanger des informations ou des données.

Il faut donc, pour certains périphériques multi-tuiles, ajouter un attribut qui définit la zone mémoire partagée au sein de la tuile :

$$Attributs(Periph_{MT}) := \text{Addresse zone memoire}, \text{Taille zone memoire}$$

4.5 Critères pour la sélection des composants de communication

Nous résumons dans cette partie de conclusion du Chapitre, l'usage qui a été fait du chemin matériel dans le modèle d'architecture pour arriver à composer les couches des différents processeurs de la plateforme. A ce stade, nous avons proposé un certain nombre de protocoles de communication, mais nous n'avons pas encore explicité comment la sélection peut s'opérer à partir des informations du modèle. Cette proposition ne tient pas encore compte de la partie *back-end* du flot, qui nous obligera à compliquer un peu la vue abstraite que nous avons adoptée jusqu'à maintenant. Nous essayons d'introduire ici les difficultés liées à la gestion des mémoires du système dans une approche de génération de logiciel comme la nôtre.

4.5.1 Communications intra-tuile

Nous avions la définition suivante pour les chemins de communication matériels intra-tuile :

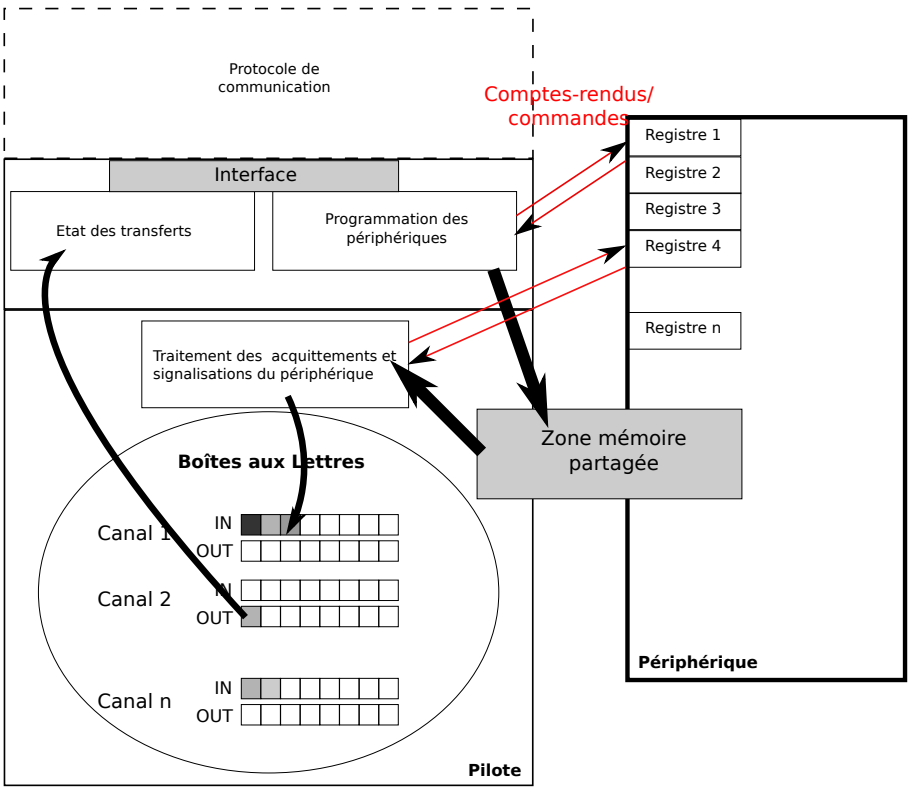

FIGURE 4.14 – Vue globale de la partie dépendante du périphérique d'un pilote

$$WritePaths_T(UC, Mem) := \{MemLocal \cdot Res_0 \cdot (Periph_i \cdot Res_{i+1})^* \cdot Mem\}$$

$$ReadPaths_T(UC, Mem) := \{MemLocal \cdot Res_0 \cdot (Periph_i \cdot Res_{i+1})^* \cdot Mem\}$$

Un premier écart avec le modèle est due à la structure en pile que nous avons choisie pour les binaires, où le HAL met en œuvre entre autres les fonctions d'accès au proche environnement du processeur. Pour cette raison, le chemin matériel en lecture ou en écriture entre $MemLocal$ (la mémoire source ou cible des passages de message) et la mémoire de communication Mem est implanté de manière optimale par les fonctions HAL_WRITE et HAL_READ, qui sont figées et ne prennent pas en compte ces chemins matériels.

Après avoir proposé un certain nombre de protocoles de communication, nous nous sommes retrouvés devant le choix suivant pour un canal C déployé sur un chemin matériel de communication intra-tuile : soit laisser le choix àl'utilisateur ou au flot du protocole à utiliser ;

$$Attribut(mapping_canal(C)) := protocole : \{\text{BUFFERISE}|\text{SYNCHRONE}|\text{READY}\}$$

soit fixer le choix de communication le plus performant par défaut, pour la plateforme considérée, le principal défaut de cette approche étant que le programmeur de l'application ne maîtrisera pas les spécificités de chaque protocole, et pourra avoir un résultat avec des interblocages ou des incohérences alors que son modèle de haut niveau est a priori correct.

A chaque protocole de communication intra-tuile a été attaché un ensemble d'attributs que nous rappelons ici :

$$Attributs(FIFO) := \text{Adresse FIFO, Taille FIFO, Identificateur Verrou}$$
$$Attributs(RENDEZ-VOUS) := \text{Adresse structure controle, Verrou Logiciel}$$
$$Attributs(\text{RENDEZ} - \text{VOUS} - \text{READY}) := \text{Adresser structure controle, Verrou Logiciel}$$

Dans tous les cas, nous voyons apparaître une adresse mémoire, qui correspond tantôt à l'adresse de la FIFO, tantôt à l'adresse d'une structure de contrôle. Celle-ci est la mémoire partagée entre l'écrivain et le lecteur pour effectuer la communication, nous l'assimilons donc à une adresse qui rentre dans la plage de Mem. En ce qui concerne l'usage de $MemLocal$, le problème est un peu plus délicat. Pour illustrer ce point, nous présentons un extrait de fonctionnalité de tâche mettant en jeu trois écritures sur un canal de communication :

```
global word a[10];

function fil1_behavior(Channel canal1)
begin
    word b[10];
    word *c =  calloc(10, sizeof(word));
...
// Envoi des 10 mots du tableau a
    channelWrite(canal1, a, 10);
// Envoi des 10 mots du tableau b
    channelWrite(canal1, b, 10);
// Envoi des 10 mots du tableau c
    channelWrite(canal1, c, 10);
end function
```

Dans cet exemple, le deuxième paramètre de la fonction *ChannelWrite* indique l'adresse mémoire où les données à envoyer sont stockées, sachant qu'il est interdit d'utiliser des adresses en dur dans ce fichier. Le paramètre utilisé est d'abord une variable globale non initialisée b, qui est donc rangée dans une section spéciale du binaire. Le second appel à la fonction *channelWrite* transmet des données à partir d'une variable locale à la fonction *fil1_behavior* dont l'emplacement mémoire est situé dans la pile d'exécution du fil alloué par le processus principal dans sa propre pile. Enfin, la dernière écriture se fait à partir d'une mémoire alloué dynamiquement, qui est également dans une section spéciale du binaire (la *heap*). Toutes ces adresses sont fixées lors de l'étape d'édition de liens !

4.5.2 Communications inter-tuiles

Tout quasiment a été dit précédemment, nous relevons juste la spécificité des chemins inter-tuiles par rapport à leur homologue intra-tuile.

$$WritePaths_{MT}(UC_i, Res_{MT}) := \{MemLocal \cdot Res_0 \cdot (Periph_i \cdot Res_{i+1})^* \cdot Periph_{MT} \cdot Res_{MT}\}$$
$$ReadPaths_{MT}(UC_i, Res_{MT}) := \{MemLocal \cdot Res_0 \cdot (Periph_i \cdot Res_{i+1})^* \cdot Periph_{MT} \cdot Res_{MT}\}$$

La même remarque est valable que précédemment par rapport au chemin au chemin connectant la mémoire cible/source $MemLocal$ et le réseau multi-tuiles Res_{MT}, qui est géré par le HAL, et le traitement de l'information $MemLocal$ dans le chemin matériel.

Nous avons introduit deux réseaux, il faut donc ajouter un attribut au réseau pour savoir quel protocole de bas niveau il supporte :

$$Attribut(Res_{MT}) := link : \text{ETHERNET|RDMA}$$

et pour chaque canal C déployé sur un chemin de communication inter-tuiles, nous avons introduit quelques protocoles :

$$Attribut(mapping\_canal(C)) := protocole : \{\text{Non connecte|Connecte|Raw}\}$$

pour un réseau supportant Ethernet et

$$Attribut(mapping\_canal(C)) := protocole : \{\text{Eager|Rendez} - \text{vous}\}$$

pour un réseau de type RDMA.

Le même choix s'impose à nous concernant le protocole, à savoir laisser le choix à l'utilisateur ou non du protocole à sélectionner pour chaque canal de communication déployé sur un chemin matériel inter-tuiles.

Enfin, pour certains périphériques qui permettent l'accès au réseau inter-tuiles, une zone mémoire peut être nécessaire :

$$Attributs(Periph_{MT}) := \text{Addresse zone memoire, Taille zone memoire}$$

Chapitre 5

Mise en œuvre d'un flot de synthèse

Sommaire

Introduction

Nous avons introduit dans le chapitre précédent notre flot de génération de logiciel ciblant des systèmes multi-tuiles hétérogènes. Les modèles de l'architecture, de l'application et du déploiement des éléments applicatifs sur les éléments de l'architecture ont été formalisés. L'étude de la partie *front-end* du flot, qui consiste en une génération du point d'entrée des tâches applicatives et en une sélection d'éléments logiciels puisés dans différentes bibliothèques, a permis de mettre en lumière un certain nombre d'écarts entre le modèle de haut niveau proposé et la formation des binaires générés par le flot. Un écart important est dû à

l'utilisation d'une structure en pile pour le binaire généré, où la couche la plus basse est une couche d'abstraction du matériel (le HAL). Le HAL propose entre autres des primitives d'entrée/sortie qui permettent aux couches supérieures d'accéder au proche environnement du processeur. Parmi ces couches supérieures, nous trouvons les composants logiciels de communication mis en œuvre sous forme de pilotes de communication, qui reposent exclusivement sur le HAL pour ces opérations d'entrée/sortie. En conséquence, les chemins matériels tels que définis à un haut niveau, qui proposent un chemin interne à la tuile pour aller d'une unité de calcul à un medium de communication, ne peuvent dans cette approche être respectés. Cela dit, aucune barrière dans notre approche n'empêche le respect de ce chemin interne, mais un travail de génération d'une partie des pilotes, non évoqué dans cette thèse, est nécessaire.

Nous avons également présenté la structure des pilotes, qui doivent mettre en œuvre un protocole de communication et pour les mécanismes de communication inter-tuiles, un ensemble de fonctions pour piloter le périphérique utilisé. Pour mener à bien la sélection des composants de communication, un certain nombre d'attributs ont été attachés aux éléments du modèle de haut niveau, en particulier le protocole de communication à attacher à un canal de communication déployé sur un chemin de communication matériel. Cependant nous n'avons pas encore délégué le choix du protocole au programmeur de l'application, nous décrirons à la fin de ce chapitre le choix qui a été fait lorsque nous commencerons à rentrer dans les détails d'implantation de notre flot. Outre le choix du protocole de communication, chaque pilote de communication présente un certain nombre d'attributs explicité pour chaque mécanisme présenté. Ces attributs sont en fait un ensemble de valeurs qui doivent être fixées une bonne fois pour toute avant la partie *back-end* de notre flot qui produit les binaires.

La présentation du Chapitre IV a introduit une vision *top-down*, où nous avons pris une vue idéale de choses pour l'écriture des composants et leurs attributs. Nous commençons dans ce chapitre par l'approche inverse, *bottom-up*, pour affiner certaines caractéristiques des composants de communication et illustrer le fait que les systèmes à base de MPSoCs hétérogènes échappent parfois aux modèles que nous essayons de leur appliquer. Ceci nous permet d'introduire le flot de génération complet, avec une phase d'automatisation de différentes configurations qui seront introduites et justifiées. Ce modèle complet du flot nous a effectivement permis d'implanter le flot et de le proposer à des programmeurs d'application qui ont réussi à l'appliquer sur des applications complexes.

5.1 Difficultés introduites par les plateformes multi-tuiles hétérogènes

Les systèmes multi-tuiles présentent un certain nombre de subtilités que nous allons illustrer dans cette partie par quelques exemples concrets. Ceux-ci concernent principalement l'écriture des composants de communication pour le système multi-tuiles basé sur un Diopsis d'Atmel. Ils peuvent avoir comme nous le verrons un impact sur le mécanisme de sélection de notre flot de génération. Ces exemples n'ont pas vocation à être universels, mais illustrent bien les difficultés de bâtir un flot aussi générique que possible autour de ces plateformes.

5.1.1 Hétérogénéité des représentations de données

La plupart des processeurs présents dans les MPSoCs disposent d'unités de calcul à virgule flottante, cette représentation étant moins coûteuse à implanter que la représentation à virgule fixe. Le Diopsis 940 ne fait pas exception à ce constat, l'ARM et le mAgicV utilisent tous deux la représentation *IEEE 754*, l'ARM en utilise la version simple sur 32 bits tandis que le mAgicV manipule une version étendue sur 40 bits comme indiqué dans le tableau ci-après.

UC	Signe	Exposant	Mantisse
ARM	1 bit	8 bits	23 bits
mAgicV	1 bit	8 bits	31 bits

$$r = (-1)^{\text{signe}} * mantisse * 2^{exposant-127}$$

L'interface esclave présentée au bus AMBA par le mAgicV est donc en charge de la conversion 40 bits-32 bits, pour cela elle fournit deux alias *DM_I* (pour le traitement des entiers) et *DM_F* (pour le traitement des flottants) situés respectivement aux adresses $0x620000$ et $0x640000$ qu'il faut alors ajouter à l'adresse source ou cible en mémoire DSP. Pour l'alias *DM_I*, l'écriture place les 32 bits dans les poids faibles du mot de 40 bits, les 8 bits de poids fort sont mis à 0 ; la lecture prend uniquement les 32 bits de poids faible du mot de 40 bits. Pour l'alias *DM_F*, l'écriture place les 32 bits dans les poids forts du mot de 40 bits, les huit bits de poids faible sont mis à zéro ; la lecture prends les 32 bits de poids fort. Un transfert de données flottantes hors du monde du DSP induit donc une perte de précision, alors qu'un transfert de données entières peut entraîner une troncature de la valeur.

En conséquence, l'accès aux données dans la mémoire interne au DSP ne se fait pas de la même manière suivant que nous avons à faire à des entiers ou à des flottants. Ceci a une conséquence pour les pilotes de communication qui visent à faire communiquer les deux unités de calcul : d'une manière ou d'une autre, le pilote doit avoir connaissance du type des données qu'il doit transférer et le traitement ne sera pas le même. Le code ci-dessous montre un extrait de la primitive *write* du pilote implantant le protocole Rendez-vous, lors de la phase où le producteur renseigne sa zone mémoire (donc l'adresse des données où le lecteur viendra récupérer les données).

```
ARM :
void write(void *src, int nwords, rdv_config_t config)
{
    ...
    prodzone -> presence = True;
    prodzone -> address = src;
    ...
}

mAgicV :
void write(void *src, int nwords, rdv_config_t config)
{
 ...
    prodzone -> presence = True;
    prodzone -> address = (config -> type == INTEGER) ? 0x620000+src : 0x640000+src;
 ...
}
```

Comme nous pouvons le voir, l'information du type de données transportées par le canal de communication est indispensable pour que l'ARM puisse lire correctement les données chez le mAgicV. La méthode présentée dans le code ci-dessus est d'incorporer l'information dans la configuration du pilote pour le canal implantant le Rendez-vous :

$$Attributs(\text{RENDEZ} - \text{VOUS}) := \text{Adresse}, \text{Verrou}, \text{Type}\ \{INTEGER|FLOAT\}$$

Il existe deux solutions pour remplir le champs "Type" de la configuration du pilote. Première solution, proposer au programmeur une interface de COM (dans *channelWrite* et *channelRead* par exemple) qui lui permet de renseigner le type de données à transférer (comme dans MPI en fait), notons que l'interface de COM ne peut transmettre l'information par le biais du système de fichiers virtuels utilisé, car celui-ci a une interface standard qui n'a aucune raison de comporter l'information. L'inconvénient principal de cette approche est la complication de l'interface COM, qui doit distinguer tous les types de pilotes pour savoir si l'information doit être passée ou pas. Ceci rend l'interface de la COM difficilement maintenable et la rend très lourde à gérer pour une unité de calcul comme le DSP.

