THÈSE DE DOCTORAT DE L'UNIVERSITÉ PARIS VI

SPÉCIALITÉ INFORMATIQUE

Présentée par Etienne FAURE

Pour obtenir le titre de

DOCTEUR DE L'UNIVERSITÉ PARIS VI

COMMUNICATIONS MATÉRIELLES/LOGICIELLES DANS LES SYSTÈMES SUR PUCES MULTI PROCESSEURS ORIENTÉS TÉLÉCOMMUNICATIONS

Soutenue le 27 Avril 2007, devant le jury composé de

M. Olivier SENTIEYS	Rapporteur
M. Pascal SAINRAT	Rapporteur
Mme. Nathalie DRACH-TEMAM	Examinatrice
M. Frédéric PÉTROT	Examinateur
M. Alain GREINER	Directeur de thèse

Résumé

Cette thèse présente un intergiciel de communication entre tâches dans le contexte des systèmes embarqués sur puce. L'application est décrite sous la forme d'un graphe de tâches communicantes. Dans ce graphe, les tâches productrices et les tâches consommatrices associées à chaque canal sont en nombres quelconques. On représente donc explicitement les communications dans ce graphe pour aboutir à un graphe bi-partite.

Dans un tel graphe, les tâches peuvent être implantées sous la forme de threads logiciels ou de coprocesseurs spécialisés. On souhaite cependant conserver un mécanisme de communication uniforme, quelle que soit la nature matérielle ou logicielle des tâches.

Ces contraintes nous conduisent à spécifier des canaux de communication par mémoire partagée et un protocole de communication en 5 étapes pour y accéder. Ce protocole est implanté sous la forme d'une bibliothèque de fonctions logicielles et sous la forme d'un contrôleur matériel permettant à un coprocesseur d'utiliser ces canaux de communication.

Cet intergiciel est évalué dans deux cadres ; d'abord dans le cadre du décodage d'image MJPEG, pour prouver sa capacité à émuler le comportement des graphes de Kahn. Ensuite, comme exemple d'un contexte multi-producteurs, multi-consommateurs, on utilise une application de classification et ordonnancement de paquets IP. Pour cette application, on développe une architecture adaptée, équipée de coprocesseurs spécialisés gérant les entrées/sorties du système. Cette seconde application est également utilisée pour mettre en évidence l'impact sur les performances et la consommation électrique du placement des canaux logiciels sur les bancs de mémoire.

Mots clefs

Parallélisme, Communications sur puce, MPSOC, Codesign matériel-logiciel, Architecture NUMA.

Abstract

This thesis presents a communication middleware for tasks running on systems-on-chip. The application is described as a communicating task graph. In this graph the number of producting tasks and number of consumming tasks both are unknown. Thus we have to explicitely represent communication channels in the graph, leading to a bi-partite graph.

In such a graph, the implementation of tasks may be done either with software thread, or dedicated hardware accelerator. Nevertheless we wish to keep a uniform communication mecanism, regardless of the nature of the tasks.

these constraints lead us to specify communication channels, using shared memory, which can be accessed by a five steps communication protocol. This protocol is implemented both as a software library and as a hardware controller. This controller allows a specialized coprocessor to use these communication channels.

This middleware is evaluated in two contexts. Firstly in the context of a MJPEG picture decoder, that proves its ability to emulate Kahn network process. Secondly, as an example of multi-producer, multi-consumers context, an application of classification/scheduling of IPv4 packets is implemented. A dedicated hardware platform is designed for this application. It comprises specialized coprocessors that manage its input/output issues. This application is also used to show the impact of the mapping of the tasks and the communication channels on the performances and the power consumption.

Keywords

Parallelism, On-chip communications, MPSOC, Hardware-software codesign, NUMA architecture.

Table des matières

Table des figures

Chapitre 1

Introduction

La capacité d'intégration des systèmes embarqués sur puce a crû de façon exponentielle au cours de ces dernières années, au point que l'on peut aujourd'hui placer sur la même puce un système de traitement complet : périphériques d'entrée/sortie, processeurs, coprocesseurs et mémoire. Cette complexification des puces permet d'y déployer des applications de plus en plus complexes. De telles applications comportent presque toujours une partie matérielle et une partie logicielle. Un ou plusieurs processeurs exécutent la partie logicielle, tandis que la partie matérielle est exécutée par des coprocesseurs spécialisés.

Pour exploiter efficacement les ressources offertes par les architectures multiprocesseurs, les applications doivent être en mesure d'exploiter le parallélisme que ces architectures offrent. L'exploitation de ce parallélisme revient à paralléliser l'application en un certain nombre de tâches communicantes. Ces tâches peuvent communiquer entre elles de différentes manières, par message ou par mémoire partagée principalement.

Cependant, bien que les outils de CAO, et notamment les simulateurs d'architectures aient fait d'importants progrès, il convient de restreindre le domaine à explorer sous peine de voir le temps de développement d'un système mixte matériel/logiciel s'allonger démesurément.

Pour cette raison, on ne s'intéresse qu'aux communications par mémoire partagée. De même on restreint l'espace des solutions à celui des systèmes multi processeurs à mémoire partagée.

On s'intéresse principalement aux applications télécom visant la manipulation de paquets ; on montrera cependant que certains résultats obtenus peuvent être extrapolés à d'autres types d'applications.

1.1 Cadre de cette thèse

Cette thèse a été réalisée au sein de l'équipe ASIM du LIP6. Elle s'inscrit comme un prolongement de la chaine d'outil Disydent, également développé au LIP6. Disydent est une méthode de conception de systèmes intégrés qui s'appuie sur la description de l'application sous la forme d'un graphe de Kahn.

Cette thèse s'appuie aussi sur la bibliothèque de composants virtuels SOCLIB. L'objet du projet SOCLIB est la constitution d'une bibliothèque de composant facilement interconnectables. Ce projet est le fruit de la collaboration d'universités et d'industriels européens à laquelle participe le LIP6.

1.2 Organisation de ce document

Dans un premier temps, nous exposerons la problématique de notre travail en formulant les questions auxquelles on se propose de répondre. Nous étudierons ensuite dans le chapitre intitulé *état de l'art* différentes solutions apportées aux problèmes des communications entre tâches, lorsque celles-ci sont embarquées sur une puce.

Nous expliquerons ensuite comment une application peut être parallélisée et quel est le graphe de tâches résultant. On verra que la mise en évidence du parallélisme d'une application entraine sa représentation dans un graphe bi-partite où les canaux de communication sont représentés explicitement. Cet exposé nous permettra de cerner les besoins pour les canaux de communication que l'on souhaite implémenter. On verra notamment qu'il doivent être capables d'accepter un nombre quelconque de tâches, ce qui déterminera leur nom : canaux MWMR pour Multi-Writers, Multi-Readers.

Le chapitre suivant étudie plusieurs implémentations des spécifications établies précédemment et compare leur performances.

Dans le chapitre 6, on évalue les canaux de communications dans le cadre des graphes de Kahn. On utilise pour cela une application de référence de décodage d'image MJPEG pour laquelle on utilise successivement deux types de canaux, d'abord des canaux optimisés spécifiquement pour l'utilisation dans le contexte des graphes de Kahn, puis nos canaux MWMR. Les performances obtenues pour les deux versions font l'objet d'un comparatif.

Dans le chapitre suivant, on évalue les canaux de communication dans le cadre pour lequel ils ont été développés : une application télécom de manipulation de paquets. Pour cette évaluation, on met au point une architecture multi-processeurs, multi-clusters disposant de coprocesseurs d'entrée/sortie. Cette étude porte aussi sur l'importance du placement des canaux de communications sur les bancs mémoire. Ce placement a des répercussions sur les performances de l'application et sur la consommation électrique de la puce. On termine par une conclusion et par quelques perspectives qui demeurent ouvertes.

Chapitre 2

Problématique

Sommaire

CE travail s'inscrit dans le contexte de la conception de systèmes embarqués sur puce. Un système embarqué est un ensemble matériel/logiciel conçu pour exécuter une application donnée. Cette application peut appartenir à différents domaines (télécommunications, vidéo, automobile) dont le caractère commun est la grande évolutivité.

La capacité d'intégration des systèmes embarqués sur puce a crû de façon exponentielle au cours de ces dernières années, au point que l'on peut aujourd'hui placer sur la même puce un système de traitement complet : périphériques d'entrée/sortie, processeurs, coprocesseurs et mémoire. Cette complexification des puces permet d'y déployer des applications de plus en plus complexes. Les contraintes économiques toujours plus fortes font en revanche que le temps imparti au développement de ces applications est paradoxalement de plus en plus court. Les méthodes de développement de ces applications doivent donc elles aussi évoluer pour prendre en compte cette complexité croissante :

– Au point de vue du matériel mis en œuvre, il faut absolument permettre de ré-utiliser des composants préexistants (IP reuse), ces composants pouvant être des cœurs de processeurs, des mémoires ou des coprocesseurs spécialisés. La conception d'une nouvelle architecture de puce se limite autant que possible à l'interconnexion de composants existants.

– Du point de vue logiciel on cherche également à réutiliser les composants logiciels existants. On cherche donc à minimiser l'impact des caractéristiques propres à chaque plateforme matérielle. Pour cela il faut homogénéiser l'interface de programmation de ces puces. Cela consiste entre autre à proposer une API (Application Programming Interface) standardisée pour le développement des applications logicielles.

Ces deux contraintes, le réemploi de composants préexistants et le masquage de la complexité reposant sur la définition d'abstractions, ne sont pas spécifiques à la conception de systèmes intégrés sur puce mais elles sont particulièrement importantes, puisque le concepteur système peut modifier à la fois les paramètres de l'architecture matérielle et les caractéristiques de l'application logicielle.

L'augmentation de la capacité d'intégration des systèmes sub-microniques a aussi pour conséquences d'agrandir considérablement l'espace des solutions architecturales disponibles. On est en effet passé d'un monde où l'architecture de référence était constituée d'un processeur relié à une mémoire et à quelques périphériques par un bus système, à un monde où l'on peut concevoir des systèmes embarquant des dizaines de cœurs de processeurs, plusieurs bancs de ram, les uns connectés aux autres par le biais d'un micro- réseau sur puce.

Cette complexité croissante des techniques mises en œuvre, ainsi que l'extraordinaire aggrandissement de l'espace des solutions ont amené à concevoir de nouveaux outils d'aide à la conception. Ces outils doivent permettre l'exploration de l'espace des solutions matérielles (nombre de processeurs, taille des caches, nombre de bancs de mémoire), ainsi que celui des solutions logicielles, plusieurs implantations pouvant être en concurence pour une même application (différents découpages de l'application logicielle en tâches capables de s'exécuter en parallèle). Cependant, bien que les outils de CAO, et notamment les simulateurs d'architectures aient fait d'importants progrès, il convient de restreindre le domaine à explorer sous peine de voir le temps de développement d'un système mixte matériel/logiciel s'allonger démesurément.

C'est dans cette optique que nous posons des limites au champ d'investigation de notre recherche. Ces limites concernent, d'une part, la gamme des applications candidates à un déploiement sur puce : on s'intéresse principalement aux applications télécom visant la manipulation de paquets. D'autre part, on restreint l'espace des solutions à celui des systèmes multi processeurs à mémoire partagée. On montrera cependant dans ce document que certains résultats obtenus peuvent être extrapolés à d'autres types d'applications.