Deuxième solution que nous adoptons dans la suite : le configurer une bonne fois pour toute dans le flot, en conséquence le canal ne pourra transférer qu'un seul type de données au cours de la durée de vie du programme. Dans ce dernier cas, c'est un attribut qui doit être incorporé au modèle de haut niveau pour le canal :

$$Attribut(Canal) := \text{Type}\ (INTEGER|FLOAT)$$

5.1.2 Possibilités offertes par le compilateur

Comme nous l'avons évoqué dans le Chapitre II de tour d'horizon, les compilateurs disponibles pour les RISCs et les DSPs n'offrent pas les mêmes possibilités, non par mauvaise volonté de la part des concepteurs du compilateur mais à cause des spécificités architecturales des DSPs. Nous évoquons ici les structures de données complexes composées de champs de types différents, par exemple des concaténations d'octet, de demi-mot (2 octets) ou de mot (4 octets sur une architecture 32 bits) qui sont très souvent utilisées dans les types de données et registres des périphériques. Le mAgicV d'Atmel propose des entiers courts sur 16 bits, mais uniquement pour un usage interne, le banc de mémoire données de 40 bits est la seule cible et source possible pour une opération vers un périphérique à l'extérieur du sous-système (sachant que seuls 32 des 40 bits sont utilisés). En conséquence, une structure de données qui a vocation à sortir du monde du mAgicV doit former des mots de 40 bits uniquement (dont 32 utiles), avec pour conséquence des opération de masquage et de décalage relativement complexes à effectuer. Nous illustrons ce point par l'initialisation de l'en-tête Ethernet faisant intervenir des adresses MAC sur 6 octets qui dépassent donc la frontière du mot et provoquent un traitement complexe chez le DSP.

Adresse MAC dst	Adresse MAC src	Type
6 octets	*6 octets*	*2 octets*

La structure de données pour définir cet en-tête pour le DSP ne sera donc pas une structure intuitive au premier abord :

```
ARM :
typedef struct __attribute__((packed)) {
   uint8_t MAC_dst[6];
   uint8_t MAC_src[6];
   uint16_t type;     // 0x0800 pour Ethernet
} eth_hdr_t;

mAgicV :
typedef struct {
   word MAC_dst_MSB;
   word mixed;
   word MAC_src_LSB
   word type;     // 0x0800 pour Ethernet
} eth_hdr_t;
```

Nous pouvons voir la catastrophe que représente cette structure de données simple pour le DSP. Non seulement une composition es nécessaire pour récupérer les adresses MAC complètes (champ *mixed* de la structure de données), mais en plus le dernier mot de la structure est partagée avec l'en-tête suivant (ARP ou IP). Ajoutons à cela que les informations dans le standard IP doivent être présentées en convention *big endian* (le mAgicV étant en *little endian*) et nous comprenons à quel point le pilote *socket* pour le mAgicV est complexe. En addition des opérations de contrôle qui pénalisent les performances du DSP, c'est une raison qui nous a poussé à ne porter que des *sockets* Ethernet en mode *Raw* pour celui-ci.

Cette caractéristique du compilateur, mais aussi d'autres particularités que nous n'évoquons pas dans ce manuscrit, nous ont poussé à opter pour deux versions de pilotes différents pour les deux processeurs de la tuile pour tous les pilotes, ce qui au départ n'était pas prévu car le HAL était censé permettre d'utiliser un et un seul pilote pour tous les processeurs de la plateforme ciblée. Le processus de sélection de composants dans notre plateforme à base de Diopsis doit donc sélectionner le pilote pour le mécanisme de communication souhaité, mais aussi pour le processeur qui exécute le pilote.

5.1.3 Ressources limitées des DSPs

Si le programmeur d'un RISC peut considérer a priori qu'il dispose d'un espace mémoire de données très grand, il n'en va pas de même pour le DSP. Le mAgicV de notre plateforme dispose d'une mémoire données très limitée de 16 Kmots de 40 bits. Nous nous sommes autorisés, en dépit de cette zone mémoire limitée, d'incorporer une gestion dynamique de la mémoire, la zone de *heap* (c'est-à-dire la zone utilisée par les fonctions comme *malloc*) étant fixée à 1 Kmot et située en fin de mémoire. Ceci signifie que les allocations en mémoire dynamique dans le DSP doivent être faites de manière très parcimonieuse. Cependant, certaines opérations requièrent parfois le recours à des allocations dynamiques dont la taille ne peut être fixée à l'avance. Si ces allocations sont trop nombreuses ou demandent une zone mémoire trop importante, le programme ne pourra continuer son cours normal.

Ceci peut arriver dans la partie dépendante du périphérique du pilote implantant le protocole RDMA Eager (pour le DNP), et plus particulièrement sa partie "Traitement des acquittements et signalisations du périphérique" (voir Chapitre IV, Figure 4.14). Le périphérique, à un moment de l'exécution du programme, envoie une signalisation comme quoi un paquet Eager d'une taille donnée est présent dans la mémoire allouée au périphérique. Une allocation et copie de données est alors nécessaire pour former le message qui sera posté dans la boîte aux lettres du périphérique adéquat afin de pouvoir dégager la zone mémoire dédiée au périphérique pour d'autres opérations. Dans le DSP, la réception d'un paquet Eager de 8 Kmots va bien

évidemment interrompre l'exécution normale du programme. Un recours à une portion de mémoire statique pour cette partie du pilote ne résoudra pas l'affaire, car il n'est pas acceptable de réserver une proportion trop importante des 16 Kmots de mémoire données et de l'enlever au programme qui a d'autres tâches à effectuer.

```
// Réception du paquet eager
    message = malloc(eager_pkt->nwords);
// Copie du paquet en mémoire interne au DSP
   HAL_READ(message, eager_pkt->payload, eager_pkt->nwords);
```

Nous ne pouvons donc autoriser un transfert de mots trop élevé pour le protocole Eager, mais ceci est également vrai pour l'interface socket qui requiert ce type de copies intermédiaires. C'est la raison qui nous a poussés à limiter le nombre de mots à envoyer par le protocole Eager, mais à proposer à l'insu du programmeur une limite au delà de laquelle le protocole Rendez-vous est utilisé. Notre pilote RDMA est donc composé des deux protocoles qui permettent d'éviter un débordement de la zone mémoire dynamique, ce choix sera d'ailleurs conforté lors de l'estimation de performances effectuée en Chapitre VI.

5.2 Flot de génération complet

Nous avons soulevé quelques subtilités inhérentes à l'hétérogénéité de la tuile, qui ont mis à l'épreuve notre modèle peut-être un peu trop idéal(iste) du Chapitre IV. Nous évoquons à présent un autre problème soulevé par les plateformes multi-tuiles hétérogènes : le besoin d'automatiser complètement le flot de génération de logiciel. Ceci nous permet d'introduire la partie de configuration automatique dans le modèle de flot que nous avons présenté précédemment.

5.2.1 La nécessité de l'automatisation du flot

Phase de spécialisation des composants logiciels

Nous illustrons la nécessité d'automatiser la phase de spécialisation des composants de communication par un simple exemple mono-tuile (voir Figure 5.1), avec le pilote implantant la FIFO logicielle introduit en Chapitre IV. Nous considérons donc deux types de canaux de communication, intra-processeur et inter-processeurs. Nous rappelons les attributs de ce pilote :

$$Attributs(FIFO) := \text{Adresse FIFO}, \text{Taille FIFO}, \text{Identificateur Verrou}$$

Nous soulignons également, comme indiqué précédemment, que les champs $Adresse\ FIFO$ et $Identificateur\ Verrou$ son superflus pour le mécanisme intra-processeur, car les ressources internes au processeur peuvent être utilisées sans besoin de configuraiton (allocation dynamique, usage de sémaphores). Pour le mécanisme inter-processeurs, le mémoire de communication partagée est basée à l'adresse 0x30A00000 et utilise des identificateurs de verrou matériel.

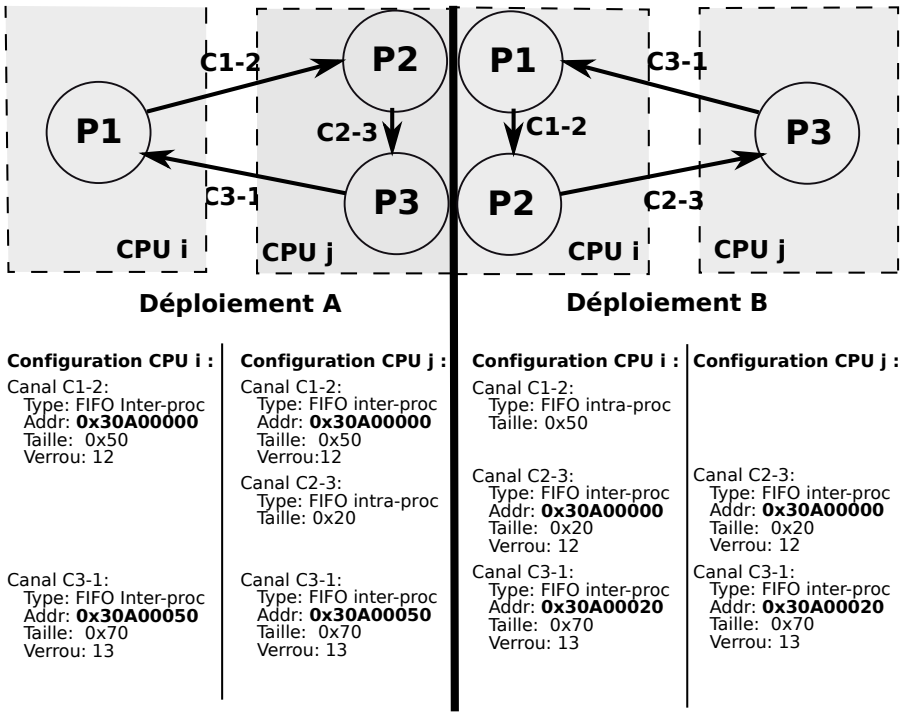

FIGURE 5.1 – Deux solutions de déploiement avec configuration des FIFOs logicielles

Nous considérons dans la Figure 5.1 trois processus P1, P2 et P3 qui sont déployés sur deux processeurs appartenant à la même tuile. Ils sont interconnectés par trois canaux de communication C1-2, C2-3 et C3-1 selon deux déploiements A et B. Dans A, P1 est placé sur le processeur CPU i tandis que P2 et P3 sont placés sur le processeur CPU j. Nous avons donc deux FIFOs inter-processeurs dans chaque processeur (aux adresses 0x30A00000 et 0X30A00050) et un mécanisme intra-processeur dans le CPU j. Quand nous changeons de déploiement et que nous passons P2 du CPU j au CPU i, la configuration de tous les pilotes s'en trouve changée. Nous avons dans le déploiement B deux FIFOs aux adresses 0x30A0000 et 0x30A00020, si nous faisons l'hypothèse que les FIFOs sont placées à des emplacements contigus dans la mémoire de communication partagée. Cet exemple montre qu'une modification simple dans le déploiement requiert de nombreux changements délicats dans la spécialisation des pilotes et souligne l'interdépendance que les communications introduisent entre les différents binaires des processeurs.

Ce pilote avec une configuration assez simple illustre bien les problèmes rencontrés quand nous considérons des systèmes multi-tuiles qui introduisent en plus de ces mécanismes intra-tuile des mécanismes inter-tuiles avec des configurations différentes. Spécialiser tous les canaux de communication à la main, en raison du nombre de pilotes introduits précédemment et du nombre de processeurs de la plateforme devient une tâche irréalisable.

Script d'édition de liens

Certains compilateurs offrent la possibilité de configurer la répartition des éléments logiciels en mémoire (voir Chapitre III). La génération automatique de la répartition des éléments logiciels dans les différentes mémoires de la plateforme cible est d'ailleurs un aspect très intéressant dans l'absolu, car les

architectures multiprocesseurs ont tendance à déléguer de plus en plus au programmeur la tâche de gérer l'emplacement mémoire de ses éléments logiciels [SYN09] (voir l'exemple du Cell).

Pour notre propos, nous revenons au problème qui a été posé en fin de Chapitre IV : la définition du chemin matériel en écriture (par exemple) puisé dans l'ensemble

$$WritePaths_T(UC, Mem) := \{MemLocal \cdot Res_0 \cdot (Periph_i \cdot Res_{i+1})^* \cdot Mem\}$$

Celui-ci fait intervenir une mémoire locale à chaque processeur, $MemLocal$, source de la communication par passage de messages. Nous avons montré par un bout de code d'une tâche contenant trois appels successifs à une primitive d'écriture, qu'il était difficile pour le flot de maîtriser la localité des variables utilisées pour l'opération d'écriture. En effet, le programmeur d'application peut utiliser des variables globales, statiques, locales, allouées dynamiquement, ces sections pouvant être située dans des mémoires différentes. Une solution pourrait être de contraindre le programmeur de l'application à déclarer ses variable dans une section spéciale du binaire, que le flot pourrait alors placer dans la mémoire conformément au chemin matériel choisi. Ceci est cependant une contrainte forte pour le programmeur qui se voit privé d'une liberté importante, sachant de plus que tous les compilateurs n'offrent pas la possibilité de configurer l'emplacement en mémoire des différentes sections du logiciel (les DSPs en général). Nous faisons donc le deuil du paramètre $MemLocal$ des chemins matériels, et nous nous replions sur une solution qui vise à proposer la meilleure répartition possible des éléments en mémoire dans le flot de génération.

Nous n'avons pas l'intention dans cette thèse d'automatiser complètement la phase de génération de script d'édition de liens pour les RISCs, mais d'en proposer un efficace pour une plateforme donnée avec tout de même quelques paramètres qui peuvent être fixés par le flot.

5.2.2 Flot incorporant les étapes d'automatisation

La Figure 5.2 introduit notre flot de génération complet.

Nous pouvons voir apparaître un module de configuration qui a deux tâches principales : d'abord, il est en charge de spécialiser les pilotes de communication choisis par le module de sélection. Pour ce faire, il dispose des informations du modèle de haut niveau, mais aussi d'informations annexes (métadonnées sur le schéma) qui lui permettent de combler les informations manquantes dans le modèle de haut niveau. Notons que celles-ci sont dépendantes du modèle de haut niveau utilisé, qui ne fournira pas le même niveau d'informations que d'autres. Par exemple, les adresses mémoires non spécifiées dans le chemin de communication comme celles allouées aux périphériques, peuvent manquer dans certains modèles très abstraits. Deuxième tâche du module de configuration : configurer les chaînes de compilation utilisées. Cela signifie d'une part qu'il faut renseigner les options passées aux compilateurs et à l'édition de liens, mais aussi générer un script d'édition de liens quand ceux-ci peuvent être pris en compte par la chaîne utilisée.