2.1 Caractéristiques des applications télécom

Les applications télécom possèdent certains points communs :
1) Les données en entrées sont constituées d'une suite de paquets n'ayant pas nécessairement de liens entre eux : le trafic est composé de différents flux entrelacés. Les données d'un flux sont indépendantes de celles des autres flux.
2) Pour les paquets appartenant à un même flux, le traitement est généralement identique : routage vers la même sortie, encapsulation, défragmentation, etc...

On appelle un flux la suite de paquets qui assurent une communication entre deux agents sur le réseau (un client web et un serveur par exemple). Les paquets d'un même flux ont donc de nombreux points communs : même adresse source, même adresse destinataire, mêmes protocoles employés. Par la suite, on verra que cette définition peut être assouplie, et que chaque application a sa propre définiton d'un flux : une application de routage considère comme appartenant au même flux tous les paquets à destination d'un de ses sous réseaux de sortie. Une application de qualité de service (QoS) conserve au contraire une définiton stricte de la notion de flux puisque son but est de déterminer la priorité d'un paquet en fonction du flux auquel il appartient. La définition assouplie garantit donc que tous les paquets d'un flux vont subir le même traitement. Par contre cela ne garantit pas que tous les paquets subissant le même traitement fassent partie du même flux.

Parmi les différents types d'applications que l'on peut trouver sous la dénomination applications réseau, on trouve des choses très différentes. Une première classification distingue deux types : les applications qui travaillent sur les données des paquets et celles qui travaillent sur leurs en-têtes. Dans la première famille, on trouve, par exemple, les applications de cryptage de données. Ce sont des applications qui nécessitent d'importants calculs sur les données des paquets. Elles traitent des débits de paquets relativement faibles. Dans la deuxième famille, on trouve des applications de routage et de classification de flux. Ce type d'application ne travaille que sur l'en-tête des paquets, sur lequel il n'effectue que peu de modifications. Ce sont des applications où la quantité de calcul à fournir par paquet est faible. Ces applications doivent en revanche faire face à des débits très importants : environ 10 Gb/s pour un routeur back bone.

Dans un cas comme dans l'autre, ces applications ont de gros besoins en puissance de calcul, que ce soit pour effectuer le cryptage des données, ou pour analyser les en-têtes d'un très grand nombre de paquets. Une autre caractéristique des applications réseau est la présence d'une mémoire partagée où sont stockées les informations de contrôle de l'application. Ces informations sont par exemple les tables de routage dans le cas d'un routeur IPv4, ou la table des flux dans le cas d'une application de type Qualité de service (QoS) ou "traffic management". Cette dernière table contient alors la liste des actions à effectuer pour tous les paquets d'un même flux, ainsi qu'éventuellement, des informations statistiques sur ces

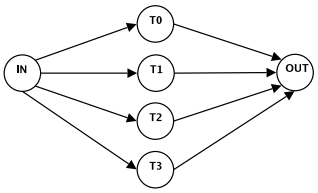

FIG. 2.1 – Schéma d'une application parallélisée selon un modèle task farm

flux. Ces tables atteignent facilement des tailles importantes : RFC 1812 [Bak95] recommande qu'un routeur IPv4 contiennent 65000 entrées dans ses tables de routage, [DBCP97] en préconise 40000 pour ce même routeur. Quand aux applications de QoS, elles visent des réseaux où il faut pouvoir traiter jusqu'à un millier de flux distincts.

Les deux types d'applications décrits ci-dessus nécessitent un important travail de parallélisation de l'application pour pouvoir s'exécuter efficacement sur une architecture multi-processeur. Cette extraction du parallélisme est une tâche non triviale qui doit, à notre avis, être effectuée par le concepteur de l'application. D'une façon générale l'organisation des données en paquets se prête plus facilement à la parallélisation "task farm", c'est à dire l'instanciation multiple de la même tâche de traitement. Toutes les instances partagent les mêmes entrées et sorties de données. L'architecture d'une application ainsi parallélisée ressemble au schéma de la figure 2.1.

L'application, ainsi parallélisée, peut plus facilement être portée sur une plateforme matérielle multi-processeurs ; chaque tâche pouvant s'exécuter sur un des processeurs.

On voit sur la figure 2.1 que les tâches IN et OUT communiquent avec plus d'une tâche chacune, et que ces tâches se retrouvent en concurence pour accéder aux données.

Il existe un formalisme de modélisation des communications dans les applications multi-tâches connu sous le nom de réseau de Kahn [Kah74]. Ce formalisme s'appuie sur des communications par mémoire partagée, par le biais de canaux point à point de type fifo. Les réseaux de Kahn ont des propriétés intéressantes, notamment le déterminisme. Le déterminisme des graphes de Kahn est la propriété grâce à laquelle, toutes les exécutions d'un même graphe de tâches donneront le même résultat final, quelles que soient les durées d'exécution des tâches. Grâce à cette propriété, le choix d'une implantation matérielle (un coprocesseur spécialisé) ou logicielle (une tâche fonctionnant sur un processeur programmable) n'a pas d'influence sur le *résultat* du programme, bien qu'il en ait sur le temps d'exécution du programme.

Cette caractéristique est particulièrement intéressante dans le contexte des ap-

plications embarquées. Cela permet de développer l'application sur une station de travail en ayant la certitude qu'elle fonctionnera de façon identique une fois déployée sur une architecture embarquée multi-processeur.

Cette propriété est due aux contraintes particulières que doivent respecter les KPN : les communications se font point à point entre deux et seulement deux tâches. C'est à dire qu'à chaque canal de communication sont connectées une tâche productrice de données, et une tâche consommatrice. Les lectures sont bloquantes : elles se terminent toujours lorsque la requète est satisfaite. Enfin, les fifos ont une profondeur infinie, ce qui fait que les écritures ne sont jamais bloquées. Cette dernière contrainte n'est pas réaliste dans le cadre d'une implantation effective, et encore moins pour une applicaiton embarquée, mais PARKS [Par95] a montré que l'on pouvait assouplir cette condition. On peut fixer la profondeur des fifos en ajoutant la contrainte corollaire qui est de rendre les écritures bloquantes en cas de fifo pleine. Les propriétés des graphes de Kahn sont conservées, à la réserve près que cette modification introduit le risque d'interbloquage si les fifos sont de profondeur insuffisantes.

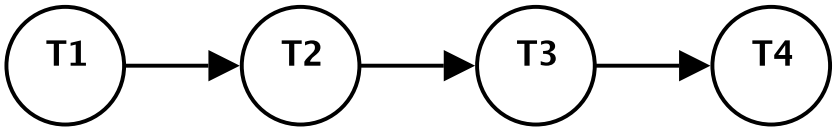

FIG. 2.2 – Exemple de graphe de tâche, utilisant le parallélisme de type pipeline

Le formalisme des grahes de Kahn est bien adapté aux applications parallèles de type pipeline où la synchronisation est réalisée par la disponibilité des données (voir la figure 2.2 pour un exemple). Le principe du parallélisme pipeline repose sur le découpage de l'application en gros grains, chacun de ces grains devenant un étage du pipeline, exécuté par une tâche. Plus l'application effectue de calculs important, et plus elle se prête à une parallélisation pipeline. Ce type de parallélisme se rencontre principalement dans les applications vidéo et multimédia, du type compression/décompression d'un flux audio/vidéo où chaque étage du pipeline effectue une partie du traitement, et où il est important que l'ordre des données soit conservé.

Ce formalisme est en revanche assez mal adapté pour les applications réseau, et particulièrement pour celles de type routage ou classification de paquets. De telles applications se caractérisent par une demande de débit très élevée, mais un besoin de calcul par paquet faible. De ce fait, ce sont de mauvaises candidates à la parallélisation pipeline, puisqu'elles se prêtent mal à un découpage en de nombreux étages. Ce type d'application se prête mieux à la parallélisation task farm, illustrée sur la figure 2.1. Mais dans ce graphe de tâche, on souhaite que la tâche IN émette ses paquets à destination de l'ensemble des tâches de traitement,

plutôt que d'envoyer chaque paquet à une tâche particulière. Or la sémantique des réseaux de Kahn nous impose que chaque canal de communication soit un canal point à point. Cette contrainte est très lourde, puisqu'elle force la tâche IN à avoir autant de canaux de communication qu'il y a d'instances de tâches d'exécution. De plus, les communications étant bloquantes, la tâche IN peut rester bloquée sur l'envoi d'un paquet vers une tâche d'exécution, alors que d'autres instances de cette tâche sont oisives. Dans ce cas, le déterminisme des KPN a un effet négatif. Ceci met en évidence que les canaux de communication point à point entre deux agents, un producteur et un consommateur, ne sont pas adaptés ici, et on voit apparaître le besoin de canaux de communication de 1 vers N tâches, ainsi que de N vers 1 tâche. La forme générale de ces nouveaux canaux de communication est N vers M, avec N et M supérieurs ou égaux à un.

Si les applications réseau montrent que les réseaux de Kahn ne sont pas adaptés à ce type d'applications, On souhaite néanmoins définir un modèle de communication qui conserve les bonnes propriétés des KPN pour les applications supportant ce formalisme. La question est donc : *Comment généraliser le modèle de communications utilisé dans des réseaux de Kahn pour l'adapter aux applications réseau ?*

2.2 Caractéristiques de l'architecture matérielle

Les progrès des techniques d'intégration sur silicium permettent aujourd'hui de réaliser des systèmes multi-processeurs intégrés sur puce (MPSoC). Ce parallélisme matériel peut être élevé puisqu'un MPSoC peut contenir plusieurs dizaines de cœurs de processeurs.

Cependant, malgré les progrès des capacités d'intégration, les contraintes techniques imposent des limites. La plus importante concerne la quantité de mémoire embarquée. La capacité de mémoire embarquée est limitée. Dans les systèmes intégrés, tous les éléments qui composent le circuit sont sur la même puce : processeurs, coprocesseurs spécialisés et au moins une partie de la mémoire. Et la surface finale de la puce est directement liée à la quantité de composants qu'elle contient. Or cette surface a une incidence sur deux facteurs : le coût de fabrication de la puce, qui est d'autant plus élevé que la puce est grosse ; le second facteur est la puissance électrique qu'elle consomme en fonctionnant : celle-ci est aussi directement liée à la surface de la puce. Or dans le contexte de systèmes embarqués, la consommation électrique doit être faible. La quantité de mémoire embarquée disponible est donc très inférieure à celle que l'on peut trouver sur une station de travail : quelques centaines de Ko, contre quelques centaines de Mo.

L'augmentation du nombre de cœurs de processeurs sur la puce a amené à revoir leur connexion avec la mémoire. C'est ainsi que le bus système a été remplacé

par la technologie des micro-réseaux sur puce (NoC, pour Network on Chip). L'utilisation de micro-réseau sur puce augmente la bande passante de façon appréciable : la bande passante disponible augmente avec le nombre d'agents (processeurs et mémoires) connectés. L'accès à l'interconnect n'est plus un goulot d'étranglement pour ces sytèmes multi processeurs. Pour éviter de reporter ce goulot d'étranglement sur la mémoire, il convient alors de la répartir en un certain nombre de bancs physiques distincts.

L'introduction des réseaux sur puce en lieu et place du bus sytème introduit le problème de la cohérence des données. Dans le cas d'un bus, cette cohérence peut être assurée par le mécanisme d'espionnage du bus (snoop). Ceci n'est plus possible avec les micro-réseau intégrés qui permettent plusieurs transactions simultanées. Les solutions matérielles proposées sont particulièrement coûteuses (voir par exemple le mécanisme Directory based de l'architecture DASH [HHG99]), mais la perte de performances due à une exécution sans cache est intolérable. Ce problème de cohérence se pose particulièrement dans le cadre de notre travail puisque nous souhaitons modéliser les applications candidates sous forme de graphe de tâches communicant à travers des tampons de mémoire partagée.