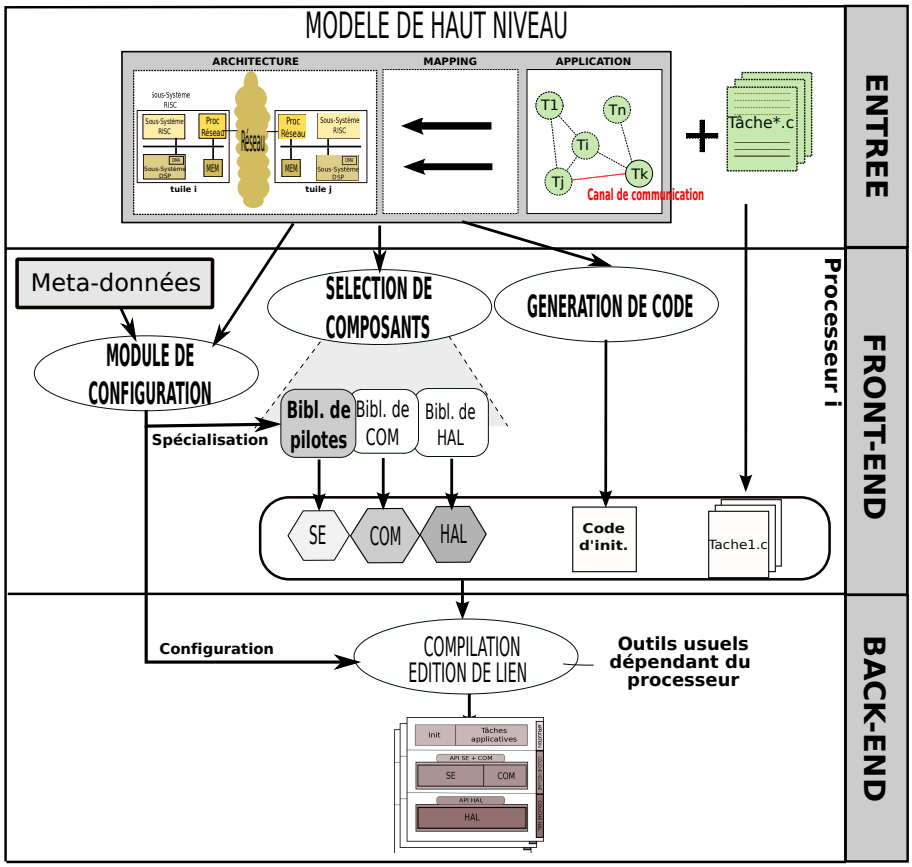

FIGURE 5.2 – Flot de génération complet

5.3 Solution d'implantation du flot

5.3.1 Difficultés

Nous introduisons à présent notre solution de mise en œuvre du flot, qui nous a permis de mener à bien les expérimentations du Chapitre VI. Plusieurs difficultés ont dû être contournées pour arriver à un flot fonctionnel :

- Gérer tout type de modèle de haut niveau, chacun n'apportant pas la même quantité de détails. Certaines données peuvent être dans quelques modèles (par exemple, des adresses physiques pour les mémoires) et en être absentes lors d'un changement de modèle d'entrée.
- Faire la distinction entre les informations qui se situent dans le modèle, celles qui doivent être gérées par le flot et celles qui peuvent être rentrées à la main.
- Sélectionner non seulement les composants de communication, mais aussi les autres éléments logiciels en fonction de l'unité de calcul visée, sachant que les informations pour sélectionner les composants du SE, par exemple, ne sont pas dans le modèle de haut niveau.
- Choisir un format de fichier pour contenir toutes ces informations supplémentaires (les *métadonnées* de la Figure 5.2) et les formats de fichiers qui contiendront les informations de manière compréhensible pour les outils du *back-end* du flot.

Ce dernier point est particulièrement important à fixer, car si le formalisme utilisé pour représenter les données absentes du modèle de haut niveau sont une affaire interne au flot, l'apport des informations manquantes pour l'étape d'édition de liens finale doit être compréhensible par les outils du *back-end* et comme nous l'avons vu, ceux-ci peuvent être très différents d'un processeur à l'autre.

5.3.2 Organisation de la suite d'outils et éléments considérés

Notre flot de génération s'appuie sur l'environnement de composants logiciels APES [GP09], qui inclut le système d'exploitation DNA. Celui-ci se présente sous forme d'un ensemble de fichiers C, dont la compilation est effectuée en fonction d'un certain nombre de variables d'environnement, mais aussi d'un ensemble de variables globales à fixer avant l'édition de liens. Les pilotes de communication présentés en Chapitre IV ont bien entendu été intégrés à l'environnement pour notre plateforme multi-tuiles, les structures de configuration devant être renseignées en C pour chaque canal utilisant le pilote.

La Figure 5.3 décrit plus en détails les différents modules du flot ainsi que les fichiers intermédiaires requis pour arriver au binaire final.

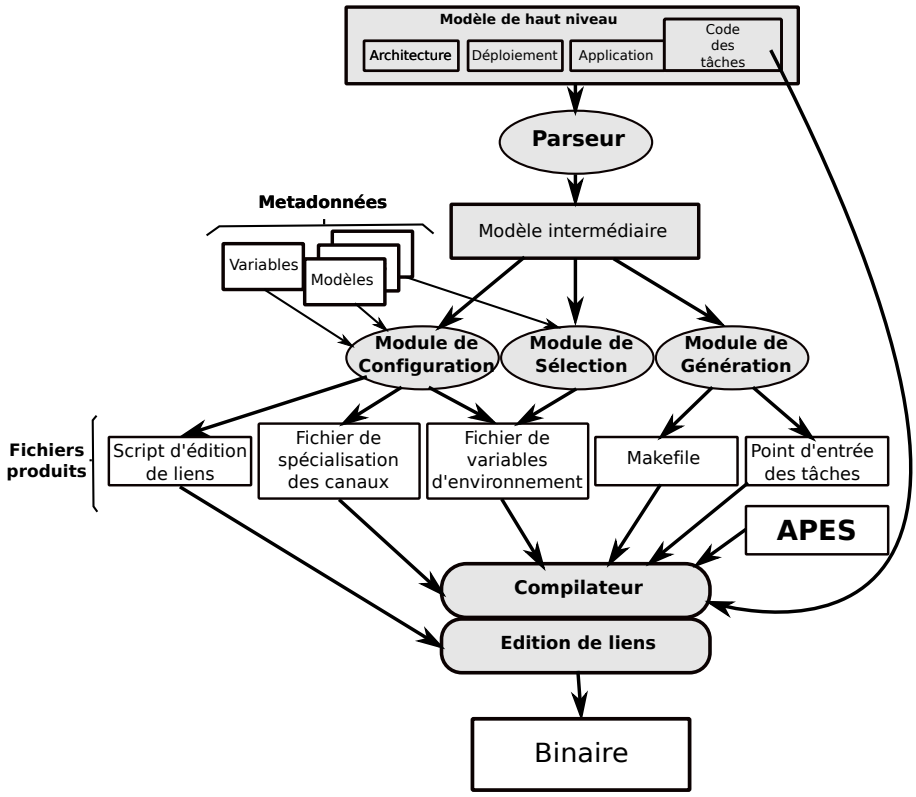

FIGURE 5.3 – Détails d'implantation du flot avec la bibliothèque de composants APES

Nous reconnaissons l'entrée et la sortie du flot, à savoir le modèle de haut niveau avec les fonctionnalités des tâches applicatives et à l'autre bout un binaire pour l'un des processeurs de la plateforme. Nous voyons aussi apparaître la bibliothèque de composants APES, et les métadonnées introduites précédemment sous formes d'un fichier "variables" et de fichiers "modèle", chacun représentant la structure d'un fichier de configuration à générer. Le fichier "variables" permet de résoudre les valeurs de certains paramètres pré-

sents dans les fichiers "modèle". Un *Parseur* commence par parcourir les trois modèles de l'application, de l'architecture et du déploiement pour former un modèle intermédiaire qui sera utilisé pour former tous les binaires. Ce modèle intermédiaire est au format XML, proche du SERIF [CP08] et regroupe toutes les informations du modèle dans un seul et unique objet à disposition des autres modules du flot. Il permet de supporter tout type de modèle de haut niveau, seul un nouveau module de traduction devra être implanté lors de l'introduction d'un nouveau modèle.

Pour la partie *front-end*, nous reconnaissons le module de *Configuration* qui comme nous le verrons dans la partie suivante doit générer le script d'édition de liens et la partie de spécialisation des canaux de communication. Le module de *Sélection* génère un fichier de variables d'environnement pour la bibliothèque de composants APES qui détermine les composants de la couche logicielle (COM, HAL, SE) mais aussi les modules du SE (ordonnanceur, gestion des fils, ...). Enfin, le module de *Génération* génère le point d'entrée de l'application, et aussi un Makefile générique qui grâce aux variables d'environnement permet de compiler toutes les sources de APES et de l'application. Ceci est effectué par le *Compilateur* (partie *back-end* du flot), et toujours grâce au Makefile, l'*Edition de liens* utilisant (ou pas) un script peut produire le binaire final.

5.3.3 Module de configuration automatique

Détails d'implantation

Nous rentrons un peu plus dans le détail du module de configuration automatique, qui est le plus délicat à concevoir. La Figure 5.4 montre plus en détails le principe du module de *Configuration*.

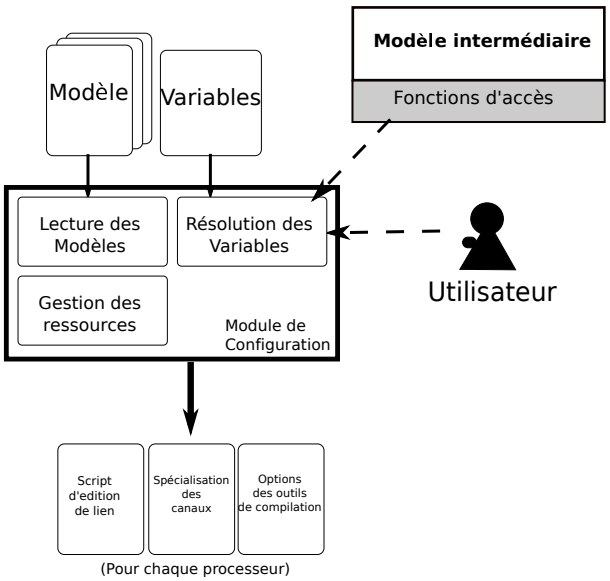

FIGURE 5.4 – Structure du module de configuration

Le module de configuration du flot prend en entrée des fichiers "modèle", qui permettent de capturer la structure du fichier à générer. Il produit trois fichiers :

- Un fichier de variables d'environnement pour mettre en place les arguments des compilateurs et de l'éditeur de liens. Celui-ci est utilisé par le Makefile générique pour construire le binaire.
- Un fichier C de spécialisation des pilotes, qui configure tous les canaux de communication pour chaque binaire de la plateforme.
- Le script d'édition de liens.

Les modèles d'entrée incluent des variables dont la valeur est à définir. Pour cela, le module de "Résolution des variables" a recours au fichier "variables" fourni en entrée qui propose une classification de toutes les variables inclues dans les fichiers "modèle". Une syntaxe particulière est utilisée dans ce fichier pour distinguer les valeurs à utiliser en fonction du processeur et de la tuile qui sont en train d'être considérés par le flot. Quatre catégories de variables sont définies.

- Les variables avec une valeur codée en dur. Ces variables sont en général des informations absentes dans le modèle de haut niveau considéré mais qui peuvent être présentes dans d'autres, par exemple les plages d'adresses physiques.
- Les variables à rentrer par l'utilisateur (avec un prompt au moment de l'exécution de l'outil), avec une valeur par défaut. Cette valeur par défaut sera utilisée lorsque l'utilisateur ne renseigne pas une valeur ou que le mode automatique de l'outil est enclenché.
- Les variables dont la valeur est résolue par un appel de fonction dans une bibliothèque liée au modèle intermédiaire (celui obtenu par le *Parseur* du flot), les noms de la fonction et de la bibliothèque étant spécifiés. Notons que chaque pilote, chacun avec une configuration différente, va donner lieu à l'écriture d'une nouvelle bibliothèque liée au modèle intermédiaire.
- Les variables qui doivent faire appel au "Gestionnaire des ressources matérielles".

Le gestionnaire des ressources matérielles a la responsabilité, lorsque le module de résolution des variables le lui demande, de réserver un espace mémoire (pour le medium de communication dans les mécanismes intra-tuile par exemple) ou des verrous matériels. Reprenons l'exemple du chemin en écriture :

$$WritePaths_T(UC, Mem) := \{MemLocal \cdot Res_0 \cdot (Periph_i \cdot Res_{i+1})^* \cdot Mem\}$$

C'est donc le gestionnaire qui va s'arranger pour donner une adresse de base à un mécanisme de communication intra-tuile qui rentre dans la plage d'adresse de Mem. Nous respectons (enfin) un paramètre du chemin intra-tuile.

Cette solution d'implantation, qui a le mérite d'exister, déporte beaucoup d'informations précieuses dans les fichiers "modèle" en entrée. En particulier, la structure du script d'édition de liens et des variables d'environnement pour mettre en place les options des compilateurs est une étape cruciale qui doit être mise en œuvre par un spécialiste des chaînes de compilation mises en jeu et de la plateforme matérielle. Une étape d'optimisation plus fine pourra supposer une modification des fichiers "modèle", avec le risque d'erreur que présente la modification de fichiers comme le script d'édition de liens.

Exemples de fichiers générés

Nous donnons un petit exemple de fichiers générés ci-après, sachant que nous avons déjà fourni au Chapitre III une illustration du script d'édition de liens. Tout d'abord, nous nous intéressons au fichier définissant des variables d'environnement pour le Makefile d'un ARM de notre plateforme.

```
export TARGET_CC="arm-gcc"
export TARGET_CFLAGS="-g -Wall  -O3 -mfloat-abi=soft -mfpu=fpa --std=c99 -mcpu=arm926ej-s"
export TARGET_LD="arm-gcc"
export TARGET_LDFLAGS="-mfloat-abi=soft"
export TARGET_ARFLAGS="cru"
export TARGET_AR="arm-ar"
export TARGET_RANLIB="arm-ranlib"
export TARGET_LDSCRIPT="-lm -T ./ldscripts/arm"
```

Il définit des options de compilateurs familiers aux programmeurs utilisant les outils GNU. Nous pouvons souligner que le *TARGET_LDSCRIPT* renferme un chemin donné à l'éditeur de liens pour ouvrir son script généré. Nous passons à présent au fichier de spécialisation des pilotes dont voici un petit extrait :

```
// Configuration des canaux utilisant le rendez-vous intra-tuile
uint32_t CHANNEL_D940_RDV_NDEV = 3;
uint32_t CHANNEL_D940_RDV_DEVICES [3][3] = {
 /* Addr     Verrou     Entier/Reel    */

//Configuration du canal "/dev/d940_rdv.0"
     {0x301000, 0,        0},
 //Configuration du canal "/dev/d940_rdv.1"
     {0x301020},1,        1},
 //Configuration du canal "/dev/d940_rdv.2"
     {0x301040},2,        0}
}

// Configuration des canaux utilisant le RDMA intra-tuile
uint32_t CHANNEL_D940_RDMA_NDEV = 2;
uint32_t CHANNEL_D940_RDMA_CONFIG[2][4] = {
  /* Id canal   Rang destination   CPU destination   Entier/Reel */

 // Configuration du canal "/dev/d940_rdma.0"
   {1,                1,              1,              0},
 // Configuration du canal "/dev/d940_rdma.0"
   {1,                4,              2,              0},
}


//Informations topologiques locales pour le RDMA
                         /* rang local   cpu local */

uint32_t RDMA_LOCAL_SETTINGS[2] = {0,   1};


//Informations spécifiques au périphérique RDMA utilisé
uint32_t RDMA_PHYSICAL_SETTINGS[3] =
/* Verrou   Adresse Memoire   Taille de memoire */
   {   4,        0x2000,             0x100  };
```

Ce fichier C généré spécialise trois canaux de communication intra-tuile utilisant le protocole Rendez-vous, et deux canaux inter-tuiles utilisant le protocole RDMA Rendez-vous, en affectant une valeur aux attributs de chacun des protocoles et du périphérique mis en jeu pour le RDMA (données locales et espace mémoire pour le périphérique inclus). Toutes les variables définies sont utilisées dans les pilotes de communication et reflètent le modèle de haut niveau en ce qui concerne l'agencement des canaux du modèle applicatif. En l'occurrence, le fichier virtuel "*/dev/d940_rdv.2*", ouvert dans le fichier d'initialisation des tâches applicatives généré par ailleurs, utilise la mémoire partagée basée à l'adresse 0x301040, le verrou matériel numéro 2 et doit transporter des entiers. L'adresse 0x301040 est dans la plage d'adresses de la mémoire définie dans le chemin matériel de communication sur lequel le canal est déployé, et a été générée par le *Gestionnaire des ressources matérielles*.

5.3.4 Module de sélection

Nous décrivons plus brièvement le fonctionnement du module de sélection du *front-end*, car celui-ci fonctionne de manière analogue au module de configuration automatique. Il est en effet également basé sur un fichier "modèle" qui définit la structure du seul fichier à générer, à savoir un fichier de variables d'environnement qui sélectionnent le HAL, la COM, le SE. La COM et le SE sont pour le moment fixés et le HAL est sélectionné en fonction de la plateforme et du processeur cibles. Ce fichier définit également les modules du SE à utiliser par des variables d'environnement interprétées par le Makefile générique généré par le module de *Génération* du flot.

Le choix des modules du SE est fixé en dur pour chaque processeur d'une plateforme donnée. Si nous prenons l'exemple de notre plateforme multi-tuiles basée sur un Diopsis 940, nous avons activé tous les modules possibles sur les ARMs, et avons limité le choix au strict minimum pour les DSPs comme indiqué dans cet extrait de fichier généré (très simplifié).

```
ARM ;
# Selection des éléments de APES
 export TARGET_OS="SYSTEM_KSP_OS_DNA"
 export TARGET_COM="SYSTEM_COM_DOL"
 export TARGET_HAL="SYSTEM_D940_ARM9"

# Selection des composants du SE
export DNA_MODULES="kernel uart timer exceptions  vfs drivers thread ..."
export DNA_SCHEDULING="round-robin"
export DNA_DRIVERS="d940_intra_rdv d940_inter_rdv  d940_socket"


mAgicV :
# Selection des éléments de APES
 export TARGET_OS="SYSTEM_KSP_OS_DNA"
 export TARGET_COM="SYSTEM_COM_DOL"
 export TARGET_HAL="SYSTEM_D940_MAGICV"

# Selection des composants du SE
export DNA_MODULES="kernel vfs drivers"
export DNA_SCHEDULING="static"
export DNA_DRIVERS="d940_intra_rdv_ready d940_inter_rdv  d940_rdma"
```

Nous pouvons souligner que l'ordonnancement statique fixé pour le DSP a une conséquence sur le choix du pilote pour les communications internes au DSP (pilote *d940_intra_rdv_ready* en l'occurrence). En effet, comme il n'y a pas de système de changement de contexte, une tâche issue du modèle de haut niveau ne peut rendre la main qu'à la fin de son exécution, il n'est donc pas possible, dans les communications internes au DSP, d'avoir des appels bloquants. C'est à l'utilisateur de fixer l'ordre d'exécution de ses tâches (dans l'ordre où les processus du haut niveau sont déclarés) pour assurer des communications internes correctes.

5.4 Bilan de l'implantation et travaux futurs

Nous résumons ici les différentes caractéristiques de l'implantation du flot que nous avons proposées dans la partie précédente. Nous faisons un petit résumé des paramètres qui ont été fixés dans cette implantation et ceux au contraire pour lesquels le flot laisse une certaine liberté de choix. Nous revenons ensuite sur la manière dont les différents attributs ajoutés aux éléments du modèle de haut niveau ont été exploités afin

d'arriver à synthétiser correctement les canaux de communication du modèle applicatif. Enfin, nous parlons des améliorations qui peuvent être apportées au flot et à la mise en œuvre proposée.

5.4.1 Gestions des différentes dépendances

Le flot de génération repose sur beaucoup d'éléments nécessaires à la production des binaires. Nous résumons ici les aspects de l'implantation qui sont modulaires et ceux qui le sont moins. Nous commençons par le modèle de haut niveau. Nous avons imposé un nombre minimal d'informations au Chapitre IV pour les modèles d'architecture, d'application et de déploiement. Cependant ces informations ne sont pas suffisantes pour construire les binaires. En utilisant un modèle intermédiaire qui découple le modèle de haut niveau des autres modules du *front-end*, nous rendons a priori le flot indépendant du modèle de haut niveau utilisé. Certaines informations cruciales peuvent toutefois faire défaut (comme les plages d'adresses mémoire ou les modules du système d'exploitation à utiliser), ces valeurs devant être renseignées à la main dans le fichier "variables", les données qui sont présentes donnent lieu à un appel de fonction qui permet d'extraire la valeur fournie. Pour le chapitre d'expérimentations, nous nous sommes basés sur un modèle DOL, où la définition des chemins matériels a été intégré au fichier d'architecture.

Nous avons ensuite l'environnement de composants logiciels qui a été plus ou moins imposé dans la partie précédente. Les dépendances introduites dans l'implantation sont ici plus nombreuses : le fichier de variables d'environnement qui sélectionne les composants logiciel et le Makefile généré sont des fichiers exploitables uniquement par l'environnement APES, tout autre environnement va requérir d'autres formats pour la configuration pour lesquels il faudra écrire les fichiers "modèle" correspondants le jour ou un autre environnement sera choisi. Plus gênant, les éléments logiciels de communication utilisés pour la synthèse sont finalement des composants de l'environnement APES, et possèdent en conséquence une interface dépendante du SE utilisé (DNA en l'occurrence). Une réécriture de tous les pilotes sera requise en raison de cette interface lors d'un changement de SE.

Enfin, nous évoquons la dépendance du flot à la plateforme ciblée qui est finalement la plus cruciale. En effet, proposer un modèle d'architecture à haut niveau doit permettre sur le papier un changement de la plateforme cible plus aisé, des modifications sont cependant nécessaires à l'intérieur du flot. Une nouvelle fois, les pilotes de communication inter-tuiles possédant une interface dépendante d'un périphérique doivent être réécrits (si la nouvelle plateforme considérée possède un service équivalent, on pense en particulier au mécanisme RDMA). Les outils de compilation utilisés peuvent également changer, et c'est dans ce cas les fichiers générés par le module de *Configuration* qui doivent avoir un fichier "modèle" modifié. L'environnement de composants logiciels doit bien entendu disposer des HAL de chaque processeur de la nouvelle plateforme. Enfin, les plages d'adresses, si elles étaient renseignées à la main dans le fichier "Variables", doivent être changées pour refléter l'agencement mémoire de la nouvelle plateforme cible.

5.4.2 Retour sur le traitement des communications

Nous revenons une dernière fois sur la définition des chemins matériels de communication introduits dans le modèle d'architecture :

$$WritePaths_T(UC, Mem) := \{MemLocal \cdot Res_0 \cdot (Periph_i \cdot Res_{i+1})^* \cdot Mem\}$$

$ReadPaths_T(UC, Mem) := \{MemLocal \cdot Res_0 \cdot (Periph_i \cdot Res_{i+1})^* \cdot Mem\}$

$WritePaths_{MT}(UC_i, Res_{MT}) := \{Mem_0 \cdot Res_0 \cdot (Per_i \cdot Res_{i+1})^* \cdot Periph_{MT} \cdot Res_{MT}\}$
$ReadPaths_{MT}(UC_i, Res_{MT}) := \{Mem_0 \cdot Res_0 \cdot (Per_i \cdot Res_{i+1})^* \cdot Periph_{MT} \cdot Res_{MT}\}$

Nous avons dans le chapitre précédent fait le deuil du chemin interne à la tuile pour aller de la mémoire locale au medium de communication (mémoire partagée ou réseau), en raison de la structure de pilote adoptée qui repose exclusivement sur le HAL pour accéder aux éléments de la tuile. En début de chapitre, nous écartions aussi la mémoire destination ou cible du passage de messages, $MemLocal$, car cela aurait imposé des contraintes trop fortes pour le programmeur des tâches applicatives.

Pour les autres éléments, nous avons attaché des attributs pour permettre la sélection et la configuration des composants de communication adéquats que nous rappelons ici. Pour un canal C reliant deux processus déployés sur la même tuile :

$$Attribut(mapping\_canal(C)) := protocole : \{\text{BUFFERISE|SYNCHRONE|READY}\}$$

Nous avons imposé le protocole "Ready" pour chaque canal intra-DSP, en raison de l'ordonnancement statique choisi dans le système d'exploitation de celui-ci. Pour notre premier prototype, le protocole "Synchrone" a été choisi et codé en dur dans les modules de *Sélection* et de *Generation* car les performances se sont révélées bien meilleures que le mode ***Bufferisé*** (voir Chapitre VI) et les caractéristiques moins dangereuses que le mode ***Ready***.

Les attributs pour les différents protocoles ont été définis comme suis :

$Attributs(FIFO) :=$ Adresse FIFO, Taille FIFO, Identificateur Verrou

$Attributs(RENDEZVOUS) :=$ Adresse structure controle, Verrou Logiciel, ***Type***

$Attributs(\text{RENDEZVOUS} - \text{READY}) :=$ Adr structure controle, Verrou Logiciel, ***Type***

Les attributs pour ces protocoles ont été simplifiés pour les canaux de communication intra-processeur, car les ressources offertes par l'environnement d'exécution (SE et bibliothèques standard du C) permettent de s'affranchir d'une adresse mémoire en dur et de verrous matériels.

$Attributs(FIFO) :=$ Taille FIFO

$Attributs(RENDEZVOUS) := \oslash$

$Attributs(\text{RENDEZVOUS} - \text{READY}) := \oslash$

Le module de *Configuration* permet de générer les configurations attendues par chacun des canaux de communication intra-tuile en fonction du protocole choisi en amont par le flot.

Si nous passons à présent à un canal C déployé sur un chemin matériel inter-tuiles, nous avons défini :

$$Attribut(Res_{MT}) := link : \{\text{ETHERNET|RDMA}\}$$

Plusieurs protocoles ont été définis pour un canal C utilisant les services d'un réseau Ethernet :

$$Attribut(mapping\_canal(C)) := protocole : \{\text{Non connecte}|\text{Connecte}|\text{Raw}\}$$

et pour un canal C utilisant les services d'un réseau RDMA :

$$Attribut(mapping\_canal(C)) := protocole : \{\text{Eager}|\text{RendezVous}\}$$

Pour un canal connectant à une ou aux deux extrémités une tâche déployée sur un DSP, les protocoles "Raw" et un mélange "Eager/RDMA" (Eager pour une quantité de données à transférer en dessous d'une limite donnée) sont choisis respectivement pour les réseaux Ethernet et RDMA. Pour les autres canaux, nous avons également penché pour un mélange "Eager/RDMA" et avons laissé le choix à l'utilisateur (par un champ de configuration du pilote Ethernet à fixer à la main) entre les modes "connectés" (UDP) et "non connectés" (TCP), pour bénéficier des avantages offerts par les protocoles IP (communication possible avec un PC). Tous ces choix seront discutés dans le Chapitre suivant.

5.4.3 Travaux futurs

L'un des principaux problèmes de notre flot de génération de logiciel est la dépendance des pilotes de communication au SE utilisé, à la plateforme cible (pour les pilotes de périphérique) et au HAL. Comme nous l'avons vu, ces dépendances se traduisent par deux maux :

- Un manque de portabilité, qui implique que chaque pilote dépend d'un SE et d'un périphérique donné, tout changement de l'un d'eux condamnant le mainteneur du flot à réécrire de nouvelles versions et à alourdir le module de *Sélection*.
- Un non respect du chemin matériel tel que défini à haut niveau, en raison de l'utilisation de primitives du HAL fixées à l'avance.

La génération d'un pilote de communication entier est une tâche très difficile. Cependant, il est envisageable de générer une ou plusieurs parties, en particulier les interfaces avec le SE et le périphérique et l'accès au medium de communication en fonction de la spécification du chemin matériel. Ceci est une première piste de continuation du travail de thèse. Il serait alors envisageable d'avoir le cœur de pilote fixé pour chaque protocole intra-tuile et inter-tuiles introduits précédemment, avec une partie de génération supplémentaire dans le flot en mesure de générer les différentes interfaces manquantes (SE, Périphérique, HAL) à partir d'un fichier de description très détaillé (nous pensons à une description IP-XACT). Ainsi, les composants de communication logiciels ne seraient plus des éléments d'un environnement comme APES, mais des composants logiciels du flot à part entière.

Une seconde piste a été décrite dans le tour d'horizon du Chapitre III. Générer un modèle intermédiaire évolué comme le *Bip* [BSS09] permettrait de vérifier certaines propriétés du modèle d'application. La construction de ce type de modèles à partir d'un langage en *Y-Chart* nécessite le parcours des fichiers d'implantation des tâches pour former un modèle intermédiaire, il est alors tout à fait possible d'imposer un respect du chemin de communication matériel complet (nous pensons en particulier à la mémoire locale $MemLocal$). Une génération à partir d'un modèle de vérification formelle imposerait un certain nombre de changements dans l'implantation du flot qu'il serait intéressant d'explorer.

Pour en revenir à l'implantation du flot sous forme d'outils, nous pouvons constater que nous avons choisi une représentation des métadonnées, c'est-à-dire les données absentes du modèle de haut niveau,

d'une manière peu réutilisable à cause du manque de formalisme de l'approche par les fichiers "modèle". Une autre piste de continuation serait d'introduire pour ces données un formalisme plus fort comme dans les travaux proposés par [MPB09] où toute une sémantique en XML est bâtie pour décrire tous les champs nécessaires à la génération de fichiers de configuration pour une plateforme à base de NoCs.

Enfin, en ce qui concerne la génération du script d'édition de liens, nous avons proposé une répartition "raisonnable" des sections logicielles dans les mémoires de la plateforme mais statique. Cette approche convient pour la tuile que nous avons considérée, car celle-ci dispose d'un système de caches élaboré qui nous a permis d'améliorer grandement les performances de nos pilotes et de nos applications. Cependant, passer sur une vraie plateforme EMM (Explicit Memory Management) comme le Cell nous permettrait d'inclure dans le flot des éléments qui permettent une gestion intelligente en logiciel de la répartition du programme sur les mémoires en cours d'exécution.

Chapitre 6

Expérimentations

Sommaire

Introduction

Notre flot de génération a été présenté dans son intégralité et une mise en œuvre à base d'outils a été proposée et implantée. Nous fournissons dans cette partie d'expérimentations quelques mesures et discussions qui permettent de justifier les choix effectués en fin de Chapitre V. Nous présentons en particulier quelques estimations de performance pour le système multi-tuiles basé sur un Diopsis 940 présenté au Chapitre I. Deux grandes difficultés se sont présentées à nous dans la phase de mesures une fois le flot de génération mis en place. Tout d'abord, contrairement à beaucoup d'autres travaux dans le domaine des supercalculateurs qui disposent d'un banc d'applications de *benchmark* portés sur MPI, nous ne disposons pas d'applications de référence portées en réseaux de processus qui nous permettent de nous positionner par rapport aux environnements logiciels embarqués comme MPI. Nous devons donc nous contenter de mesures spécifiques montrant les performances de nos composants logiciels. Une seconde difficulté qui découle de la première, il a été très difficile pour nous de trouver des travaux analogues, en particulier d'autres études d'architectures avec une fréquence de fonctionnement faible et un périphérique attaché sur le bus de communications. Les mesures de performances que nous présentons pour notre plateforme sont donc difficiles

à comparer à d'autres travaux. Par exemple, [LJW+04] montre l'implantation de la partie zero copies (donc avec les protocoles RDMA) de MPI sur un nœud de calcul à base de Xeon 2,40GHz et présente les latences et bande passante pour un canal RDMA, une bande passante qui atteint l'ordre des 800Mbits par secondes avec l'aide d'un bus Infiniband de haute performance. Nous ne pouvons pas encore, bien entendu, rivaliser avec ces architectures, mais proposons nos estimateurs de performance principalement pour justifier certains choix qui ont été faits dans la gestion des communications de notre flot.

Nous finissons ce chapitre par quelques études de cas, ou plusieurs applications complexes ont été portées par des membres du projet européen SHAPES [SHA] non spécialistes du logiciel de bas niveau, Nous ne rentrerons pas trop dans les détails des applications elles-mêmes (que nous n'avons pas écrites), mais décrivons les caractéristiques de chaque application et le retour d'expérience du point de vue du support pour le flot sur ces différentes mises à l'épreuve de nos outils.

6.1 Tailles de pile

Nous donnons les tailles de piles en Kmots de nos différents composants de communication logiciels, pour la section programme uniquement. Nous ne présentons pas la taille des sections données, car celles-ci dépendent de beaucoup de paramètres (configuration du SE, taille des boîtes aux lettres des pilotes, interface avec le SE) qui n'apportent pas d'informations supplémentaires. Nous rappelons que le mot instruction fait 32 bits pour l'ARM9 et 128 bits pour le mAgicV, sachant que la mémoire instructions du mAgicV fait 8 Kmots.

Tailles de piles en Kmots				
Element logiciel	**ARM**		**mAgicV**	
	protocole	*périphérique*	*protocole*	*périphérique*
COM + SE + HAL	14.5		0,7	
Communications intra-tuile				
FIFO logicielle	0,63		1,08	
Rendez-vous	0,05		0,83	
Rendez-vous Ready	0,05		0,79	
Communications inter-tuiles				
Ethernet	2,16	0,5	1,52	0,51
RDMA Eager+Rendez-vous	0,3	1,33	0,58	3,09

TABLE 6.1 – Tailles de nos composants logiciels de communication (section programme)

Tout d'abord, nous pouvons constater qu'un SE minimal a été utilisé pour le DSP, car la taille de pile pour l'ensemble COM+SE+HAL est moins importante que pour tous les autres pilotes présentés, ce qui n'est pas le cas pour l'ARM. Pour ce qui est des mécanismes de communication intra-tuile, nous présentons les trois modes bufferisé (FIFO logicielle), synchrone (Rendez-vous) et *ready* (Rendez-vous en mode ready) pour des communications entre proceseurs différents, sachant que nous avons écrit leur équivalent pour les communications intra-processeur et que ceux-ci présentent des tailles de pile similaires. Cependant, le mode ready étant assez dangereux à proposer au programmeur d'application, nous ne nous intéresserons pas à des caractéristiques plus poussées pour ce mode. La FIFO logicielle, aussi bien pour l'ARM que pour le DSP, a une taille plus élevée que le Rendez-vous, ce qui se ressent beaucoup plus chez le DSP en raison du nombre d'opérations de contrôle supplémentaires nécessaires pour la gestion du tampon circulaire.