On a ici affaire à trois contraintes apparamment inconciliables : on souhaite faire communiquer les tâches entre elles au moyen de tampons de mémoire partagée ; on ne peut pas se passer de l'utilisation de cache pour des raisons de performances et finalement le type d'interconnect, le micro-réseau sur puce, empêche de mettre en place un mécanisme de surveillance du bus (le snoop) permettant d'assurer la cohérence des données partagées. On peut formuler le problème de la façon suivante :
comment résoudre le problème de la cohérence des caches dans une architecture multiprocesseur, où le snoop est imposible, tout en utilisant des tampons de communication en mémoire partagée ?

2.3 Caractéristiques des communications entre tâches

Les applications réseaux, comme les autres, ont tendance à se complexifier. L'extraction du parallélisme de l'application est une contrainte supplémentaire qui s'ajoute aux difficultés intrinsèques de l'application.

Dans ce contexte, il est essentiel d'abstraire autant que possible les spécificités de la plateforme matérielle pour le concepteur de l'application logicielle. On utilise pour cela un système d'exploitation multi-tâches dont l'objet est de masquer les détails de la plateforme matérielle sous une API (Application Programming Interface) connue telle que celle définie par la norme POSIX. Le rôle du système d'exploitation est de fournir une interface entre le matériel et l'application. Ceci permet au concepteur de l'application de s'affranchir (dans une certaine mesure)

des caractéristiques particulières de la plateforme matérielle visée. Lorsqu'on développe une application composée d'un grand nombre de tâches communicantes, les communications doivent être un service fourni par le système d'exploitation ou par une couche logicielle intermédiaire de communication (on parle d'intergiciel de communication).

Dans la plupart des applications télécoms, la structure du graphe qui décrit l'ensemble des tâches qui s'exécutent en parallèle et les canaux de communication entre ces tâches peut être définie de façon statique car cette structure ne varie pas à l'exécution. Mais le nombre de tâches qui utilisent (en lecture comme en écriture) un canal de communication particulier est quelconque. On ne se limite pas à des communications point à point entre un écrivain et un lecteur.

On souhaite en particulier avoir un modèle de communication "homogène" qui permette à une tâche logicielle (c'est à dire s'exécutant sur un processeur) de communiquer avec une autre tâche, que celle ci soit logicielle ou matérielle (c'est à dire réalisée par un coprocesseur spécialisé). Lorsqu'on définit le graphe des tâches communicantes, on souhaite pouvoir définir le plus tard possible le nombre d'instances de chaque tâche (une ou plusieurs), ainsi que leur nature matérielle ou logicielle. Les applications télécoms sont des applications où les données sont indépendantes les unes des autres, et où le temps de traitement nécessaire pour un paquet est très variable en fonction de son type. Il n'est donc pas connu a priori. Chaque tâche peut avoir plusieurs canaux en entrée ou en sortie, affectés de certaines priorités. C'est le cas des applications de classifications. Dans ce cas, on sert les canaux par ordre décroissant de priorité. Si cette tâche cherche à lire dans un canal de communication, mais que celui-ci est vide, il faut qu'elle puisse effectuer une tentative de lecture dans un autre canal, moins prioritaire. Cela signifie que les primitives d'accès à ces canaux, en lecture comme en écriture doivent être non bloquantes.

Ceci définit trois caractéristiques de ces canaux de communications :

– Le nombre de tâches productrices et consommatrices est quelconque.
– La nature matérielle/logicielle de chaque tâche est inconnue.
– Les méthodes d'accès aux canaux doivent être non bloquantes.

Une autre contrainte est liée au besoin de mise au point de l'application logicielle. La parallélisation d'une application logicielle est un processus complexe au cours duquel il existe un fort risque d'introduire des erreurs de synchronisation entre les tâches. Il faut donc pouvoir valider l'application logicielle avant de la déployer sur l'architecture matérielle multi-processeur visée. Donc dans cette phase de mise au point, on souhaite pouvoir exécuter cette application logicielle multi-tâche sur une station de travail standard (Linux ou autre). Faire fonctionner l'application sur une station de travail permet d'effectuer sa mise au point fonctionnelle avant de s'occuper de son déploiement sur l'architecture matérielle multi-processeur cible. Il faut donc que les tâches et les canaux de communica-

tion soient exécutables sur une station de travail. Pour les tâches, on choisit de les écrire dans un langage de programmation standard : le langage C. Pour ce qui concerne les primitives de communication utilisées par les tâches, il faut fournir une implantation de l'intergiciel de communication compatible avec le système d'exploitation de la station de travail. En pratique, on s'impose de porter l'intergiciel de communication sur l'API définie par la norme POSIX, de façon à permettre l'exécution de l'application logicielle sur n'importe quelle station de travail supportant les threads POSIX. De cette façon on peut mettre l'application au point en la compilant pour une station de travail, puis une fois cette mise au point accomplie, la compiler pour l'architecture multi-processeur cible et s'intéresser aux problèmes de performance posés par son déploiement sur cette cible.

La question posée est donc la suivante :

Comment définir un modèle de communication homogène, quelle que soit la nature matérielle ou logicielle des tâches productrices et consommatrices ?

2.4 L'importance du placement des objets logiciels sur les composants matériels

Les architectures matérielles multi-processeurs intégrées sur puce (MPSoC) sont souvent clusterisées. Elles sont constituées de sous-systèmes qui communiquent entre eux par un micro-réseau intégré. Chaque sous-système peut contenir un ou plusieurs processeur, de la mémoire locale, ainsi que différents périphériques. Tous les éléments d'un même sous-système communiquent entre eux par un bus local. L'espace d'adressage reste partagé, et n'importe quel processeur peut accéder à n'importe quelle mémoire ou périphérique, mais la mémoire est physiquement distribuée, et cette topologie fait que les bancs de RAM ne sont pas équidistants des processeurs. On parle d'architecture NUMA, pour Non Uniform Memory Access. Cette caractéristique est accentuée par la présence d'une mémoire externe dont le temps d'accès est beaucoup plus long.

C'est au cours du déploiement de l'application logicielle multi-tâches sur la plateforme matérielle que l'on met en évidence le fait que l'application et la plateforme cible doivent être mises au point conjointement.

On s'aperçoit en effet que dans une architecture multi-processeurs, multi-bancs de mémoire, la répartition des tâches sur les processeurs et des objets logiciels sur les bancs de mémoire a une forte influence sur les performances de l'ensemble du système. Ce qui met en évidence la notion de placement des objets logiciels (tâches, canaux de communication) sur les ressources matérielles (processeur, mémoire). De même que l'extraction du parallélisme d'une application, le placement de ces objets sur les composants de la puce est un travail délicat qui

reste à la charge du concepteur de l'application. Ce qui pose en fait deux questions : *Comment optimiser le placement des objets logiciels sur les composants matériels ? Quel est l'impact de ce placement sur les performances de l'application (surcoût en terme de temps d'exécution lié à un mauvais placement) ?*

Quels outils peut-on donner au concepteur pour lui permettre de contrôler et d'optimiser ce placement ?

2.5 Conclusion

L'objectif de cette thèse est donc de concevoir une plateforme cible pour l'exécution d'applications télécom. Une telle plateforme comporte deux parties : l'architecture matérielle, comportant des proceseurs programmables et des coprocesseurs spécialisés, et une bibliothèque de communications efficace.
On essaiera de répondre aux questions suivantes :

Comment peut-on aider le concepteur à expliciter le parallélisme de l'application logicielle pour lui permettre d'utiliser le parallélisme matériel offert par les systèmes multi-processeurs sur puce ?

Comment résoudre le problème de la cohérence des caches dans des architectures multi-processeurs à base de NoC, tout en utilisant des tampons de mémoire partagée ?

Peut-on définir un modèle de communication homogène qui permette des communications entre des tâches indifféremment matérielles ou logicielles ?

Quel est l'impact du placement des tâches sur les processeurs et des objets logiciels sur les bancs mémoire sur les performances du système considéré ?

Chapitre 3

Etat de l'art

Sommaire

Cet état de l'art fait un tour d'horizon de différents intergiciels de communications proposés pour les architectures multiprocesseurs intégrées sur puce (Disydent, StepNP, COSY...), ce qui est également notre cible. D'autres bibliothèques de communication pour architecture parallèle, non spécifiquement destinées aux applications embarquées sur puce, méritent l'intérêt de par la variété des types de canaux de communications qu'elles proposent. C'est dans cette seconde optique que l'on étudie le modèle de programmation MPI.

3.1 STepNP

StepNP [PPB02] est un outil d'exploration architecturale au niveau système, particulièrement destiné au domaine des processeurs réseau. Cet outil s'appuie sur le langage SystemC pour la description des composants. Parmi les composants existants, on trouve un interconnect générique, des modèles de processeurs RISC (ARMv4, PowerPC), des bancs de mémoire et des coprocesseurs. StepNP utilise la bibliothèque standard Newlib pour l'exécution du logiciel embarqué.

StepNP a été développé par STMicroelectronics en collaboration avec des universités nord- américaines.

Le coeur de StepNP est constitué par une plateforme de simulation d'une architecture multi-processeur. A cette plateforme de simulation, on peut connecter différents outils d'instrumentation. Ces outils permettent d'obtenir des informations sur l'ensemble du système en cours d'exécution. Cela comprend le nom d'une fonction en cours d'exécution sur un processeur, ou la valeur d'un signal dans l'interconnect.

StepNP met particulièrement l'accent sur l'instrumentation de la plateforme matérielle et du logiciel embarqué. Pour cela, une interface logicielle a été conçue. Il s'agit de SIDL (SystemC Interface Definition Language). Cette inteface est destinée a faciliter la communication entre une simulation systemC et des programmes de déboguage ou d'instrumentation.

Une architecture matérielle est constituée de processeurs, de mémoire et de coprocesseurs, connectés entre eux par un interconnect. De plus, StepNP propose un coprocesseur *Semaphore engine* qui permet une gestion matérielle des verrous.

L'interface des composants matériels est une interface originale. Elle se nomme SOCP (SystemC Open core Protocol). C'est une interface de communication qui emprunte beaucoup à OCP-VCI, avec en plus, la possibilité de modéliser des transactions au niveau fonctionnel, c'est à dire l'écriture ou la lecture d'une structure de données complète effectuée par une tâche, en un seul accès à l'interconnect. Ceci est réalisé en passant des tableaux d'adresses et de données à la place des adresses et données utilisées pour des transactions précises au cycle. Il s'agit d'une communication en rafale, mais où tout le contenu de la rafale est exprimé dans une seule commande. Cette interface permet donc d'avoir sur une même plateforme de simulation, plusieurs niveaux d'abstractions différents qui peuvent communiquer entre eux. En fait tout repose sur la capacité du modèle utilisé pour simuler le fonctionnenment du système d'interconnexion.

La modélisation de l'interconnect n'est pas très précise, elle ne permet pas d'observer finement les phénomènes de contention qui peuvent s'y produire.

Dans [PPB03], la plateforme StepNP est utilisée pour effectuer un portage de l'application logicielle Click [KMC+00]. Click est une application de routage IP modulaire qui permet d'écrire des applications réseaux simplement en connectant

entre eux des modules élémentaires qui effectuent des actions simples comme vérifier un entête IP, supprimer un entête Ethernet, etc. Une version parallélisée de cette application, appelée SMP Click [CM01] a été développée pour exploiter le parallélisme offert par les machines PC à plusieurs processeurs. C'est cette seconde version qui est utilisée pour le portage sur le système multi-processeurs de StepNP. Chaque processeur exécute l'application Click, et les points de synchronisation sont les coprocesseurs d'entrée/sortie, ainsi que l'accélérateur matériel NPSE (Network Processor Search Engine) [SRH+03].