En ce qui concerne les mécanismes de communication inter-tuiles, nous considérons un seul pilote par unité de calcul pour le protocole Ethernet : le pilote Ethernet pour l'ARM comprend les protocoles de la

couche transport TCP, UDP et Raw, le protocole à utiliser est sélectionné par un champ de configuration du pilote ; celui pour le DSP ne comporte que le mode Raw, sans couche transport. Le module de communication avec le périphérique (MAC+PHY) est relativement léger, car il ne comporte pas de machine à états, basiquement les tampons intermédiaires doivent être préparés par le logiciel dans une zone mémoire allouée au périphérique avec toutes les en-têtes Ethernet, IP et transport préparées (l'adresse MAC de destination devant être au préalable récupérée par le protocole ARP). Le contrôle du transfert s'effectue ensuite par une programmation et scrutation des registres du périphérique.

En revanche, le DNP dispose d'une interface logiciel/matériel plus complexe, avec une complétion à base de huit mots à aller lire dans un tampon circulaire en mémoire partagée, qu'il faut interpréter et traduire pour le mettre dans la boîte aux lettres adéquate. Cet état de fait explique les tailles de piles extrêmement élevées pour l'ARM mais surtout pour le DSP, dont la partie dépendante du périphérique occupe à peu près 1,6 kMots, soit à peu près le quart de la taille de la mémoire programme. Nous voyons que la partie protocole du pilote est relativement légère, ce qui explique que nous ayons fait le choix d'embarquer dans un seul pilote les deux protocoles Eager et Rendez-vous.

Le code suivant donne une idée du coût que représente une opération de contrôle dans le DSP, par le désassemblage de la partie dépendante du périphérique du pilote RDMA, plus précisément la partie du pilote qui scrute les registres du DNP pour remplir les boîtes aux lettres. Nous remarquons que seule une instruction sur les quatre que comporte le mot d'instruction de 128 bits est utilisé, cette sous-utilisation des capacités VLIW du DSP explique la forte empreinte mémoire du pilote RDMA évoquée précédemment.

```
dnp_event_handler :
719
        ...

912    FL66=0x4 : - - : - - : - - - -
913    5.A=0x500 : - - : - - : - - - -
914    CALL  4651 : - - : - - : - - - -
915    RFL65=0xffffff : - - : - - : - - - -
916    - : - - : TMP1=15.A+56 - : - - - -
917    RFL64=0x700110 : - - : 0.A=TMP1 - : - - - -
918    RFL64=0x700114 : - - : - - : - - - -
919    RFL65=0xffffff : - - : - - : - - - -
920    2.A=0xffff : - - : - - : - - - -

        ...

1840   RETURN : - - : - - : - - - -
1841        - : - - : 15.A=15.A+76 - : - - - -
1842        - : - - : - - : - - - -
 .label void_dnp_event_handler__end
1843        - : - - : - - : - - - -
```

Nous pouvons voir par l'étude de ces tailles de pile, qu'embarquer le pilote RDMA pour le DSP implique déjà un sacrifice de pratiquement 50% de la taille totale du programme mémoire du DSP, les déploiements qui choisissent d'implanter des communications inter-tuiles sur un réseau RDMA à partir ou vers un DSP devront donc être très prudentes sur la taille de leur application sur le DSP.

6.2 Estimation de performances

Les estimations de performances dont les résultats sont résumés ci-après se basent toutes sur la même application de test, composée de deux fils d'exécution *fil1* et *fil2* connectés par deux canaux de sens opposés. Le *fil1* initie une écriture sur le premier canal et lit ensuite sur le second, et vice et versa pour *fil2*. Seul le déploiement des tâches et des canaux et les protocoles utilisés changent d'une expérimentation à l'autre.

Les fréquences utilisées sont assez basses pour les éléments de nos tuiles (de l'ordre de la centaine de mégahertz), nous ne choisissons donc pas l'unité habituelle pour mesurer la bande passante, à savoir le nombre de bits par seconde transférés, mais le nombre de cycles de bus requis en moyenne pour le transfert d'un mot, ce avec des tailles de données croissantes. L'estimation, dans tous les cas, s'est basée sur un échange de vingt messages par taille considérée, moyennée pour obtenir les valeurs présentées.

Nous noterons que toutes les expérimentations se sont effectuées avec un ou deux Diopsis 940, seules les expérimentations mettant en jeu le DNP (non encore fondu à l'heure de l'écriture de cette thèse) se sont basées sur un simulateur précis au niveau instruction [GKK+08]. Ce simulateur intègre huit tuiles connectées par un réseau en Torus 3D, chaque tuile incluant un modèle SystemC du DNP avec des annotations issues du modèle VHDL.

6.2.1 Communications intra-processeur intra-tuile

Nous commençons par une estimation de performance (voir Table 6.2) d'un canal déployé sur le chemin de communication suivant :

$WritePaths_T := RDM \cdot armbus \cdot RDM$

$ReadPaths_T := RDM \cdot armbus \cdot RDM$

La mémoire partagée est donc la mémoire locale à l'ARM (RDM pour RISC Data Memory), la zone d'échanges de données est allouée dynamiquement par le pilote. Physiquement, le medium (et donc la zone *heap*) se trouve en mémoire externe, nous ne respectons donc pas a priori le chemin matériel. Cependant, il est intéressant de noter que nous faisons les mesures également avec les caches données et instructions activés, en utilisant le protocole *write-through*, ce qui signifie qu'une écriture en mémoire est toujours propagée à l'adresse physique. Dans ce dernier cas, les données sont effectivement dans une mémoire du sous-système ARM au moment de l'utilisation du medium de communication, mais pas exactement celle escomptée.

Nous pouvons remarquer la grande pénalité introduite par les deux protocoles pour les tailles de messages faibles. Pour les tailles plus élevées, nous convergeons assez rapidement vers un plancher incompressible pour chaque protocole. Nous pouvons bien entendu noter l'importance des caches, qui accélèrent tout le programme et l'accès aux données. Ne sortant pas du sous-système ARM, il n'y a de plus pas de problème de cohérence. En revanche, ces communications introduisent des changements de contexte entre *fil1* et *fil2* qui font partie des pénalités incompressibles évoquées. Autre chose à noter, le protocole Rendez-vous est plus efficace que le protocole FIFO logicielle à partir d'une taille de 64 mots, ce avec les caches activés ou non. Il faut cependant noter que l'expérimentation choisie inhibe quelque peu l'avantage principal de la FIFO qui est de proposer des écritures non bloquantes. Nous considérons ici des échanges de messages

a/FIFO logicielle ARM@200MHz Bus@100Mhz		
Taille	Cycles/mot (Caches activés)	Cycles/mot (Caches désactivés)
1	737	2978
2	342	1529
4	168	711
8	87	381
16	47	219
32	28	136
64	18	96
128	13	73
256	11	64
512	10	58
1024	10	56
2048	10	54
4096	10	53
8192	10	53

b/ Rendez-vous ARM@200MHz Bus@100Mhz		
Taille (mots)	Cycles/mot (Caches activés)	Cycles/mot (Caches désactivés)
1	946	3986
2	430	1798
4	216	908
8	110	467
16	57	247
32	31	137
64	17	82
128	11	54
256	8	41
512	6	34
1024	5	30
2048	5	29
4096	5	28
8192	5	28

TABLE 6.2 – Estimation de performances d'une communication ARM-ARM. a/ Protocole FIFO. b/ Protocole Rendez-vous

entre deux fils d'exécution (c'est à dire un envoi et une réception d'un côté, une réception et un envoi de l'autre), l'écriture non bloquante en FIFO donne lieu immédiatement à une attente en lecture.

6.2.2 Communications inter-processeurs intra-tuile

Nous montrons en Table 6.3 une estimation de performances d'un canal déployé sur le chemin de communication suivant :

$$WritePaths_T := RDM \cdot armbus \cdot pontARM \cdot busamba \cdot RAM$$

$$ReadPaths_T := DDM \cdot magicbus \cdot DMA \cdot busamba \cdot RAM$$

La mémoire partagée se situe dans la RAM du Diopsis 940, la mémoire cible est la mémoire interne au sous-système DSP (DDM pour DSP Data Memory). Les mêmes mesures que précédemment ont été réalisées.

La Table 6.3 montre des résultats performants pour les communications ARM-DSP. Pour les messages de plus petite taille, les résultats sont meilleurs que pour le cas intra-processeur, car les tâches mises en jeu sont situées sur deux processeurs différents, sans besoin de changement de contexte. Pour les tailles plus grandes, les résultats sont moins performants car il faut faire communiquer deux sous-systèmes différents (avec un DMA à programmer côté DSP, un pont à franchir). De plus, il n'y a pas de cohérence de cache entre l'ARM et le DSP, le cache doit donc être vidé pour chaque opération de lecture côté ARM (seulement en lecture car nous utilisons un protocole *write-through*).

Le protocole Rendez-vous est dans ce cas plus performant que la FIFO logicielle pour toute taille de données, mais il faut nuancer ce constat avec le mode d'expérimentation qui favorise le Rendez-vous en imposant des synchronisations entre les deux fils concernés, comme expliqué précédemment.

a/FIFO logicielle ARM@200MHz : DSP@100Mhz Bus@100MHz		
Taille	Cycles/mot (Caches activés)	Cycles/mot (Caches désactivés)
1	1128	2144
2	740	804
4	494	398
8	242	214
16	87	119
32	43	74
64	18	51
128	12	38
256	10	33
512	9	30
1024	8	29
2048	8	29
4096	8	28
8192	8	28

b/ Rendez-vous ARM@200MHz : DSP@100Mhz Bus@100MHz		
Taille (mots)	Cycles/mot (Caches activés)	Cycles/mot (Caches désactivés)
1	680	1005
2	335	490
4	172	254
8	83	137
16	44	79
32	25	50
64	16	35
128	10	29
256	8	25
512	7	23
1024	6	22
2048	6	22
4096	5	21
8192	5	21

TABLE 6.3 – Estimation de performances d'une communication ARM-DSP. a/ Protocole FIFO. b/ Protocole Rendez-vous

6.2.3 Ethernet : protocoles RAW, UDP et TCP

Nous montrons en Table 6.5 une estimation de performances d'un canal reliant deux tuiles T_1 et T_2 déployé sur le chemin de communication suivant :

$$WritePaths_{T_1} := RDM_{T_1} \cdot armbus_{T_1} \cdot pontARM_{T_1} \cdot busamba_{T_1} \cdot MAC_{T_1} \cdot ETH$$

$$ReadPaths_{T_2} := tRDM_{T_2} \cdot armbus_{T_2} \cdot pontARM_{T_2} \cdot busamba_{T_2} \cdot MAC_{T_2} \cdot ETH$$

avec les protocoles TCP, UDP et Raw et

$$WritePaths_{T_1} := RDM_{T_1} \cdot armbus_{T_1} \cdot pontARM_{T_1} \cdot busamba_{T_1} \cdot MAC_{T_1} \cdot ETH$$

$$ReadPaths_{T_2} := DDM_{T_2} \cdot magicbus_{T_2} \cdot DMA_{T_2} \cdot busamba_{T_2} \cdot MAC_{T_2} \cdot ETH$$

avec uniquement le protocole Raw.

Le Table 6.4 montre des résultats bien entendu moins performants que pour le cas intra-tuile. Il ne faut pas oublier cependant que ce canal nécessite plusieurs étapes coûteuses : la formation de la trame Ethernet, la programmation du périphérique MAC, la sortie hors de la puce pour attaquer le bloc PHY qui fait l'interface avec la prise RJ45 (croisée) faisant le pont avec l'autre tuile. Nous notons que la pénalité introduite par le protocole TCP est assez importante par rapport aux protocoles UDP et RAW, ce qui s'explique aisément par le fait que le protocole TCP implique plus de copies intermédiaires que ceux-ci pour vérifier et éventuellement trier l'ordre des séquences reçue. La pénalité introduite par le protocole UDP est elle négligeable par rapport au mode RAW, la formation de l'en-tête UDP et le calcul du checksum côté émission et réception semble à première vue négligeable par rapport aux autres opérations concernant la sortie hors de la puce. Notons également les mauvaises performances de ce mécanisme de communication sur le DSP dûes pour une grande part à une limitation du Diopsis 940 qui ne permet pas l'accès direct du DMA intégré au MAC à la mémoire donnée du DSP. Les données préparées par le pilote Ethernet sur ce processeur sont donc copiées dans la mémoire RAM, ce qui représente une copie intermédiaire qui pénalise les performances

a/Socket TCP ARM@200MHz : MAC@100Mhz Bus@100MHz		
Taille	Cycles/mot (Caches activés)	Cycles/mot (Caches désactivés)
1	9639	12906
2	3426	6507
4	1743	3322
8	936	1737
16	489	990
32	275	579
64	170	387
128	118	288
256	91	239
512	89	258
1024	83	226
2048	82	225
4096	82	225
8192	82	225

b/ Protocole UDP ARM@200MHz : MAC@100Mhz Bus@100MHz		
Taille (mots)	Cycles/mot (Caches activés)	Cycles/mot (Caches désactivés)
1	972	18108
2	807	9012
4	397	4505
8	215	2250
16	125	1125
32	80	563
64	57	281
128	46	141
256	40	70
512	40	68
1024	38	51
2048	38	50
4096	39	50
8192	39	50

C/Socket RAW Arm-Arm ARM@200MHz : MAC@100Mhz Bus@100MHz		
Taille	Cycles/mot (Caches activés)	Cycles/mot (Caches désactivés)
1	1942	18102
2	702	9030
4	344	4506
8	176	2253
16	99	1127
32	67	563
64	51	282
128	42	141
256	38	70
512	38	68
1024	37	51
2048	37	50
4096	38	48
8192	38	47

d/Socket RAW ARM-DSP ARM@200MHz : DSP@100Mhz MAC@100MHz : Bus@100MHz		
Taille	Cycles/mot (Caches activés)	Cycles/mot (Caches désactivés)
1	3528	32421
2	1834	18943
4	890	9231
8	578	4530
16	322	2729
32	178	1385
64	82	723
128	65	381
256	61	329
512	60	291
1024	60	287
2048	60	286
4096	59	286
8192	59	286

TABLE 6.4 – Estimation de performances d'une communication utilisant le réseau Ethernet. a/ ARM-ARM TCP b/ ARM-ARM UDP c/ ARM-ARM Raw d/ ARM-DSP Raw

mais permet la réception de grandes tailles de données.

6.2.4 RDMA : protocoles Eager et Rendez-vous

Nous montrons en Table 6.5 une estimation de performances d'un canal reliant deux tuiles T_1 et T_2 déployé sur les chemins de communication suivant :

$$WritePaths_{T_1} := RDM_{T_1} \cdot armbus_{T_1} \cdot pontARM_{T_1} \cdot busamba_{T_1} \cdot DNP_{T_1} \cdot TORUS$$

$$ReadPaths_{T_2} := RDM_{T_2} \cdot armbus_{T_2} \cdot pontARM_{T_2} \cdot busamba_{T_2} \cdot DNP_{T_2} \cdot TORUS$$

et

$$WritePaths_{T_1} := RDM_{T_1} \cdot armbus_{T_1} \cdot pontARM_{T_1} \cdot busamba_{T_1} \cdot DNP_{T_1} \cdot TORUS$$

$$ReadPaths_{T_2} := DDM_{T_2} \cdot magicbus_{T_2} \cdot DMA_{T_2} \cdot busamba_{T_2} \cdot DNP_{T_2} \cdot TORUS$$

Comme nous l'avons mentionné auparavant, nous avons porté les deux protocoles Eager et Rendez-vous dans un et un seul pilote pour chaque processeur, nous présentons les performances séparées de chaque protocole.

a/Canal RDMA ARM-ARM ARM@200MHz : DNP@100Mhz Bus@100MHz				
	Caches activés		Caches désactivés	
Mots	Eager	Rendez-vous	Eager	Rendez-vous
1	1635	4882	1666	4507
2	880	2533	914	2803
4	446	1198	479	1170
8	222	677	245	681
16	175	358	118	330
32	75	164	83	145
64	42	75	47	86
128	26	48	30	47
256	23	32	28	32
512	17	20	20	19
1024	14	13	17	15
2048	12	11	14	12
4096	11	10	13	11
8192	10	10	12	10
16384	10	9	12	10

b/Canal RDMA ARM-DSP ARM@200MHz : DSP@100Mhz DNP@100MHz : Bus@100MHz				
	Caches désactivés		Caches activés	
Mots	Eager	Rendez-vous	Eager	Rendez-vous
1	9114.4	30122	5180	29896
2	5234	14015	2671	15207
4	2633	7090	1390	7173
8	1265	3480	685	3869
16	597	1786	359	1979
32	303	878	177	995
64	120	448	84	403
128	60	225	49	186
256	-	123	-	117
512	-	75	-	64
1024	-	40	-	38
2048	-	25	-	22
4096	-	16	-	17
8192	-	12	-	12
16384	-	9	-	10

TABLE 6.5 – Estimation de performances pour deux protocoles RDMA en Cycles de bus par mot. a/ Canal ARM-ARM b/ Canal ARM-DSP

Pour le cas d'un transfert inter-tuiles ARM-ARM, nous noterons la forte pénalité introduite par le protocole Rendez-vous par rapport au protocole Eager pour les faibles tailles de données. Cependant, pour une taille de 512 mots, la tendance s'inverse dans tous les cas et les copies intermédiaires requises par le protocole Eager prennent le pas sur les paquets supplémentaires requis par le protocole Rendez-vous. Les résultats sont globalement meilleurs que pour le cas des sockets, nous pouvons attribuer ces meilleures performances à une interface logiciel/matériel avec le DNP plus performante, le prix à payer étant une gestion de l'interface côté pilote plus lourde. Ces résultats expliquent que nous ayons fait le choix d'intégrer les deux protocoles dans un seul pilote, et à fixer dans la configuration de chaque canal une taille de données à transférer limite qui permet de choisir le protocole. L'effet bénéfique des caches (activés dans les deux tuiles mises en jeu) sont moins spectaculaires que pour le cas intra-tuile, ils apportent même une contribution négative dans certains cas (à relativiser car nous vidons une grande portion du cache du côté de l'ARM lors de la lecture, plus que nécessaire). Rendre le système de caches cohérent côté DNP est un travail indispensable pour améliorer encore les performances, qui sont cependant satisfaisantes pour les tailles de données élevées.