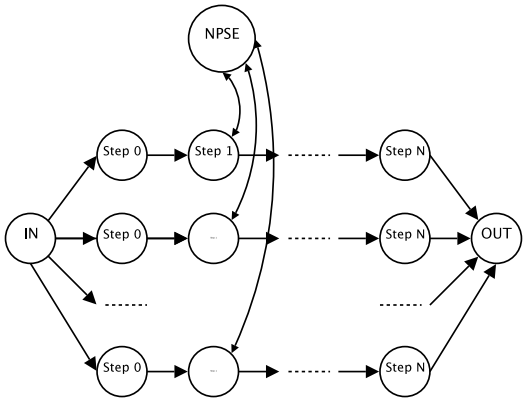

FIG. 3.1 – Schéma du graphe de tâche de l'application exécutée sur la plateforme StepNP.

Le graphe de tâches utilisé pour cette application parallèle est illustré sur la figure 3.1. Chacun des éléments *Step i* correspond à un étage de la chaîne de traitement de l'application. Chacune des chaînes est implantée sous la forme d'une tâche qui est exécutée sur un processeur. Les communications avec la tâche matérielle *NPSE* sont bidirectionnelles : la tâche envoie une requête et attend une réponse.

Ce coprocesseur NPSE est chargé de déterminer la route de sortie de chaque paquet en fonction de son adresse de destination. Ce coprocesseur a deux interfaces, l'une avec le NPU, l'autre avec la RAM, où sont stockées les tables de routage. C'est l'interface avec le NPU qui peut être comparée à une communication entre deux tâches. C'est une interface bi-directionnelle : les tâches envoient un descripteur de paquet et se voient retourner une interface de sortie en retour. Ces communications sont organisées suivant une logique fifo. Cependant, ce sont les tâches logicielles qui sont maîtresses des communications dans les deux sens, et il arrive qu'une tâche vienne demander un paquet réponse alors que celui-ci n'est pas encore prêt. Dans ce cas, le canal de communication émet un code spécial pour signaler le statut de la requête : "*not ready*", "*miss*" ou "*error*". Ces communications sont schématisées sur la figure 3.2, où l'on voit le coprocesseur NPSE, avec ses deux interfaces, l'une avec le processeur réseau NPU, l'autre avec la mémoire

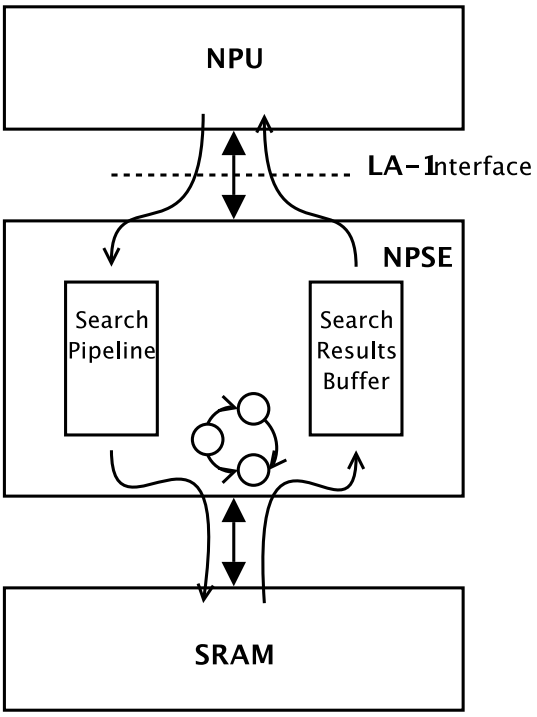

FIG. 3.2 – Schéma des communications entre le coprocesseur *NPSE*, le processeur réseau NPU et la mémoire SRAM.

SRAM. La première interface, entre le NPU et le coprocesseur est une interface de type LA-1. Le NPU vient enfiler des paquets de requêtes par cette interface dans le tampon *Search Pipeline*. Ces requêtes sont transformées en accès à la mémoire externe par un processus implémentant l'algorithme de recherche, schématisé par le petit automate. Les réponses aux accès mémoire sont enfilées dans le tampon *Search Results Buffer*.

Le fait que cette communication soit implantée en matériel a l'avantage que la gestion de la cohérence soit également effectuée par le matériel, sans surcoût ni en matériel, ni en latence. Les communications sont également très rapides : un seul cycle du coprocesseur vers le canal, et pas de surcoût dû à la prise d'un verrou pour une communication initiée par une tâche logicielle. L'inconvénient corollaire est que l'on a là affaire à un mécanisme ad hoc : le canal de communication est enfoui dans le coprocesseur, et sa modification est donc difficile.

Il n'existe pas d'équivalent aux canaux DPN ou MWMR. L'exploitation du parallélisme offert par le matériel est entièrement à la charge du développeur de l'application. Les communications entre tâches, matérielles ou logicielles doivent recevoir une implantation ad hoc.

3.2 MPI

MPI signifie Message Passing Interface.

MPI est une bibliothèque de communication par passage de messages entre des processus s'exécutant sur différents processeurs d'une machine parallèle ou d'un réseau de stations de travail. On n'est donc pas dans le cadre des communications sur puce. Néanmoins certaines des problématiques soulevées par ces communications entre agents distants se retrouvent dans les communications sur puce.

3.2.1 Principe de la programmation MPI

Il s'agit d'un modèle de programmation parallèle qui repose sur l'échange de messages. Une donnée est échangée entre deux processus par l'appel à des primitives de communication prédéfinies utilisant des canaux explicitement définis par l'utilisateur.

Le modèle de parallélisme employé est de type SPMD (Single Program, Multiple Data). Dans ce modèle de programmation, tous les processus exécutent le même code. Ce modèle peut être employé pour émuler un comportement MPMD (Multiple Program, Multiple Data). Ce comportement MPMD est obtenu en utilisant les fonctions `fork` et `exec` pour créer dynamiquemment des tâches, auxquelles on donne un code spécifique à exécuter, voir la figure 3.3 pour un exemple.

```
if (thread_num == 0) {
  thread_func_0 (Arguments);
} else if (thread_num == 1) {
  thread_func_1 (Arguments);
} ...
```

FIG. 3.3 – Exemple de code permettant une émulation du fonctionnement MIMD à l'aide du parallélisme MPI.

Dans le modèle MPI, les communications se font par passage de messages. Un message contient les données à échanger entre le producteur et le(s) consommateur(s). En plus des données, un message contient les informations suivantes :

- l'identificateur du processus émetteur
- Le type de données
- La taille du paquet de données
- l'identificateur du processus récepteur

Ces informations sont utilisées par les sous-programmes de communications pour acheminer le message, et éventuellement lui appliquer des transformations. Il

n'existe pas à proprement parler de canaux de communication par lesquels les données seraient acheminées. A leur place, on trouve la notion de communicateurs, qui sont des groupes de tâches autorisées à effectuer des communications entre elles. Lors de la création d'un programme MPI, un communicateur par défaut est créé, il s'agit du communicateur MPI_COMM_WORLD qui regroupe toutes les tâches.

Dans le modèle MPI, les communications sont gérées par un environnement qui se charge de router chaque message vers son destinataire. Les processus destinataires des messages doivent être capables de recevoir les messages. Ceci signifie que les messages ne sont distribués que lors d'une demande explicite du destinataire. Les communications sont réalisées par des appels explicites de la part des producteurs à des sous-programmes (MPI_SEND par exemple), et par des appels explicites des destinataires à d'autres sous-programmes (MPI_RECV par exemple). L'ensemble de ces sous-programmes définit la bibliothèque MPI. Les sous-programmes appartiennent à plusieurs familles :

- Communications point à point
- Communications multi-points
- Groupes et communicateurs

Il existe d'autres sous ensembles de sous-programmes, par exemple ceux servant à initialiser l'environnement du programme, mais leur existence est liée à l'utilisation concrète de la bibliothèque MPI, et ils sortent du cadre de l'exposé des principes de fonctionnement du modèle MPI.

Dans MPI, toutes les communications portent sur des *communicateurs*. Les communicateurs sont des groupes de processus qui échangent des messages entre eux. Le communicateur par défaut est l'ensemble des processus actifs. Certains sous-programmes servent à définir d'autres communicateurs qui permettent d'avoir des sous ensembles de processus qui communiquent entre eux, sans interférer avec les autres processus.

Le standart MPI offre une importante bibliothèque de communication entre un ou plusieurs consommateurs et producteurs.

3.2.2 Les fonctions de communication point à point

Le modèle de communication le plus simple est le modèle point à point, où un producteur fabrique un message et l'envoie à un consommateur.

Le producteur envoie son message à l'aide de la fonction `MPI_SEND`. Le destinataire du message est explicitement indiqué dans les paramètres de la fonction. Le consommateur reçoit ce message avec la fonction `MPI_RECV`. Un champ de cette fonction est réservé pour indiquer le numéro de l'expéditeur. Lorsque cette information n'est pas connue, ou sans importance, on utilise la valeur `MPI_ANY_SOURCE`.

Le sous-programme `MPI_SEND` ne se termine que lorsque le consommateur exécute le sous-programme `MPI_RECV`, ce qui réalise une synchronisation entre les deux processus.

Dans le modèle MPI, les communications standards sont synchrones, ce qui veut dire qu'un producteur de message attend l'accusé de réception d'un consommateur avant de continuer.

Il existe plusieurs possibilités d'effectuer une communication point à point avec MPI. Il existe notamment la possibilité de faire de façon atomique un envoi suivi d'une réception. Ce type de communication envoi-reception permet d'éviter les interbloquages décrits plus haut.

3.2.3 Communications bloquantes et communications non bloquantes

Les fonctions qui implémentent les envois et les réceptions de données (respectivement `MPI_SEND` et `MPI_RECV`) sont fondamentalement bloquantes : elles ne retournent à la tâche appelante qu'une fois la transaction achevée. Cependant, d'autres types de synchronisation entre tâche émetrice et tâche réceptrice sont disponibles. Ils ont pour but principal l'optimisation des communications entre les tâches afin d'éviter des synchronisations inutiles. Pour effectuer cette optimisation, le programmeur utilisant MPI a à sa disposition quatre modes d'envoi des données :

mode standard : un envoi peut être démarré même si la réception correspondante n'a pas été initiée. Cet envoi ne se termine que lorsque le récepteur a reçu le message.

mode bufferisé : comme standard, mais la terminaison est toujours indépendante de la réception correspondante, et le message peut être bufferisé pour s'en assurer. Le système de gestion des communications MPI effectue une copie du message. Cela permet à la tâche émetrice de continuer son traitement, en considérant son message comme transmis. C'est ce mode qui permet la synchronisation la plus lâche entre l'émetteur et le récepteur.

mode synchrone : comme standard, mais l'envoi ne se terminera pas tant que la réception n'est pas garantie.

mode ready : un envoi ne peut être démarré que si la réception correspondante a été initiée. Ce qui implique une synchronisation forte entre l'émetteur et le récepteur.

Chacun de ces quatre modes de communication existe en version bloquante et en version non bloquante. Cependant, dans tous les cas, le succès de la communication est garanti : aucun message ne peut se perdre entre l'émetteur et le récepteur. Tous ces modes de fonctionnement servent à assurer plus ou moins

strictement la synchronisation entre l'émission d'un message et sa réception.