Pour le cas d'un transfert inter-tuiles ARM-mAgicV, nous retrouvons un phénomène que nous avons expliqué en Chapitre V, à savoir le problème de la gestion du Eager pour les tailles de données élevées, car nous utilisons une zone de mémoire dynamique limitée à 1 Kmot (sur les 16 Kmots disponibles) pour stocker temporairement les paquets Eager, ce qui explique qu'au-delà d'une taille de 256 mots, nous commençons à avoir des problèmes (il suffit de trois messages reçus pour obtenir un débordement, la mémoire dynamique étant utilisée par ailleurs dans le code du DSP). L'interface matériel/logiciel du DNP introduit là encore une forte pénalité, mais qui est compensée à mesure que la taille des transferts augmente. Nous noterons que le cache activé du côté de l'ARM amène plus souvent une contribution négative, mais là encore nous ne nous sommes pas attachés à vider uniquement les lignes du cache nécessaires lors d'une lecture côté ARM.

6.3 Applications

Nous nous intéressons à présent à trois applications écrites en réseau de processus par des partenaires du projet européen SHAPES. Nous avons choisi le DOL (voir Annexe 1) comme formalisme pour le modèle de haut niveau. Ceci signifie que le programmeur a eu à écrire le modèle d'application au format XML et ses tâches applicatives avec une interface de programmation imposée par DOL (pour la COM et la gestion des fils). Le fichier d'architecture lui était fourni, le programmeur a eu alors le choix d'écrire les fichiers de déploiement à la main ou de les générer automatiquement à l'aide des outils fournis avec le modèle.

Ces applications ont des caractéristiques différentes, mais sont toutes constituées d'un cœur de calcul qui a été la plupart du temps déployé sur le DSP. Un support de la part des concepteur du flot a été nécessaire, mais les programmeurs d'application, non spécialistes de la plateforme matérielle multi-tuiles, ont tous réussi à faire fonctionner au moins quelques déploiements passés au travers de nos outils. Notons que nous avons fait le choix d'imposer les chemins matériels pour chaque type de canal (intra-processeur, inter-processeurs et inter tuiles), et avons prévenu les concepteurs que toutes les communications étaient en mode synchrone (protocoles Rendez-vous et RDMA Rendez-vous).

6.3.1 Lattice Quantum Chromo-Dynamics

La chromodynamique quantique (QCD) décrit la loi d'intéraction forte, c'est-à-dire la loi qui lie les quarks et les gluons pour former des hadrons. Une résolution analytique n'est pas possible, la plupart des résultats des chercheurs dans le domaine vient donc de simulations de la version discrète du QCD, le Lattice QCD. Dans ce modèle, les quatre dimensions du continuum espace-temps sont discrétisés sous forme d'une *lattice* (treillage en français), les quarks étant symbolisés par les sites de la lattice et les gluons étant symbolisés par les liens entre ces sites.

Le cœur de calcul du LQCD, le plus coûteux en termes de calculs, est réalisé par les opérateurs de Wilson-Dirac [IB07] mettant en jeu des multiplications entre des matrices éparses, les matrices du champ de gauge (3x3 variables complexes) et les matrices du champ de quark (4 vecteurs de trois variables complexes).

Dans l'implantation proposée par l'INFN [BST+06] (Institut National de Physique Nucléaire de Rome) sous forme de réseau de processus, chaque tuile représente une sous-lattice, qui effectue ses calculs en fonction de l'état courant et des contributions des sous-lattices voisines sur le Torus, et exporte les valeurs des champs de quarks et de gauge aux frontières à chacun de ses voisins sur le Torus.

Le réseau de processus est composé de quatre tâches :

- *Data_Fetcher* qui récupère les données des tuiles voisines pour sa tuile.
- *Data_Calc* qui lit les données de la tâche *Data_Fatcher*, fait ses calculs à base d'opérateurs Wilson-Dirac et écrit ses valeurs vers le *Result_Collector*.
- *Result_Collector* reçoit les données de *Data_Calc* et les écrit dans la mémoire partagée de la tuile
- *Intertile_Comm* reçoit un signal de *Result_Collector* et s'occupe des échanges de données aux frontières de la sous-lattice.

La Table 6.6 résume les différentes tailles en mémoire programme pour chacun des éléments du réseau

de processus sur les deux processeurs de la tuile.

LQCD process	ARM	DSP
Data Fetcher	0,85	-
Data Calc	6,48	0,82
Result Collector	0.09	-
Intertile Comm	9,4	-

TABLE 6.6 – Taille en Kmots des éléments du LQCD (section programme)

Nous pouvons constater qu'une version très optimisée du *Data_Calc* a été portée sur le DSP en notant la faible taille de sa section programme. Plusieurs essais de déploiement ont été passées par le flots pour des version une tuile, deux tuiles et huit tuiles avec la tâche de calcul déployée tantôt sur l'ARM tantôt sur le mAgic. Tous les déploiements simulent huit sous-lattices réparties sur les tuiles utilisées, La Table 6.7 résume le nombre total de canaux intra-processeur, inter-processeurs et inter-tuiles utilisés par l'application :

Déploiement	Canaux intra-proc	Canaux inter-procs	Canaux inter-tuiles
1 tuile, ARM	96	0	0
1 tuile, ARM+DSP	80	16	0
2 tuiles, ARM	72	0	24
2 tuiles, ARM+DSP	56	16	24
8 tuiles, ARM	34	0	62
8 tuiles, ARM+DSP	18	16	62

TABLE 6.7 – Nombre total de canaux intra- et inter-tuiles pour le LQCD

La première remarque que nous pouvons faire est le nombre de canaux de communications mis en jeu pour une version avec seulement huit tuiles. Toutes les configurations des canaux n'auraient pu être faites à la main. Autre point intéressant à noter, le concepteur de l'application a fait le choix de ne pas utiliser de canaux inter-tuiles impliquant le DSP, ce qui est un choix raisonnable au vu de la taille du pilote RDMA et de ses performances sur cette unité de calcul. Nous ne détaillons pas les quantités de données échangées, nous pouvons dire que les taille de données échangées vont de 18 mots (taille de la matrice SU3 du champ de quark) à 3648 mots (taille récupérée par la tâche de calcul incluant les champs de quark et de gauge voisins).

Nous fournissons ici quelques données importantes à retenir, fournies par l'équipe de programmation de l'INFN, concernant le déploiement huit tuiles ARM+DSP : la tuile en fonctionnement opère à 2800 MFlops/W (contre 722 MFlops/W pour le QPACE, premier au classement Green500 en Novembre 2009 [cFC07]), avec une puissance de calcul mesurée à 1,32 GFlops par tuile (soit 53% de la puissance pic du mAgicV qui est de 2,5 GFlops).

6.3.2 Wave Field Synthesis

L'application Wave Field Synthesis (WFS) est basée sur la théorie de Huygens : tous les points sur un front d'onde servent de source ponctuelle secondaire à des fronts d'onde sphériques secondaires. Ce principe est utilisé en acoustique par l'utilisation de rangées de petits haut-parleurs légèrement espacés. On injecte dans chaque haut-parleur un signal calculé par des algorithmes basés sur les intégrales de Kirchhoff-Heimholtz et la représentation de l'onde de Rayleigh, ce qui donne en tout point de la salle d'écoute un sentiment d'immersion.

Sa décomposition en réseaux de processus est décrite dans [SBF+07]. Le cœur de calcul est composé de convolutions entre deux sources distinctes (processus *Convolution*) sommées par le processus *Blending* pour obtenir le signal en entrée de chaque haut-parleur. Le processus *Control* est connecté à chacun de ces modules pour éventuellement commander un effet sonore ou un mixage particulier à partir d'une console de mixage externe, tandis que le processus *Impulse Response Database* génère un filtre passe-bas dépendant de la position de chaque source afin de compenser les effets sonores indésirables produits par le WFS et les caractéristiques acoustiques de la salle d'écoute.

Taille des processus du WFS en Kmots		
Processus	**ARM**	**DSP**
ControlModule	0.9	-
Impulse Response Database	0.2	-
Convolution	0.9	1.4
Blending	0.2	0.7
Input	0.2	-
Output	0.3	-

TABLE 6.8 – Taille en mémoire programme des processus du WFS

Plusieurs versions de l'application ont été proposées, avec un nombre de sources variant de 1 à 4 et 4 haut-parleurs. La variation du nombre de sources a une influence sur le nombre de processus *Convolution* (nombre de sources fois nombre de haut-parleurs) et de processus *Blending* (autant de processus que de haut-parleurs). Si l'application a fonctionné dans toutes les configurations avec des déploiements ne faisant intervenir que l'ARM, certains déploiements faisant intervenir le DSP pour les deux opérations de calcul ont atteint les limites (en mémoire programme et/ou données) des DSPs. En effet, la plupart des configurations à quatre sources faisant intervenir 16 processus *Convolution* et 4 processus *Blending* à répartir sur les DSP, les déploiement sur une ou deux tuiles ont eu tendance à atteindre les limites physiques des DSPs en raison du nombre de canaux, de processus et de pilotes déployés sur ceux-ci, Certains déploiements utilisant des communication inter-tuiles entre DSP (très gourmands en termes d'espace mémoire) ont également été confrontés au même problème. Le volume de données échangées dans cette application varie de 64 mots à 4096 mots.

6.3.3 Application ultrason

L'application Ultrason à usage médical que nous présentons ici a été portée en Réseaux de Processus par la société Esaote [Esa]. Le but de cette application est de transformer le signal radiofréquences récolté par une rangée de transducteurs en images (images qui sont obtenues par des traitements ultérieurs non évoqués dans cette partie). Les signaux radiofréquences se présentent sous forme de vecteurs chacun composés de 1020 échantillons, chacun de ces vecteurs représentant une ligne de l'image à reconstituer. Ces n vecteurs sont traités par une moyenne sur l'ensemble des échantillons et une transformée de Hilbert pour récupérer l'enveloppe du signal. L'application consiste en plusieurs processus : *Read-File-To-Matrix* qui lit un fichier contenant les échantillons RF de l'image à décoder et l'envoie à l'un des cœur de calcul consistant en un processus *Mean-Hilbert* qui récupère l'enveloppe du signal. Le processus *Write-Matrix-To-File* récupère chaque vecteur traité par *Mean-Hilbert* et le stocke dans un fichier pour les traitements de mise en forme suivants.

Le réseau de processus est plus simple que pour les applications précédentes, car il y a une indépendance totale entre chaque vecteur à traiter. En conséquence le nombre de canaux de communication est égal à deux

fois le nombre de processus de calcul. Plusieurs déploiements une tuile et 8 tuiles ont été testés, avec les processus de calcul sur l'ARM ou sur le DSP. Nous retranscrivons ici quelques mesures effectuées pour un nombre de vecteurs croissant de 1 à 8 (un processus de calcul alloué à chaque vecteur), la version à 8 vecteurs étant déployée sur huit tuiles.

Une tuile	
1 vecteur	1,59ms
2 vecteur	2,74ms
3 vecteurs	5,23ms
4 vecteurs	6,74ms
Huit tuiles	
8 vecteurs	18,25ms

TABLE 6.9 – Performances de l'application ultrason

Nous pouvons constater que la version huit tuiles présente de moins bons résultats que la version une tuile, le coût des communication inter-tuiles reste encore trop élevé par rapport au temps de calcul pour des vecteurs de 1024 échantillons complexes (donc 2048 mots). Nous voyons ici l'un des effets néfastes des communication inter-tuiles qui introduisent une grande pénalité, cependant comme nous l'avons vu lors de l'estimation de performances, l'échange de 1024 mots est bien plus pénalisant avec le protocole RDMA rendez-vous que l'échange de quantité de données plus élevées, une augmentation du nombre d'échantillons pourrait permettre de recouvrir le temps de communications avec le temps de calcul mais ceci nécessiterait une phase d'optimisation propre à l'application.

Conclusion générale

Nous avons présenté dans ce manuscrit un flot de conception de logiciel embarqué spécialement conçu pour les plateformes multi-tuiles hétérogènes. Ce flot se base sur une représentation à un haut niveau d'abstraction de l'architecture, de l'application et du déploiement des éléments logiciels sur les éléments de l'architecture pour générer un binaire par unité de calcul de la plateforme reflétant le comportement de l'application décrite en entrée. En raison de la grande quantité et de l'hétérogénéité des chemins matériels introduits par les plateformes multi-tuiles visées, nous nous sommes attachés tout au long du manuscrit à décrire le processus de *synthèse* des communications entre processus du modèle de haut niveau de l'application pour arriver à produire des binaires corrects.

Nous avons soulevé un certain nombre de problèmes concernant le mécanisme que nous souhaitions mettre en œuvre dans le flot, les contributions de cette thèse répondent à ces problématiques qui concernent l'interface de communication avec le programmeur d'application et la manière dont le flot arrive à produire des binaires fonctionnels. Ceux-ci doivent intégrer les informations permettant de faire communiquer les différentes tâches déployées sur les nombreux processeurs de la plateforme. Notre flot est constitué d'une partie *front-end*, composée d'un module de génération et d'un module de sélection de composants pour former les couches logicielles, et d'une partie *back-end* qui permet grâce à des outils de compilation d'obtenir les binaires finaux.

Nous avons commencé par décrire les informations vitales qui doivent être présentes dans les trois modèles de haut niveau, informations qui permettent au flot de puiser des composants logiciels de communication dans une bibliothèque existante en fonction des critères du modèle. Nous avons en particulier introduit formellement les chemins de communication matériels dans le modèle d'architecture, chaque canal du modèle d'application en réseaux de processus étant déployé sur un chemin matériel de communication. Ces chemins reposent sur un medium de communication (une mémoire partagée pour les communications intra-tuile, un réseau de communication commuté par paquets pour les communications inter-tuiles). Nous avons ensuite décrit la structure de nos composants de communication, qui se présentent sous forme de pilotes de communication permettant aux tâches de communiquer entre elles par passages de messages. La structure interne de ces pilotes et plusieurs protocoles de communication pertinents ont été détaillés. Comme le système d'exploitation, les pilotes reposent exclusivement sur une couche d'abstraction du matériel (le HAL) qui permet en particulier un accès au proche environnement dans la tuile.

Nous nous sommes rendus compte que le respect rigoureux des chemins matériels, qui spécifient tous les blocs matériels à traverser pour atteindre le medium de communication, n'était pas possible en raison de la structure en couches du logiciel qui impose un chemin (optimal) pour atteindre les mémoires et les périphériques de la tuile. Une autre raison à cette infraction vient de la mémoire source ou cible du mécanisme par passage de messages dont l'adresse n'est pas décidée directement par le flot mais par le

programmeur d'application d'une part et par le mécanisme d'édition de liens d'autre part. Nous avons donné les informations vitales à ajouter au modèle pour que le module de sélection du flot puisse associer un pilote à chaque canal de communication du modèle de l'application.