Ce que l'on appelle ici une communication non bloquante est une communication desynchronisée de la tâche appelante. La communication non bloquante est exécutée par un autre processus, créé pour l'occasion. Cela ne présume pas de la synchronisation entre les processus qui effectuent le transfert des données.

L'usage de fonctions non bloquantes est conseillé pour masquer la latence des communications. Cela permet notamment d'effectuer un calcul *indépendant des données impliquées dans la communication* juste après l'appel à une fonction non bloquante. Ces primitives non bloquantes fournissent des mécanismes de synchronisation qu'il convient de tester pour s'assurer que le transfert a effectivement eu lieu.

Cette définition d'une fonction non bloquante n'est pas la même que celle que l'on a donnée lors de l'exposé de la problématique. Ici la communication est non bloquante car son exécution est déléguée à un sous-programme, qui s'exécute indépendamment du programme l'ayant appelé.

3.2.4 Les fonctions de communication multipoints

Les communications multipoints sont celles où l'on a plus d'un écrivain et un consommateur. Dans MPI, elles se séparent en trois sous-groupes : un écrivain et plusieurs récepteurs, plusieurs écrivains pour un récepteur unique, et enfin plusieurs écrivains vers plusieurs récepteurs.

La première communication multipoint est la communication BROADCAST. C'est une communication de type diffusion : Un producteur émet un message, et ce message est copié aux N consommateurs. Le modèle de cette communication est de 1 vers N ; et il n'y a qu'une seule donnée émise, copiée aux N tâches consommatrices. Si l'on veut envoyer N données distinctes aux N consommateurs, il faut utiliser N communications point à point.

A l'opposé, on peut trouver le modèle GATHER, où N producteurs envoient leurs messages vers un unique consommateur. C'est ce consommateur qui regroupe tous les messages. Il s'agit là d'une communication N vers 1, sans copie de données.

Il existe également des communications multi-écrivains, multi-lecteurs. Il en existe de deux types : les communications ALLGATHER, où N producteurs envoient chacun un message, et où les consommateurs recoivent chacun une copie de chaque message, soit N messages pour chaque consommateur. Ce type de communications est représenté sur la figure 3.4. On y voit que chaque tâche émetrice envoie un message, et chaque tâche réceptrice reçoit une copie de chaque message.

L'autre type de communications multipoints est le mode ALLTOALL. Dans ce modèle, chacun des N producteurs produits M messages Et chacun des M consom-

Processus	Messages
P0	A0
P1	A1
P2	A2
P3	A3

MPI_ALLGATHER
$\longrightarrow$

Processus	Messages
P0	A0 A1 A2 A3
P1	A0 A1 A2 A3
P2	A0 A1 A2 A3
P3	A0 A1 A2 A3

FIG. 3.4 – Modèle de communication MPI_ALL_GATHER où tous les producteurs émettent chacun un message, dupliqué à destination de tous les consommateurs

Processus	Messages
P0	A0 A1 A2 A3
P1	B0 B1 B2 B3
P2	C0 C1 C2 C3
P3	D0 D1 D2 D3

MPI_ALLTOALL
$\longrightarrow$

Processus	Messages
P0	A0 B0 C0 D0
P1	A1 B1 C1 D1
P2	A2 B2 C2 D2
P3	A3 B3 C3 D3

FIG. 3.5 – Modèle de communication MPI_ALL_TO_ALL où tous les producteurs émettent des messages à destination de tous les consommateurs

mateurs reçoit N messages. Chaque consommateur reçoit un message de chaque producteur, et chaque producteur a envoyé un message à destination de chaque consommateur. Ce type de communication est illustré sur la figure 3.5. Sur cette figure, on voit que chaque tâche productrice compose autant de messages qu'il y a de tâches consommatrices, et que ces messages sont distribués de façon à ce que chaque consommateur reçoive un message de chaque producteur. Ces communications multi-points couvrent un large éventail de possibilités. Cependant un point commun à tous ces types de communication est l'aspect statique de l'ordre de distribution des messages. Cet ordre est déterminé par le numéro de chaque tâche dans le communicateur et ne peut être modifié. Dans le cas qui nous intéresse, où le temps de traitement dépend de la donnée à traiter, cette particularité est un inconvénient. On souhaite plutôt avoir une distribution dynamique des données aux tâches, où une donnée est envoyée à une tâche parce que celle ci est disponible plutôt que parce que c'est son tour.

3.2.5 Conclusion

MPI offre une grande variété de modèles de communications, soit point à point, soit multi-points. Et les fonctions de communications définies permettent même d'effectuer des opérations de traitement sur les données qu'elles transportent. Certaines de ces communications ne sont d'aucune utilité pour le problème que l'on se pose. Il s'agit des communications avec duplication des données, telles que `MPI_ALLTOALL`.

Il existe de nombreuses implantations de MPI, dont les performances en terme de débit et de latence sont variables. Pour cette raison, il est difficile de donner une valeur de performance brute.

MPI est destiné à fonctionner sur des machines multi-processeurs à mémoire distribuée, ce qui le distingue sensiblement des architectures multi processeurs embarquées sur puce, à mémoire partagée, qui sont l'architecture cible de nos travaux. Pour cette raison, certains des choix faits pour MPI ne conviennent pas pour les communications embarquées. Le choix d'un fonctionnement synchrone par exemple est handicapant puisqu'il force la tâche émettrice à attendre que la tâche destinataire du message accuse réception de celui-ci avant de pouvoir continuer. MPI propose des primitives de communication non bloquantes, mais ce terme sert à masquer l'appel à un sous-programme, qui effectue la communication de façon bloquante. Une communication non bloquante est une communication postée, qui utilise un mécanisme de synchronisation pour signaler son accomplissement. Ce type de communication est utilisé pour masquer la latence dûe à la communication elle même. Dans ce type de communication, la communication proprement dite reste bloquante pour chacun des sous-programmes qui y prend part. Ce type de fonctionnement est possible sur une station de travail où il est possible de créer un sous-programme dont l'exécution est différée. Ce n'est pas envisageable dans le cadre de systèmes embarqués.

Un autre inconvénient de MPI est l'aspect statique des communications. Chaque message émis par un producteur est destiné à un consommateur, ou à un groupe de consommateurs, mais ce ou ces consommateurs sont déterminés dès l'écriture de la tâche, et il ne peut pas y avoir d'adaptation dynamique des communications en fonction de la charge de telle ou telle tâche.

3.3 StreamIt

StreamIt est un langage de description de flux qui permet d'écrire des applications parallèles d'une façon indépendante de l'architecture visée. Le but de ce langage de description est d'aider le développeur d'application à extraire le parallélisme de son application et d'exposer la topologie des communications de l'application. les spécifications de ce langage sont décrites dans [TKA02].

Ce langage de description est utilisé pour décrire une application de décodage MPEG-2 dans [DHRA06].

Le domaine d'applications couvert par StreamIt est vaste : il s'agit de toutes les applications qui traitent de flux de données. Il peut s'agir d'applications multimédia comme le décodage d'un flux vidéo, ou d'applications réseau comme le routage de paquets IP. Pour les auteurs, les applications candidates partagent les caractéristiques suivantes :

- Elles traitent de grands flux de données, composés d'items de données plus ou moins indépendants les uns des autres. Chaque item est traité pendant un certain temps, avant de laisser sa place à l'item suivant.
- Les filtres de l'application sont indépendants les uns des autres, sans références à des variables globales ou à d'autres filtres. Chaque étape du traitement global d'un item de données est appelé un filtre. Les filtres sont l'équivalent de nos tâches. Ces filtres, sont connectés ensemble par des canaux par lesquels ils effectuent leurs communications. L'usage de variables globales est proscrit.
- Le graphe de tâches est défini de façon statique et il reste généralement inchangé pendant le temps de l'exécution de l'application.
- Bien que les modifications de la structure du graphe soient rares, elles ne sont pas interdites. Aussi, le graphe peut-il être reconfiguré dynamiquemment à une faible fréquence. Un exemple de variation est, dans un réseau sans fil, l'ajout d'un filtre pour réagir à un haut niveau de bruit sur une interface d'entrée.
- Il peut y avoir de temps à autre des communications hors flux entre des tâches du graphe. Par exemple l'appui sur une touche par l'utilisateur d'un téléphone sans fil, l'émission d'un message d'erreur.
- Ces applications ont de fortes contraintes de performances, qui peuvent être assorties de contraintes d'exécution en temps réel. De plus ces applications sont souvent destinées à être embarquées. S'ajoutent alors des problèmes de consommation électrique.

Les 6 points énoncés ci-dessus constituent les caractéristiques principales d'une application candidate à un déploiement selon la méthode proposée par StreamIt.

L'élément standart de StreamIt est le Filtre. Un Filtre produit un traitement élémentaire sur un item de données. Un Filtre est donc caractérisé par sa fonction *work* qui défini le traitement, et par ses entrées/sorties. Ses communications s'effectuent par les canaux *input* et *output*. Il s'agit de canaux FIFO qui supportent trois type d'opération : `pop()` qui supprime le dernier élément du canal, et renvoie sa valeur. `push(x)` qui ajoute l'élément `x` en tête du canal et `peek(i)`, qui renvoie la valeur du `i`ème élément du canal, sans supprimer cet élément.

Une restriction de StreamIt est qu'il oblige tous les Filtres à avoir des débits de lecture et d'écriture constants. Ces débits sont définis lors de l'initialisation de chaque Filtre et ne peuvent être modifiés. Cette limitation devrait être supprimée dans une prochaine version.

Les graphes de tâches construits avec StreamIt sont composés de trois types de blocs : *Pipeline*, *SplitJoin* et *FeedbackLoop*. Chacune de ces trois structures prédéfinit une façon d'interconnecter les Filtres entre eux. Chacun de ces blocs a une entrée et une sortie. Un programme StreamIt est une composition hiérarchique de ces blocs. Ces trois types de structures sont représentées sur la figure 3.6.

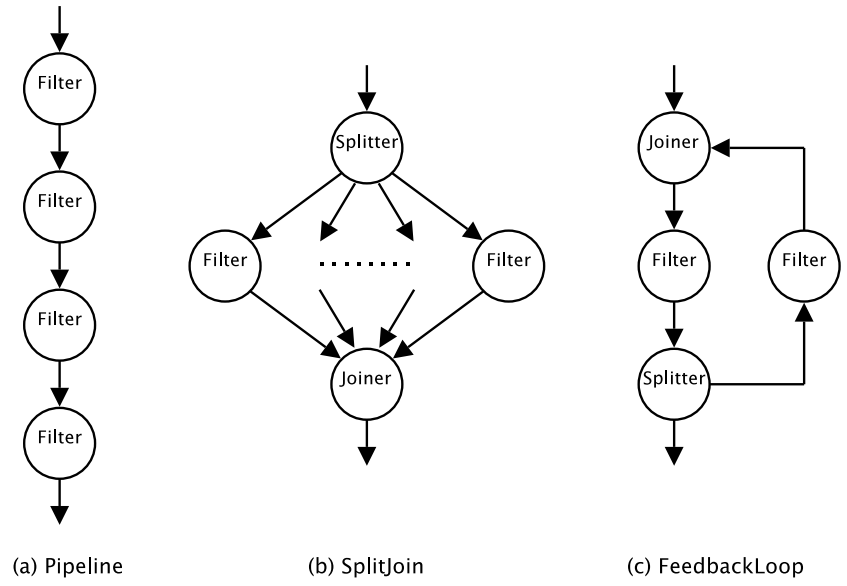

FIG. 3.6 – Les trois structures proposées par StreamIt.

La structure *pipeline* ne requiert pas d'explications particulières, il s'agit d'une suite ordonnées de Filtres où la sortie du Filtre i est l'entrée du Filtre $i+1$.

La structure *SplitJoin* spécifie des flux parallèles à partir d'une entrée unique. Le point d'entrée est un élément *splitter* qui distribue les données aux filtres auxquels il est connecté. Il existe trois façons de répartir ces données :

1) par duplication, toutes les données sont envoyées à toutes les tâches.

2) Round Robin$(i_1, i_2, ...i_n)$ où chaque tâche reçoit à son tour k élément de données, en fonction de son débit entrant.

3) Null, aucune tâche ne reçoit aucune donnée en entrée.

De le même façon, il existe un *joiner* qui collecte les données aux sorties des Filtres. Il peut entrelacer les flux sortant de deux façons : Round Robin et Null.

La structure *FeedbackLoop* permet de construire des cycles dans le flux de données. Cette structure sert par exemple à implémenter des boucles réccursives.

En plus de ces canaux de communications à haute bande passante, StreamIt fournit des canaux à faible bande passante. Ces canaux sont destinés à implémenter les communications de contrôle entre des Filtres. Cette deuxième famille de canaux comprend des canaux point à point, entre deux Filtres, et des canaux à large diffusion, où une tâche envoie un message à tout un sous ensemble de tâches.

3.4 YAPI

Yapi [dKSvdW+00] est une extension du modèle des réseaux de Kahn. C'est un formalisme basé sur des communications FIFO entre des tâches productrices et des tâches consommatrices. Les canaux de comunication sont des canaux FIFO à sens unique. Chacun a un producteur et un consommateur. Les primitives pour accéder à ces canaux sont `read()`, `write(x)` et `select().Read()` renvoie la valeur du dernier élément dans la FIFO et l'y supprime, `write(x)` écrit la valeur `x` dans la première case libre de la FIFO. `Select()` sert à sélectionner un canal parmi plusieurs canaux d'entrée ou de sortie, succeptibles de fournir une donnée ou d'en consommer une. L'ajout de cette fonction `select()` fait disparaitre le déterminisme des réseaux de Kahn, puisqu'on peut choisir le canal où consommer. Le taux de remplissage des FIFO dépend de la vitesse d'exécution des threads. En contrepartie, cela permet de prendre en compte des évènements non déterministes tels que l'intervention de l'utilisateur.

L'algorithme régissant le choix d'un canal implanté dans la fonction `select` n'est pas spécifié.

3.5 COSY

COSY [BKK+00] est une implémentation de YAPI.

COSY propose trois niveaux d'abstractions des communications : application (FIFO), system (canaux dpn), Logique (VCI) et physique.

Les canaux de communication au niveau applicatif sont des canaux FIFO, point à point entre un producteur et un consommateur. Ces canaux supportent les primitives `read`, `write` et `select`. Les fonctions `read` et `write` sont bloquantes, c'est à dire qu'elle se terminent lorsqu'elles on réussi à lire, respectivement écrire la quantité de données demandée. La fonction `select` sert à selectionner un canal parmi une liste pour y lire ou y écrire des données. Elle est également bloquante, et elle retourne dès qu'une des FIFO de la liste peut fournir le service demandé. L'utilisation des fonctions `read` et `write` avec ces canaux de communication permet d'implémenter des réseaux de Kahn. Lorsqu'on utilise la fonction `select`, on sort de ce contexte car on introduit un indéterminisme : le statut des canaux dépend de la vitesse d'exécution des tâches.

L'implémentation sépare nettement les canaux de communication des tâches. Ces derniers effectuent leurs communications en utilisant les primitives `read`, `write`ou `select`, mais n'ont pas de connaissance du fonctionnement du canal. De même, les FIFO n'ont aucune connaissance des tâches qu'elles servent.

Lors de l'initialisation du système l'utilisateur doit spécifier les tailles des canaux qu'il utilise. Une attention particulière doit être donnée à cette étape, car de

trop petits canaux peuvent conduire à un interbloquage des tâches, tandis que de grands canaux se revèlent trop chers à implémenter dans la mémoire embarquée sur puce.

3.6 SPADE

SPADE [vdWLG+99] est aussi une implémentation de YAPI. SPADE se présente comme une méthodologie pour effectuer le déploiement d'une application particulière sur une plateforme matérielle paramétrable.

SPADE repose sur une parallélisation de l'application candidate sous la forme d'un graphe de Kahn. Les communications se font par FIFO, et les lectures dans ces FIFO sont bloquantes. Les FIFO ont une profondeur infinie, ce qui exclut le risque d'interblocage, mais pose un problème pour une implantation réelle des canaux de communication.

La modélisation et la simulation se font à un niveau "système", où les processeurs et les processus qu'ils exécutent sont mélangés, et où les canaux de communications ne distinguent pas nettement le code exécuté du média (bus ou autre) employé.

Le "mapping" concerne les tâches, mais pas les canaux de communication.

L'analyse de performances se fonde sur la simulation combinée du modèle de l'application et du modèle de l'architecture. Le modèle d'architecture est composé de blocs non fonctionnels. Et la simulation d'architecture utilise les résultats de l'exécution du modèle d'application. ces résultats sont fournis sous forme de fichier de trace. L'analyse de ces fichiers permet de paramètrer les modèles non fonctionnels afin de leur faire adopter le comportement des tâches matérielles (principalemment la latence entre les communications et le volume de celles ci). La modélisation des communications prend en compte à la fois le type de communication (des canaux FIFOs) et le support sur lequel elles ont lieu (le bus système).

3.7 Disydent

Disydent (Digital System Design Environment) [APDG05] est un environnement de développement de systèmes numériques conçu au laboratoire Lip6. Il permet de réaliser l'implantation d'une application logicielle multi-tâches écrite en C sur une architecture matérielle multi-processeur intégrée.

3.7.1 Principes

Un problème de conception y est décrit sous la forme d'un triplet : (*système, application, contraintes*). Le **système** est un ensemble composé d'une plateforme matérielle générique et d'un micro-noyau minimal. On peut ajouter des coprocesseurs à cette plateforme matérielle, ou y faire varier le nombre de processeurs. L'application candidate est décrite sous la forme d'un graphe de tâches respectant le formalisme des réseaux de Kahn. Ces tâches sont écrites en langage C, sous la forme de threads POSIX [Tan95]. Les contraintes concernent la latence et le débit de l'application (par exemple un certain nombre d'images / seconde pour une application de décodage M-JPEG) ou la surface de silicium utilisée.

C'est un environnement à base de plateforme, en ce sens que les grandes lignes de l'architecture matérielle sont déjà fixées avant le début de la conception. La plateforme matérielle cible est une plateforme multi-processeur, disposant de plusieurs bancs de mémoire connectés ensemble via le bus système PI-Bus [Ope96]. On peut adjoindre à ce système un ou plusieurs accélérateurs matériels (des coprocesseurs). Les tâches communiquent entre elles par le biais de canaux fifo. Le graphe de tâches est explicitement écrit en C dans la fonction `main`. Le fait d'utiliser le langage C et la norme POSIX pour écrire le graphe de tâches permet de mettre l'application au point en la faisant fonctionner sur la machine hôte, ce qui offre une vitesse d'exécution appréciable.

Le système d'exloitation embarqué est le micro-noyau Mutek [PGH03] qui fournit une implémentation des threads POSIX et des mécanismes de synchronisation (Sémaphores, Mutex).

3.7.2 Flot de conception

On a vu que le problème du déploiement d'une application sur un système donné se présentait sous la forme d'un triplet *(système, application, contraintes)*. Dans ce triplet, l'application est écrite sous forme d'un graphe de tâches communicantes. La première étape de la conception consiste à déployer ce graphe de tâches sur la plateforme matérielle cible. Sur cette plateforme, on mesure les performances atteintes, en terme de latence et de débit. Lorsque les contraintes sont respectées, la conception s'arrête.

Si les contraintes de performances ne sont pas respectées, il est possible d'ajouter des processeurs ou des accélérateurs matériels (des coprocesseurs spécialisés) au système initial.

Pour concevoir des accélérateurs matériels, il faut d'abord sélectionner des tâches de l'application qui seront de bonnes candidates à une implantation en matériel. Cette sélection repose sur le profilage de l'application.

En vue de la synthèse sur matériel, la tâche est reécrite dans un sous- ensemble

du langage C, dans lequel sont notamment proscrits les pointeurs, et on utilise l'outil de synthèse d'architecture UGH.

3.7.3 Canaux de communication DPN

La partie de Disydent à laquelle on s'intéresse particulièrement est la bibliothèque DPN. Cette bibliothèque est spécialement destinée à implémenter des applications parallèles multi-tâches respectant la sémantique des réseaux de Kahn, et a bénéficié d'optimisations dans ce sens. Chaque canal de communication DPN est explicitement connecté à un unique producteur et à un unique consommateur. Les deux primitives de communication utilisent chacune trois arguments : le premier est un identifiant du canal DPN, le second est un pointeur sur un tampon en mémoire locale, et le troisième représente la quantité de données à transférer. L'algorithme utilisé par ces primitives de communication, en lecture comme en écriture est le suivant :

```
ChannelTransfert(Channel *c, void *buf, unsigned long n)
```

- **tant que** (n)
  - m = min (n, c->contenu)
  - transfert (buf, c->data, m) */* Lecture ou écriture */*
  - buf+=m
  - c->pointeur = (c->pointeur + m) % c->profondeur
  - n-=m
  - pthread_mutex_lock(c->mutex) ;
  - c->contenu -= m ;
  - pthread_cond_signal(c->cond) ;
  - **Si** (c->status == 0 && n != 0) alors
    - pthread_cond_wait(c->cond, c->mutex) ;
  - **Finsi**
  - pthread_mutex_unlock(c->mutex) ;
- **fin tant que**

Dans cet algorithme, les ressources préfixées en **c->** sont des champs de la structure *fifodpn*. Chaque canal utilise deux mécanismes de synchronisation : un `mutex` qui est utilisé pour garantir l'accès aux ressources partagées en exclusion mutuelle. Ici les ressources partagées se résument à une variable : le contenu du canal, c'est à dire le nombre d'éléments qu'il contient. Le second mécanisme de synchronisation est une `condition`. Cette condition sert à chacune des deux tâches productrice ou consommatrice à notifier à l'autre qu'elle a effectué une modification sur le contenu du canal. La présence d'un mécanisme capable de suspendre la tâche appelante est rendue nécessaire par la sémantique des primitives de communication employées : lectures et écritures sont bloquantes, et ne se

finissent qu'une fois que la totalité du transfert demandé a été effectuée. L'utilisation de la condition permet d'endormir une tâche si elle ne peut pas effectuer son transfert complètement. Elle est alors réveillée lorsque l'autre tâche connectée au canal y effectue un accès. On évite ainsi une attente active sur une ressource du canal fifo. Dans le cas d'une communication effectuée par un module matériel, une interruption est envoyée à la fin du transfert. Le traitement de cette interruption consiste pour le système à réveiller la tâche logicielle en attente sur ce canal.

3.7.4 Conclusion et limites

L'environnement Disydent présente des avantages certains. D'une part le choix des graphes de Kahn comme ensemble des applications candidates permet de garantir que le comportement global de l'application est indépendant de la vitesse d'exécution des différentes tâches. Ceci permet de valider fonctionnellement l'application sur une station de travail où l'on contrôle mal l'ordonnancement des tâches.

Viennent ensuite les canaux de communication DPN. Leur implémentation est optimisée pour l'utilisation dans le cadre de graphes de Kahn. On verra qu'il s'agit là du principal but de ce travail que de définir et d'implémenter de nouveaux canaux de communications.