Ceci nous a mené à nous intéresser à la partie *back-end* du flot composé d'outils de compilation. Nous avons pu mettre en évidence quelques difficultés introduites par l'hétérogénéité de la plateforme ciblée, aussi bien pour l'écriture des composants logiciels de communication que pour les conditions d'utilisation de ces composants. Nous avons proposé une implantation du flot sous forme d'outils dont nous nous sommes servis pour mener à bien les expérimentations. Ces expérimentations se sont attachées à montrer l'empreinte mémoire des différents composants de communication présentés et une estimation de performance pour chacun d'eux, en utilisant une plateforme multi-tuiles basée sur un Diopsis 940 d'Atmel. Trois applications (dont une appartenant au monde du calcul de haute performance) ont été décrites en réseaux de processus et plusieurs déploiements ont été passés à travers notre flot, celui-ci produisant un résultat correct dans chaque cas. Pour ces trois applications, nous avons fait le choix d'imposer les mécanismes de communication les plus efficaces pour la plateforme considérée.

Perspectives

L'objectif du flot de conception proposé dans cette thèse est avant tout de donner au programmeur de l'application une vue simple de la plateforme matérielle, mais aussi de lui octroyer un fort niveau de contrôle sur la mise en œuvre de son programme déployé sur le matériel. Le fait que nous ciblions des systèmes multi-tuiles hétérogènes nous a poussé à trouver un moyen de présenter les communications de manière intuitive mais efficace au programmeur. C'est dans cette optique que nous avons introduit le chemin de communication matériel dans la description de haut niveau de l'architecture. La définition du chemin matériel adoptée reflète la volonté de donner une certaine maîtrise au programmeur de l'application, en lui permettant de spécifier les blocs matériels traversés par les canaux logiciels de son programme représenté sous forme de réseaux de processus.

Nous avons montré un certain nombre de difficultés pour préserver l'équilibre simplicité/contrôle à haut niveau, à mesure que nous sommes rentrés dans les détails de l'implantation du flot pour arriver à un outil fonctionnel. Tout d'abord, la définition du chemin matériel comme la succession des blocs traversés lors d'un transfert n'est pas suffisante pour arriver à une implantation des communications, d'autres informations doivent être fournies comme le type de réseau et le protocole de communication à utiliser. Nous avons longtemps hésité entre une approche à la MPI où le choix du protocole est explicitement présenté dans l'interface de programmation, et une approche qui consiste à proposer un mécanisme de communication efficace pour chaque chemin matériel. C'est cette dernière approche que nous avons adoptée vers la fin du manuscrit, la simplicité de l'interface offerte au programmeur étant compensée par une phase d'optimisation de l'application plus difficile à metttre en œuvre (nous pensons en particulier au recouvrement du temps de calcul par les communications, rendu plus aisé dans MPI par les différents modes de communication à disposition). La deuxième difficulté est plus structurelle et inhérente aux choix faits dans le flot. En particulier, la structure en couches du logiciel, avec des primitives fixées à l'avance pour chaque plateforme ne donnent pas une liberté totale sur les blocs mis en jeu lors d'un transfert de données. Ceci est une conséquence directe du choix de synthèse des communications, car nous puisons des composants logiciels fixés dont les accès au matériel sont figés et optimaux (grâce à la couche d'abstraction de matériel la plus basse). Enfin, le processus de compilation croisée exige un certain nombre de paramètres à fixer comme le

déploiement en mémoire des éléments logiciels qu'il est quelques fois possibles de contrôler ou non, ceci ne pouvant être connu à l'avance en raison de l'hétérogénéité des unités de calcul visées. Il est très difficile de présenter de manière intuitive au programmeur ces informations, aussi nous avons choisi de les lui masquer. C'est une nouvelle fois la phase d'optimisation de la couche applicative qui s'en trouve compliquée. Une connaissance poussée des outils du *back-end* et du fonctionnement du flot est nécessaire pour cette phase.

Nous avons proposé dans les travaux futurs des idées pour fortement améliorer le contrôle à haut niveau du comportement de l'application et en particulier des communications. Nous avons montré qu'une simple synthèse ne suffisait pas, mais qu'une phase de génération de logiciel est nécessaire. Nous avons également proposé une possible continuation de ce travail pour élargir le périmètre de notre flot à tout type de système embarqué, en proposant un modèle formel intermédiaire qui permet de vérifier le bon fonctionnement du programme autrement que par simple exécution, ce qui rendrait cette étude également pertinente pour des systèmes massivement parallèles homogènes.

Annexe A

Illustration de DOL et du point d'entrée de l'application généré

Nous montrons dans cette annexe quelques échantillons de fichiers manipulés ou gérés par notre flot de génération. Nous nous appuyons sur un formalisme DOL, dans lequel nous avons ajouté la notion de chemin matériel de communication (*hwpath*), et sur la bibliothèque de composants logiciels APES. Pour des raisons de simplicité de présentation, nous avons choisi une application décodeur MJPEG décomposée en trois tâches : *Fetch* qui se charge de décoder les sections du fichier MJPEG stocké en mémoire (incluant la lecture des arbres de Huffman, décompression et quantification), *IDCT* (transformée cosinus inverse) qui permet un passage du mode fréquentiel au mode temporel et *Dispatch* qui reconstitue les lignes de l'image pour les afficher (envoi sur le port série de la carte). Dans l'exemple présenté, les tâches *Fetch* et *Dispatch* sont déployées sur l'ARM du Diopsis 940, l'*IDCT* est déployée sur le mAgic. Trois canaux de communication sont définis au niveau du modèle de l'application : un entre *Fetch* et *IDCT*, un entre *IDCT* et *Dispatch* et un entre *Fetch* et *Dispatch* (pour transférer les dimensions de l'image).

A.1 Modèle DOL

A.1.1 Fichier d'application

Le fichier d'application consiste en un fichier XML qui commence par définir tous les éléments du réseau de processus et les connecte entre eux. L'élément d'interconnexion est le *port*, qui permet également dans le fichier d'implantation des tâches de spécifier sur quel canal doit se faire la communication.

```
<!-- Définition des processus -->
  <process name="fetch">
          <port name="0" type="output"/>
          <port name="1" type="output"/>
          <source location="fetch.c" type="c"/>
  </process>

      <process name="idct" basename="idct" range="1">
              <port name="0" type="input"/>
```

```
            <port name="1" type="output"/>
            <source location="idct.c" type="c"/>
      </process>
 </iterator>

  <process name="dispatch" basename="dispatch">
         <port name="0" type="input"/>
         <port name="1" type="input"/>
         <source location="dispatch.c" type="c"/>
  </process>

<!-- Canaux logiciels -->

        <sw_channel name="fetch_idct_channel">
                <port name="0" type="input"/>
                <port name="1" type="output"/>
        </sw_channel>

        <sw_channel name="idct_dispatch_channel">
                <port name="0" type="input"/>
                <port name="1" type="output"/>
        </sw_channel>

        <sw_channel name="fetch_dispatch_channel">
                <port name="0" type="input"/>
                <port name="1" type="output"/>
        </sw_channel>

<!--Connexions>
     <connection name="fetch_out_cnx">
                <origin name="fetch">
                    <port name="0">
                </origin>
                <target name="fetch_idct_channel">
                        <port name="0"/>
                </target>
        </connection>

            <connection name="idct_in_cnx">
                <origin name="fetch_idct_channel">
                        <port name="1"/>
                </origin>
                <target name="idct">
                        <port name="0"/>
                </target>
        </connection>
```

A.1.2 Fichier d'architecture

```
 <!-- sous-système arm (tile_0)  -->
  <processor name="tile_0.arm" type="RISC"></processor>
  <memory name="tile_0.rdm" type="RAM"></memory>

  <hw_channel name="tile_0.armbus" type="BUS">
    <configuration name="frequency" value="100000000"/>
    <configuration name="bytespercycle" value="1"/>
  </hw_channel>

  <!-- sous-système magic (tile_0) -->
  <processor name="tile_0.magic" type="DSP"></processor>
  <memory name="tile_0.ddm" type="RAM"></memory>

  <hw_channel name="tile_0.magicbus" type="BUS">
    <configuration name="frequency" value="100000000"/>
    <configuration name="bytespercycle" value="1"/>
```

```
 </hw_channel>

 <hw_channel name="tile_0.dma" type="DMA">
   <configuration name="frequency" value="100000000"/>
   <configuration name="bytespercycle" value="1"/>
 </hw_channel>

 <!-- matrice de bus AHB (tuile_0)  -->
 <hw_channel name="tile_0.ahb0" type="BUS">
   <configuration name="frequency" value="100000000"/>
   <configuration name="bytespercycle" value="1"/>
 </hw_channel>

 <hw_channel name="tile_0.ahb1" type="BUS">
   <configuration name="frequency" value="100000000"/>
   <configuration name="bytespercycle" value="1"/>
 </hw_channel>

<!-- quelques chemins matériels -->
 <writepath name="tile_0.rdmtordm">
   <processor name="tile_0.arm"/>
   <txbuf name="tile_0.rdm"/>
   <hw_channel name="tile_0.armbus"/>
   <chbuf name="tile_0.rdm"/>
 </writepath>

 <readpath name="tile_0.ddmfromddm">
   <processor name="tile_0.magic"/>
   <chbuf name="tile_0.ddm"/>
   <hw_channel name="tile_0.magicbus"/>
   <rxbuf name="tile_0.ddm"/>
 </readpath>

 <readpath name="tile_0.ddmfromdxm">
   <processor name="tile_0.magic"/>
   <rxbuf name="tile_0.ddm"/>
   <hw_channel name="tile_0.magicbus"/>
   <hw_channel name="tile_0.dma"/>
   <hw_channel name="tile_0.ahb1"/>
   <chbuf name="tile_0.dxm"/>
 </readpath>
```

A.1.3 Déploiement

```
 <!-- déploiement des processus sur les processeurs -->
        <binding name="fetch_binding" xsi:type="computation">
                <process name="fetch"/>
                <processor name="tile_0.arm"/>
        </binding>

        <binding name="dispatch_binding" xsi:type="computation">
                <process name="dispatch"/>
                <processor name="tile_0.arm"/>
        </binding>

        <binding name="idct_binding0" xsi:type="computation">
                <process name="idct_0"/>
                <processor name="tile_0.magic"/>
        </binding>

  <!-- déploiement des canaux logiques sur les chemins de communicaiton  -->
        <binding name="fetch_idct_binding0" xsi:type="communication">
         <sw_channel name="fetch_idct_channel_0"/>
             <writepath name="tile_0.rdmtodxm"/>
             <readpath name="tile_0.ddmfromdxm"/>
```

```
        </binding>

        <binding name="idct_dispatch_binding0" xsi:type="communication">
         <sw_channel name="idct_dispatch_channel_0"/>
            <writepath name="tile_0.ddmtodxm"/>
             <readpath name="tile_0.rdmfromdxm"/>
        </binding>

        <binding name="fetch_dispatch_binding" xsi:type="communication">
         <sw_channel name="fetch_dispatch_channel"/>
         <writepath name="tile_0.rdmtordm"/>
         <readpath name="tile_0.rdmfromrdm"/>
        </binding>
```

A.1.4 Implantation des tâches

Nous ne montrons qu'un extrait du fichier idct.c attaché au processus idct dans le modèle d'application, principalement pour souligner les prototypes des fonctions de lecture et d'écriture, *DOL_read* et *DOL_write*. Le DOL impose la définition d'une structure de type *DOLProcess* qui contient toutes les informations propres à l'instance du processus (ports, variables locales), et de deux fonctions *init* et *fire*, *init* étant appelé une seule fois et *fire* étant ensuite appelé en boucle.

```
int idct_fire(DOLProcess *p) {
    Idct_State * loc = (Idct_State *)p -> local;
    DOL_read(0, loc -> block_YCbCr, sizeof(loc->block_YCbCr), p);
    IDCT(p, loc -> block_YCbCr, loc -> Idct_YCbCr);
    DOL_write(1, loc -> Idct_YCbCr, sizeof(loc->Idct_YCbCr), p);
    return 0;
}
```

A.2 Fichier généré

Le fichier généré par notre flot représente le point d'entrée de l'application généré. Il initialise les structures nécessaires pour les communications, ouvre les fichier virtuels et lance les tâches.

```
int main (void) {
        /**********************
         * DOL port creations *
         **********************/

        DOLcreate_port (& dispatch, 2, 0);
        DOLcreate_port (& fetch, 0, 2);

        /***************************
         * DOL port instantiations *
         ***************************/

        DOLinit_port (& dispatch, IN, 0, "/devices/d940_rdv.0", 0);
        DOLinit_port (& dispatch, IN, 1, "/devices/rdv.0", 0);

        DOLinit_port (& fetch, OUT, 0, "/devices/d940_rdv.1", 0);
        DOLinit_port (& fetch, OUT, 1, "/devices/rdv.0", 0);

        /**********************
```

```
         * DOL tasks creations *
         ***********************/

        DOL_create (& dispatch, "dispatch", 0);
        DOL_create (& fetch, "fetch", 0);


        /*********************
         * END of generation *
         *********************/

        DOL_join (& dispatch);
        return 0;
}
```

Nous pouvons nous demander pourquoi le dernier paramètre de chaque fonction est toujours 0, en fait le DOL permet d'itérer les instances de processus et de ports, le dernier argument de chaque fonction est donc une liste qui permet de gérer les itérateurs introduits à haut niveau.

ANNEXE A. ILLUSTRATION DE DOL ET DU POINT D'ENTRÉE DE L'APPLICATION GÉNÉRÉ

Annexe B

En-têtes et types de données pour le protocole RDMA

Nous n'avons pas fait une présentation exhaustive du pilote RDMA qui est bien trop complexe pour être présenté dans un manuscrit de thèse. Nous donnons ici quelques structures de données manipulées par le pilote pour donner une idée de la complexité du périphérique piloté (le DNP) et faire comprendre pourquoi la taille en mémoire du pilote est très élevée.

B.1 En-têtes au niveau logiciel

Ces en-têtes caractérisent chaque paquet des protocoles RDMA Eager et Rendez-vous :

```
// Type de paquet
typedef enum rdma_pkt_types {
    RDMA_PKT_NONE,
    RDMA_PKT_RDV_INIT,
    RDMA_PKT_RDV_END,
    RDMA_PKT_EAGER
} rdma_pkt_types_t;

// Protocole Eager
typedef struct _pkt_rdv_eager {
    uint32_t buf_nwords;
} pkt_eager_t;

// Protocole Rendez-vous, paquet d'initialisation
typedef struct _pkt_rdv_init {
    uint32_t buf_address;
    uint32_t buf_nwords;
} pkt_rdv_init_t ;


// Protocole Rendez-vous, paquet de fin
typedef struct _pkt_rdv_end {
    uint32_t ack;
} pkt_rdv_end_t;

// En-tête pour tous les paquets
typedef struct _rdma_pkt {
```

```
    rdma_pkt_types_t pkt_type;
    uint32_t channel_id;
    uint32_t use_float;
    union {
        pkt_eager_t     eager_pkt;
        pkt_rdv_init_t  init_pkt;
        pkt_rdv_end_t   end_pkt;
    } pkt;
}  rdma_pkt_t;
```

Nous pouvons voir que l'en-tête pour tous les paquets fait 5 mots, nous pourrions bien entendu optimiser un peu ce choix en fondant des champs dans le même mot, mais cette définition a été utilisée pour la phase de développement et de deboguage.