Le travail présenté dans cette thèse constitue essentiellement en une extension de Disydent. Il s'agit principalement de définir de nouvelles fonctions de communications et de les implémenter, mais également de faire évoluer le système matériel vers une architecture supportant un grand nombre de processeurs.

3.8 Conlusion

Il existe une grande variété d'implantation des communications entre les tâches. Une part importante de celles étudiées ici implémentent les réseaux de Kahn. Ces réseaux présentent des caractéristiques intéressantes, principalement le déterminisme. Néanmoins, ce formalisme n'est pas adapté aux applications télécom, et on ne souhaite pas s'y cantonner. Cependant, afin de pouvoir couvrir un large spectre d'applications, on essaiera d'englober les réseaux de Kahn dans quelque chose de plus vaste et de moins contraignant.

En ce qui concerne les mécanismes de communication plus particulièrement destinés aux applications embarquées, ils sont la plupart du temps présentés comme faisant partie d'un environnement de développement conjoint logiciel/matériel. Cette caractéristique rend difficile la généralisation des résultats obtenus lorsque les communications entre tâches recouvrent à la fois les échanges de données entre

les tâches logicielles et le transfert de ces données sur le lien physique d'interconnexion (bus ou autre).

D'un autre coté, des systèmes de communication entre processus distants tels que MPI permettent ce type de communications multipoints. Il s'agit cependant d'une bibliothèque destinée à assurer les communications entre des processsus fonctionnant sur des machines distinctes. Par conséquent, les contraintes sont différentes et les processus échangent de gros paquets de données (plusieurs dizaines de Ko) à des fréquences faibles. Une autre limitation de MPI concerne l'aspect déterministe de ces communications multipoints. Toutes permettent de connaitre la tâche destinataire d'un paquet de données particulier en fonction du numéro de celui-ci dans la file des données émises. Les données sont distribuées aux tâches réceptrices en fonction de l'ordre dans lequel elles ont été envoyées par les tâches émetrices plutôt qu'en fonction de la disponibilité des tâches réceptrices.

Ces différents points, la spécificité des moyens de communications sur puce, dédiés à un environnement de conception particulier, et la non adaptabilité aux communications sur puce d'une bibliothèque telle que MPI nous ont conduit à définir un nouveau protocole de communication, destiné à des tâches s'exécutant sur la même puce, et qui permette une plus grande souplesse quand au nombre de tâches qui les utilisent.

L'environnement Disydent, développé au laboratoire LIP6, constitue la base de ce travail. Aussi cherchons-nous à construire un protocole de communication multi-points qui soit une évolution des canaux de communication DPN de Disydent.

Chapitre 4

Les spécifications de l'application

Sommaire

La question à laquelle on cherche à répondre est : "comment tirer parti efficacement du parallélisme offert par les systèmes multi-processeurs embarqués sur puce."

La réponse consiste à séparer l'application candidate en un certain nombre de tâches qui s'exécutent indépendamment les unes des autres. Cette étape s'appelle la parallélisation, et elle est l'objet du présent chapitre.

4.1 Les différents types de parallélisme

Mettre en évidence le parallélisme gros grain d'une application, ou extraire ce parallélisme est une opération non triviale qui nécessite une bonne connaissance

de l'application candidate. Par conséquent, nous croyons que le concepteur de l'application est la personne la mieux placée pour en extraire le parallélisme.

Pour extraire le parallélisme gros grain d'une application, il y a deux méthodes possibles.

4.1.1 Le parallélisme pipeline

La première méthode repose sur la segmentation en gros grains dans le code de l'application séquentielle. L'algorithme est séparé en tâches fonctionnelles qui s'exécutent en séquence. Toute les tâches sont connectées entre elles, comme sur la figure 4.1. Chacune effectue une part du traitement et fournit sur sa sortie un résultat intermédiaire du traitement. Ce résultat intermédiaire est la donnée d'entrée de l'étage suivant du pipeline.

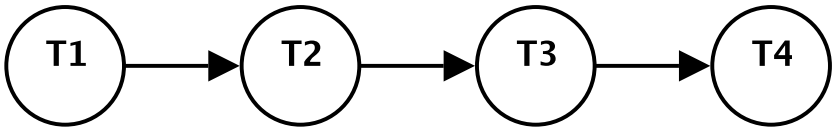

FIG. 4.1 – Exemple de parallélisme de type pipeline

Le traitement complet est effectué en passant les données à travers tous les étages du pipeline. Dès que l'étage N du pipeline a envoyé son résultat à l'étage N+1, il est prêt à traiter une nouvelle donnée. Par exemple, sur la figure 4.1, la tâche T4 travaille sur la donnée D, pendant que T3 travaille sur la donnée $D+1$, T2 sur la donnée $D+2$ et T1 sur $D+3$. On a 4 données en cours de traitement, à différentes étapes de ce traitement.

Ce type de structure est beaucoup utilisé dans les applications de traitement vidéo. Un point important de ce type de parallélisme est que toutes les tâches doivent avoir grosso modo la même durée, faute de quoi la tâche la plus longue deviendra le goulot d'étranglement du système.

Une caractéristique de ce type de parallélisme est que l'on produit un graphe de tâches qui ne comporte qu'un seul chemin. De la sorte, on peut garantir l'ordre de traitement des données. C'est un point essentiel pour les applications multimédia où les données doivent être émises dans l'ordre où elles sont entrées. Cette caractéristique est importante dès lors que l'on a des dépendances dans le flux de données.

4.1.2 Le parallélisme task farm

La seconde approche ne cherche pas à modifier le corps de l'application, mais plutôt à créer plusieurs clônes de celle-ci. Tous ces clônes travaillent en même

temps, sur des données différentes. Ce type de parallélisme est connu sous le nom de "task farm" (ou "orphaning" pour OpenMP). Il est illustré par la figure 4.2. Sur cette figure, on suppose que toutes les tâches effectuent le même traitement.

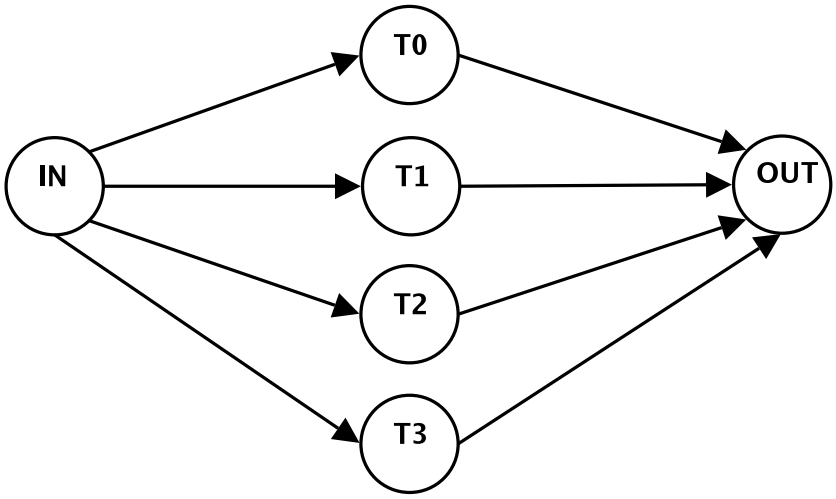

FIG. 4.2 – Exemple de parallélisme de type task farm

L'avantage d'un tel type de parallélisme est que, pour le traitement d'un paquet de données, les communications entre les tâches sont aussi réduites que possible et n'occupent qu'une faible part du temps de traitement total de la donnée. De plus, ce type de structure est bien adapté lorsque le temps de traitement dépend des données, ce qui est typiquement le cas des applications télécom. Un paquet qui nécessite un temps de traitement important ne ralentit pas tout le sytème : une seule tâche de traitement s'en trouve affectée.

L'inconvénient corollaire est que l'on ne peut plus garantir l'ordre de sortie des données. Le même paquet qui nécessite un traitement long va être doublé par d'autres paquets, au temps de traitement plus court, et qui sont passés par d'autres instances de la tâche. Si l'ordre de sortie des données doit absolument être maintenu, il faut numéroter les paquets à leur entrée dans la partie comportant du parallélisme task farm, et utiliser ce marquage pour les réordonner à leur sortie.

Pour ces différentes raisons, le parallélisme task farm est bien adapté au traitement de données structurées en paquets indépendants, ce qui est typiquement le cas des flux Ethernet Gigabit. Il est même particulièrement bien adapté aux applications dont le rôle est de multiplexer/démultiplexer des flux de paquets : les applications de routage, de classification et de qualité de service principalement.

Le parallélisme task farm est également plus adapté pour exploiter le parallélisme d'une machine disposant de nombreux processeurs : le parallélisme pipeline exige la répartition de l'algorithme en un certain nombre de sous-tâches, chaque

processeur en exécutant une pour un rendement optimum. Si c'est envisageable pour une machine à 2, 3, 4, 10 processeurs, ça devient irréaliste lorsqu'on a affaire à des systèmes disposant de plusieurs dizaines de processeurs. Il faut alors utiliser plusieurs clones, ou instances, de la même tâche.

Dans la littérature, et particulièrement pour MPI et OpenMP, il existe un troisième type de parallélisme. Celui-ci consiste à séparer une boucle d'exécution de type *while* ou *for* en plusieurs sous-ensembles, chacun d'eux étant traité par une tâche distincte. Les résultats sont compilés en fin de traitement. Ce type de parallélisme est décrit comme le parallélisme au niveau boucle (looplevel parallelism). Dans le contexte des applications embarquées qui nous interesse, ce découpage de boucle en sous-ensembles est effectué de préférence statiquement lors de la compilation (à defaut de connaitre le nombre d'itérations, on connait le nombre de sous-ensembles (tâches) entre lesquelles partager le travail). Cela se rapporte donc plus ou moins au parallélisme task farm, puisque c'est le même code qui est exécuté par plusieurs tâches.

4.1.3 Les formes hybrides de parallélisme

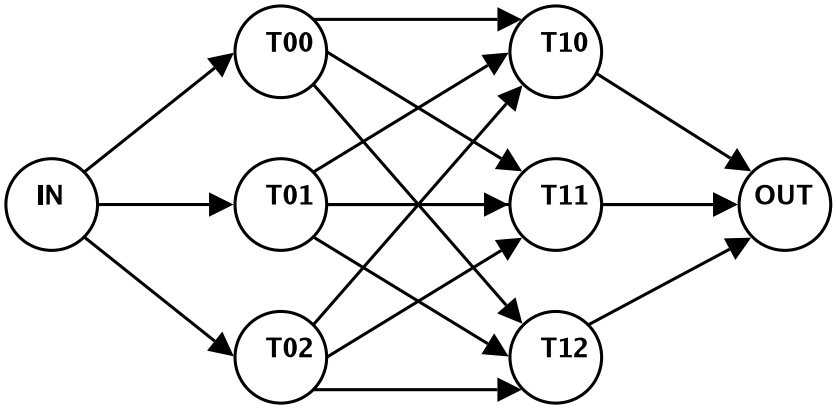

FIG. 4.3 – Exemple d'un parallélisme hybride entre pipeline et task farm : un pipeline à deux étages. Chaque étage a trois instances.

Task farm et pipeline peuvent être combinés pour donner naissance à n'importe quel type d'hybride, comme illustré sur les figures 4.3 et 4.4. Ce moyen permet d'accélérer une partie d'un pipeline qui se prête mal à une séparation en plusieurs sous-tâches. Sur la figure 4.3, on a un pipeline à deux étages, où chaque étage est instancié 3 fois, ce qui donne 6 tâches. Sur la figure 4.4 on a un pipeline à 4 étages, avec un nombre d'instances différent pour chaque étage.