B.2 Structure de la commande à envoyer au DNP

Comme nous l'avons évoqué, le pilote RDMA envoie des commandes de sept mots au DNP sur son interface esclave. Nous présentons ici la structure de données avec des champs de bits tel que définis dans l'ARM (le compilateur du mAgic ne fournissant pas cette fonctionnalité, les champs de la structure doivent être manipulés par des masques et des décalages).

```
typedef struct dnp_command_s {

        uint32_t pad1: 13;
        uint32_t dstx: 6;
        uint32_t dsty: 6;
        uint32_t dstz: 6;
        uint32_t dstdev: 1;
#define DNP_DEV_M0_IF 0
#define DNP_DEV_M1_IF 1

        uint32_t len : 32; // len (in bytes)

        uint32_t pad3: 13;
        uint32_t srcx: 6;
        uint32_t srcy: 6;
        uint32_t srcz: 6;
        uint32_t srcdev: 1;

        uint32_t pad4      : 15;
        uint32_t broadcast : 1;
        uint32_t comp      : 8;  // completion mode :IRQ, ...
        uint32_t cmd       : 8;

        uint32_t src_address; //src address


        uint32_t dst_address; //dst_address

        // if comp&(COMP_CQ|COMP_CQ_FRAG) != 0: this is the magic word written into the Completion Queue
        uint32_t magicw;
```

B.3 Structure de la complétion envoyée par le DNP

A la fin de chaque opération (acquittement, signalisation de paquet entrant), le DNP écrit huit mots de complétion dans un tampon circulaire que celui-ci gère et qui est scruté par la partie adéquate du pilote pour être informé de l'état des transferts. Ces mots doivent être interprétés par le pilote pour former des messages à mettre dans les boîtes aux lettres des canaux de communication.

```
typedef struct dnp_completion_event
{
    uint32_t cflags :   7;
    uint32_t dstdev :  1;
    uint32_t dstx :    6;
    uint32_t dsty :    6;
    uint32_t dstz :    6;
    uint32_t vchan :    4;
    uint32_t broadcast: 1;
    uint32_t routing:   1;


    uint32_t pflags : 3;
    uint32_t srcdev : 1;
    uint32_t srcx :   6;
    uint32_t srcy :   6;
    uint32_t srcz :   6;
    uint32_t len :  10;

//3ème mot pour l'usage du GET
    uint32_t rdma_srcx:   6;
    uint32_t rdma_srcy:   6;
    uint32_t rdma_srcz:   6;
    uint32_t rdma_len:   10;
    uint32_t tgtdev:  1;
    uint32_t cmd:     3;

    uint32_t src_vaddress : 32;

    uint32_t dst_vaddress : 32; // vaddr dest

    uint32_t crc :   23;
    uint32_t nhops :  8;
    uint32_t err :    1;

    uint32_t lflags : 32; // LUT flags

    uint32_t magicw : 32; // LUT completion word
```

Liste des tableaux

Table des figures

Bibliographie

[AGM+04] R. Ammendola, M. Guagnelli, G. Mazza, F. Palombi, R. Petronzio, D. Rossetti, A. Salamon, and P. Vicini. Apenet : a high speed, low latency 3d interconnect network. In *Cluster Computing, 2004 IEEE International Conference on*, pages 481–, Sept. 2004.

[Atm] Atmel. mAgicV VLIW DSP AND Diopsis. http ://www.atmel.com.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3) :560–599, 1984.

[BLL+08] Christopher Brooks, Edward A. Lee, Xiaojun Liu, Stephen Neuendorffer, Yang Zhao, and et al. Heterogeneous concurrent modeling and design in java (volumes 1 : Introduction to ptolemy ii). Technical report, 2008.

[BSS09] Marius Dorel Bozga, Vassiliki Sfyrla, and Joseph Sifakis. Modeling synchronous systems in bip. In *EMSOFT '09 : Proceedings of the seventh ACM international conference on Embedded software*, pages 77–86, New York, NY, USA, 2009. ACM.

[BST+06] Francesco Belletti, Sebastiano Fabio Schifano, Raffaele Tripiccione, Francois Bodin, Philippe Boucaud, Jacques Micheli, Olivier Pene, Nicola Cabibbo, Sergio de Luca, Alessandro Lonardo, Davide Rossetti, Piero Vicini, Maxim Lukyanov, Laurent Morin, Norbert Paschedag, Hubert Simma, Vincent Morenas, Dirk Pleiter, and Federico Rapuano. Computing for lqcd : apenext. *Computing in Science and Engineering*, 8(1) :18–29, 2006.

[BWH+03] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis : An integrated electronic system design environment. *Computer*, 36(4) :45–52, 2003.

[cFC07] Wu chun Feng and Kirk Cameron. The green500 list : Encouraging sustainable supercomputing. *Computer*, 40 :50–55, 2007.

[COB95] P.H. Chou, R.B. Ortega, and G. Borriello. The chinook hardware/software co-synthesis system. *System Synthesis, International Symposium on*, 0, 1995.

[CP08] A. Chureau and F. Petrot. An intermediate format for automatic generation of MPSoC virtual prototypes. In *Embedded Computer Systems : Architectures, Modeling, and Simulation, 2008. SAMOS 2008. International Conference on*, pages 165 –172, jul. 2008.

[Esa] Esaote. Esaote. http ://www.esaote.com/.

[FBB+97] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux oskit : a substrate for kernel and language research. In *SOSP '97 : Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 38–51, New York, NY, USA, 1997. ACM.

[Fea95] Paul Feautrier. Compiling for massively parallel architectures : a perspective. *Microprocessing and Microprogramming*, 41(5-6) :425 – 439, 1995. Parallel programmable architectures and compilation.

[FHN+03] Wu-chun Feng, Justin (Gus) Hurwitz, Harvey Newman, Sylvain Ravot, R. Les Cottrell, Olivier Martin, Fabrizio Coccetti, Cheng Jin, Xiaoliang (David) Wei, and Steven Low. Optimizing 10-gigabit ethernet for networks of workstations, clusters, and grids : A case study. In *SC '03 : Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 50, Washington, DC, USA, 2003. IEEE Computer Society.

[Fly72] M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, C-21 :948+, 1972.

[FSLM02] Jean-Philippe Fassino, Jean-Bernard Stefani, Julia L. Lawall, and Gilles Muller. Think : A software framework for component-based operating system kernels. In *ATEC '02 : Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 73–86, Berkeley, CA, USA, 2002. USENIX Association.

[GEA] GEA. 10 Gigabit Ethernet Alliance. http ://www.10gea.org/.

[GKK+08] Lei Gao, Kingshuk Karuri, Stefan Kraemer, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Multiprocessor performance estimation using hybrid simulation. In *DAC*, pages 325–330, 2008.

[GP09] Xavier Guérin and Frédéric Pétrot. A system framework for the design of embedded software targeting heterogeneous multi-core SoCs. In *ASAP*, 2009.

[GPY+07] Xavier Guerin, Katalin Popovici, Wassim Youssef, Frédéric Rousseau, and Ahmed Amine Jerraya. Flexible application software generation for heterogeneous multi-processor system-on-chip. In *COMPSAC*, pages 279–286, 2007.

[HAB+09] Jim Holt, Anant Agarwal, Sven Brehmer, Max Domeika, Patrick Griffin, and Frank Schirrmeister. Software standards for the multicore era. *IEEE Micro*, 29 :40–51, 2009.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8) :666–677, 1978.

[HPC] HPC. TOP500 Supercomputing Sites. http ://www.top500.org/.

[HV87] M.A. Holliday and M.K. Vernon. A generalized timed petri net model for performance analysis. *IEEE Transactions on Software Engineering*, 13 :1297–1310, 1987.

[IB07] Khaled Z. Ibrahim and François Bodin. Implementing Wilson-Dirac Operator on the Cell Broadband Engine. Research Report, 2007.

[Inf] Infiniband. Infiniband Trade Association. http ://www.infinibandta.org/.

[ITR07] ITRS Group. ITRS Report on System Drivers. http ://ecos.sourceware.org, 2007.

[KAV+] Ali Koudri, Denis Aulagnier, Didier Vojtisek, Christophe Moy, Joël Champeau, Jorgiano Vidal, and Jean christophe Le Lann. Using marte in a co-design methodology.

[KDH+05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5) :589–604, 2005.

[LJW+04] Jiuxing Liu, Weihang Jiang, Pete Wyckoff, Dhabaleswar K. Panda, Dhabaleswar K. P, David Ashton, William Gropp, Darius Buntinas, and Brian Toonen. Design and implementation of mpich2 over infiniband with rdma support. In *In In Proceedings of Int£l Parallel and Distributed Processing Symposium (IPDPS 04*, 2004.

[LM87] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. In *Proc. of the IEEE*, Sept. 1987.

[LMG+04] Philip Levis, Sam Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric Brewer, and David Culler. The emergence of networking abstractions and techniques in tinyos. In *NSDI'04 : Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 1–1, Berkeley, CA, USA, 2004. USENIX Association.

[LP95] E.A. Lee and T.M. Parks. Dataflow process networks. In *Proc. of the IEEE*, May. 1995.

[Ma05] José E. Moreira and al. Blue gene/l programming and operating environment. *IBM Journal of Research and Development*, 49(2-3) :367–376, 2005.

[Mas02] Anthony Massa. *Embedded Software Development with eCos*. Prentice Hall Professional Technical Reference, 2002.

[MPB09] Amin El Mrabti, Frédéric Pétrot, and Aimen Bouchhima. Extending ip-xact to support an mde based approach for soc design. In *DATE*, pages 586–589, 2009.

[MT08] Yannis Mazzer and Bernard Tourancheau. Mpi in wireless sensor networks. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 334–339, Berlin, Heidelberg, 2008. Springer-Verlag.

[NSD08] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(3) :542–555, March 2008.

[NXP] NXP. Nexperia. http ://www.nxp.com.

[OK06] S-C. Oh and S-W. Kim. An Efficient Linux Kernel Module supporting TCP/IP Offload Engine on Grid. In *GCC '06 : Proc. of the Fifth International Conference on Grid and Cooperative Computing*, pages 228–235, Washington, DC, USA, 2006. IEEE Computer Society.

[OOJ98] Mattias O'Nils, Johnny Öberg, and Axel Jantsch. Grammar based modelling and synthesis of device drivers and bus interfaces. In *EUROMICRO '98 : Proceedings of the 24th Conference on EUROMICRO*, page 10055, Washington, DC, USA, 1998. IEEE Computer Society.

[Par95] Thomas Martyn Parks. *Bounded scheduling of process networks*. PhD thesis, Berkeley, CA, USA, 1995.

[PBP04] H. V. Shah P. Balaji and D. K. Panda. Sockets vs RDMA Interface over 10-Gigabit Networks : An In-depth analysis of the Memory Traffic Bottleneck. In *Workshop on Remote Direct Memory Access : Applications, Implementations, and Technologies (RAIT) ; in conjunction with IEEE International Conference on Cluster Computing (Cluster), San Diego, CA*, September 20th 2004.

[PEP06] Andy D. Pimentel, Cagkan Erbas, and Simon Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Comput.*, 55(2) :99–112, 2006.

[Qui46] W. V. Quine. Concatenation as a basis for arithmetic. In *The Journal of Symbolic Logic*, volume 11, pages 105–114. Association for Symbolic Logic, 1946.

[RFC] RFC1122. Requirements for Internet Hosts – Communication Layers. http ://-tools.ietf.org/html/rfc1122.

[SBF+07] Thomas Sporer, Michael Beckinger, Andreas Franck, Iuliana Bacivarov, Wolfgang Haid, Kai Huang, Lothar Thiele, Pier S. Paolucci, Piergiovanni Bazzana, Piero Vicini, Jianjiang Ceng, Stefan Kraemer, and Rainer Leupers. SHAPES - a Scalable Parallel HW/SW Architecture Applied to Wave Field Synthesis. In *Proc. of the International Conference on Audio Engineering Society, Hillerod, Denmark*, 2007.

[SHA] SHAPES. Projet européen SHAPES. http ://www.shapes-p.org/twiki/bin/view/ShapesPublic/WebHome.

[Spi] Spirit. Spirit Consortium. http ://www.spiritconsortium.org/home/.

[ST] ST. Nomadik. http ://www.st.com.

[SYN09] Scott Schneider, Jae-Seung Yeom, and Dimitrios S. Nikolopoulos. Programming multiprocessors with explicitly managed memory hierarchies. *Computer*, 42(12) :28–34, 2009.

[SZT+04] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. System design using kahn process networks : The compaan/laura approach. In *In Proceedings of the Design, Automation and Test in Europe Conference*, page 2004, 2004.

[Tar] Target Compiler Technology. Chess/Checkers, a retargetable tool-suite for embedded processors. http ://www.retarget.com/doc/target-whitepaper.pdf.

[TBHH07] Lothar Thiele, Iuliana Bacivarov, Wolfgang Haid, and Kai Huang. Mapping applications to tiled multiprocessor embedded systems. In *ACSD '07 : Proceedings of the Seventh International Conference on Application of Concurrency to System Design*, pages 29–40, Washington, DC, USA, 2007. IEEE Computer Society.

[Tex] Texas Instrument. OMAP. http ://www.omap.com.

[VdLG+09] Jorgiano Vidal, Florent de Lamotte, Guy Gogniat, Philippe Soulard, and Jean-Philippe Diguet. A co-design approach for embedded system modeling and code generation with uml and marte. In *DATE*, pages 226–231, 2009.

[WHLM03] Oliver Wahlen, Manuel Hohenauer, Rainer Leupers, and Heinrich Meyr. Instruction scheduler generation for retargetable compilation. *IEEE Design and Test of Computers*, 20 :34–41, 2003.

[ZKKC03] W. Zhang, M. Karakoy, M. Kandemir, and G. Chen. A compiler approach for reducing data cache energy. In *ICS '03 : Proceedings of the 17th annual international conference on Supercomputing*, pages 76–85, New York, NY, USA, 2003. ACM.

Résumé

Dans cette étude, nous nous intéressons aux outils de génération de logiciel embarqué ciblant des plateformes multi-tuiles hétérogènes. Dans ces plateformes, un système sur puce multiprocesseurs hétérogène (ou tuile) est répliqué et connecté par des réseaux externes à la tuile, extensibles et commutés par paquets. Ces outils se basent sur une représentation abstraite de l'architecture, de l'application et du déploiement des éléments applicatifs sur les éléments de l'architecture.

Programmer de zéro ces architectures complexes n'est pas concevable, cependant nous ne pouvons nous contenter des environnements de programmation embarqués classiques en raison de l'hétérogénéité de la tuile de base, qui embarque des RISCs, des DSPs et une infrastructure interne à la tuile non uniforme et complexe. L'un des enjeux dans ce contexte est de masquer cette complexité au programmeur de l'application pour qu'il puisse se concentrer sur l'écriture de son programme sans se soucier dans un premier temps de son déploiement sur la plateforme cible. L'une des difficultés des systèmes multi-tuiles est le nombre de chemins de communication que ceux-ci proposent, c'est pourquoi nous nous concentrons dans ce manuscrit sur la gestion (transparente pour le programmeur) des communications dans notre flot.

Nous définissons donc les informations minimales à inclure dans le modèle d'entrée du flot pour arriver à synthétiser les communications de l'application. Grâce à ces informations, nous arrivons à puiser les composants logiciels de communication adéquats, qui se présentent sous la forme de pilotes d'un système d'exploitation. Cette sélection n'est pas suffisante, il faut ensuite spécialiser ces composants pour chaque canal de communication de l'application afin d'arriver à un résultat correct. En raison du nombre d'unités de calcul de la plateforme ciblée et du nombre de canaux des applications considérées, une automatisation totale du flot est requise, nous abordons donc les difficultés que cela représente en raison du processus de compilation croisé mis en jeu par le flot, et la solution que nous avons retenue pour arriver à un flot fonctionnel. Trois applications (dont une appartenant au monde du calcul de haute performance), écrites par des programmeurs ne maîtrisant pas la plateforme multi-tuiles choisie, ont été soumises à notre flot, qui a généré de manière correcte plusieurs déploiements de ces applications.

Mots-clés : communications, logiciel embarqué, flot de génération, MPSoC hétérogène, calcul de haute performance

Communication Synthesis in an embedded software generation environment targeting multi-tile architectures

Abstract

The aim of this thesis is to study an embedded software generation flow targeting heterogeneous multi-tile platforms. In these platforms, many heterogeneous multiprocessor system-on-chip are replicated and interconnected by scalable and packet-switched networks. This flow is fed with an abstract representation of the architecture, the application and the mapping of application elements into the architecture blocks.

Programming these complex architectures from scratch is not conceivable, however we cannot consider using classical embedded programming environments because of the heterogeneity of the tile, which is composed of RISCs, DSPs and a non-uniform intra-tile communication fabric. One of the challenges in this context is to mask this complexity to the application programmer so that he can concentrate on the design of his application without caring in a first phase about the mapping of his application into the chosen platform. One of the difficulties introduced by multi-tile systems is the high number of existing communication paths, this is the reason why this thesis focuses on the handling of communications in our flow, and how we manage to make this transparent for the programmer,

We define the minimal information that have to be included in the input high-level models to synthetize the communications of the application. Thanks to that, the flow can select the correct software communication components, which are written under the form of device drivers of an operating system. This selection phase is not enough, each communication component affected to an application channel has to be specialized in order for the flow to generate correct binaries.

In reason of the great number of processors in the target multi-tile platform and of the great number of channels of the considered applications, a complete automation of the flow is required. We evoke the difficulty of this automation, principally in reason of the cross-compilation process introduced by our flow, and then describe the solution that we have chosen to reach an operational flow. Three applications (one belonging to the high-performance computing world), written by programmers not mastering the chosen multi-tile platform, were successfully passed through our flow, which generated correctly several mapping versions of each application.

Key-words : communications, embedded software, generation flow, heterogeneous MPSoC, high-performance computing