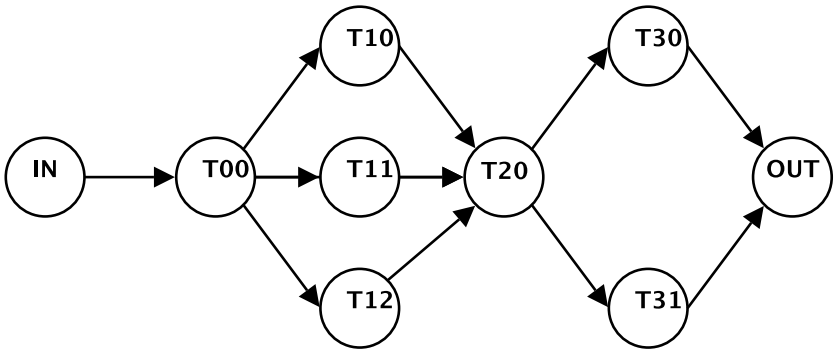

FIG. 4.4 – Exemple d'un parallélisme hybride entre pipeline et task farm : un pipeline à 4 étages dont les étages 1 et 3 sont instanciés respectivement 3 et 2 fois.

4.1.4 Communications synchrones ou asynchrones

La question de la synchronisation entre les tâches émétrices et réceptrices n'a pas encore été abordée. Par communication synchrone, on entend que la tâche émétrice reste bloquée jusqu'à ce qu'une tâche consommatrice lui confirme la consommation du message. De ce fait on est sûr que le message ne se perd pas en chemin. C'est intéressant dans le cadre d'échanges de messages entre plusieurs tâches, la confirmation de réception d'une communication assure à la tâche émétrice que le message est parti, et que la mémoire utilisée pour le stocker avant envoi est à nouveau disponible. Cela permet également, dans le cadre d'applications réparties sur des nœuds de calcul de s'assurer que la tâche réceptrice est accessible.

Cependant, notre contexte est différent, puisqu'il s'agit d'applications embarquées sur puce, où le nombre de processeurs disponibles est environ le même que le nombre de tâches à exécuter. Dans ce cas on ne souhaite pas synchroniser les tâches entre elles plus que nécessaire (synchroniser les tâches signifie en pratique attendre la plus lente).

4.2 graphe bi-partite

Le point commun entre ces schémas est que toutes les communications sont effectuées point à point, entre un émetteur et un récepteur. Ceci est en grande partie dû au fait que l'on utilise un graphe orienté pour représenter les communications entre les tâches. Ce type de représentation est commode mais il force la représentation des communications sous la forme de liens point à point, entre deux noeuds du graphe. Si ce mode de représentation ne pose pas de problème particulier dans le cas du parallélisme pipeline, où toutes les communications ont

lieu entre un émeteur et un récepteur, dans le cas du task farm, les choses sont un peu différentes. Dans la figure 4.2, les quatre tâches T0, T1, T2 et T3 sont quatre instances d'une même tâche de traitement. La tâche IN envoie ses données indistinctement à l'une ou l'autre de ces tâches. On a donc finalement 4 canaux de communications distincts pour réaliser les communications entre L'entrée IN et la tâche T. Ces 4 canaux partagent tous les mêmes caractéristiques : même tâche émétrice , même tâche réceptrice (modulo le numéro d'instance), et même type de donnée véhiculé (les quatre instances de tâches sont indifférenciées, les données qu'elles se communiquent le sont donc aussi dans les 4 canaux utilisés. Si l'on veut avoir des types de données différents identifiés par les canaux utilisés, il faut doubler les voies de communication : 8 canaux).

Or, si le schéma du graphe de communication nous impose 4 canaux de communication point à point entre toutes ces tâches, ce n'est pas nécessairement ce que l'on souhaite implémenter.

4.2.1 La factorisation des canaux de communication

Dans de nombreux cas, illustrés notamment par les figures 4.2 et 4.3, les données émises par une tâche ne sont pas destinées à une tâche en particulier, mais plutôt à une classe de tâches. Supposons que T01, T02 et T03 soient trois instances d'une même tâche. De même T11, T12 et T13, trois instances d'une autre tâche. Dans ce cas, les trois premières tâches envoient leurs sorties vers l'une des trois tâches de l'étage suivant. Dans cet exemple, on considère que toutes les données émises par les tâches T0x ont la même priorité. Dans cette situation, il est peut-être judicieux de remplacer les 9 canaux séparés par un seul commun à tous les émetteurs et tous les récepteurs. Ceci introduit la notion de factorisation des communications, où une seule fifo peut être lue et remplie par un nombre quelconque de producteurs et de consommateurs. Ce type de graphe met en évidence la notion de partage des canaux de communication. Dans ce cas, les canaux de communication doivent être explicitement représentés sur le graphe de tâches. La figure 4.5 montre la même application que la figure 4.3 en introduisant explicitement les canaux de communication. Il y a une fifo connectée à trois producteurs et trois consommateurs. On peut remarquer que les communications avec les tâches IN et OUT peuvent être factorisées de la même façon. De même la figure 4.6 représente la même application que la figure 4.4 avec ses canaux de communications factorisés.

4.2.2 Nœud tâche et nœud fifo

Le fait d'introduire des canaux de communications supportant un nombre quelconque de producteurs et de consommateurs nous amène à devoir les repré-

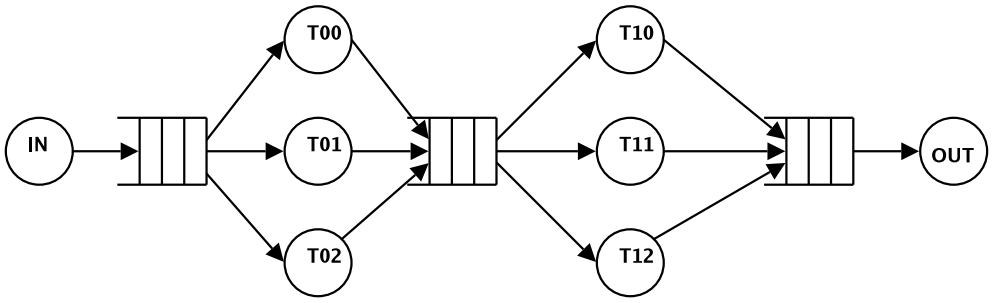

FIG. 4.5 – Nouveau graphe de l'application parallèle à deux étages de pipeline, avec mise en évidence explicite des communications

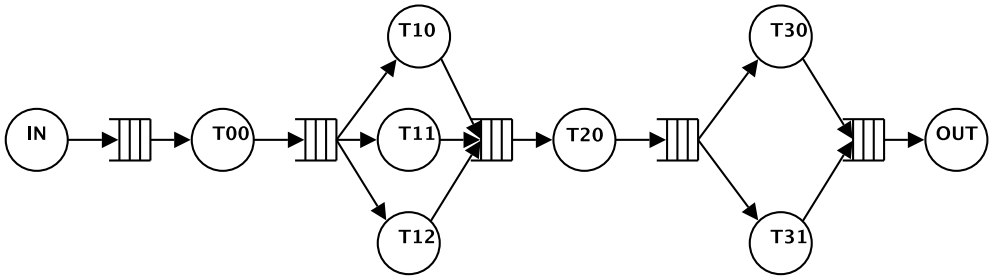

FIG. 4.6 – Nouveau graphe de l'application parallèle à quatre étages de pipeline, avec mise en évidence explicite des communications

senter explicitement sur le graphe de tâches. On passe donc d'un graphe de tâches communicant entre elles à un graphe où des tâches et des canaux de communication cohabitent.

Ce nouveau type de graphe est formellement différent du précédant dans la mesure où il s'agit maintenant d'un graphe bi partite. Il y a maintenant deux types de noeuds : les noeuds représentant des tâches et des noeuds représentant des fifos.

Les arcs vont nécessairement d'un type de noeud à l'autre. Ils sont donc également de deux types : d'une tâche vers une fifo, ou d'une fifo vers une tâche. Avec pour sémantiques respectives que la tâche écrit dans la fifo, où que la tâche lit dans la fifo.

4.3 Exemple d'application parallèle : l'application classification / ordonnancement

Ce paragraphe décrit une application de classification/ordonnancement de paquets IP qui utilise le parallélisme hybride que l'on a évoqué. Cette application utilise également les canaux de communication multipoints que l'on vient de spé-

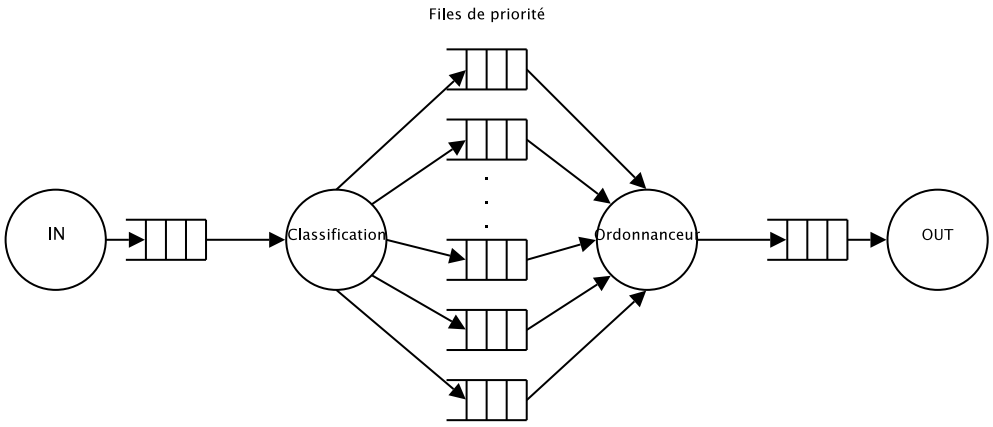

FIG. 4.7 – Le graphe de tâches de l'application générique, Le parallélisme pipeline y est mis en évidence.

cifier. Elle est donc un exemple d'application parallèle utilisant ce type de canaux. Elle est utilisée au paragraphe 7.3 pour valider le fonctionnement et les performances des canaux en situation d'utilisation réelle ; ainsi que pour mesurer l'impact en terme de performance du placement de ces objets logiciels sur les ressources matérielles.

On décrit ici ses spécifications. Les détails concernant l'implémentation sont donnés au paragrahe 7.1.

L'application de classification est une application réseau dont le but est de classer les paquets IP arrivant sur sa source d'entrée, puis d'ordonner leurs réémission sur ses interfaces de sorties. Pour notre cas, on se limite à deux ports d'entrée et deux ports de sortie.

Cette application comporte donc deux tâches distinctes. La première est la tâche de classification, qui répartit les paquets entrant dans un certain nombre de fifo de priorités différentes (appelées files de priorités par la suite) en fonction du flux auquel chacun d'eux appartient.

La seconde tâche est la tâche d'ordonnancement, qui extrait les paquets des différentes fifos en fonction de leur priorité.

Les fifos de priorités sont le résultat de contrats passés entre l'application et ses utilisateurs : chaque contrat garantit au flux de paquets concerné qu'il dispose d'une certaine part de la bande passante totale offerte par le système. Cette part a été négociée préalablement et est fixe pendant la durée d'exécution de l'application. Dans notre cas, ces flux sont identifiés par les adresses IP de destination des paquets.

Conceptuellement, on voit bien que cette application comporte deux parties, et que ces deux parties, la classification et l'ordonnancement communiquent par l'intermédiaire des fifos de priorité. C'est cette application, implantée sous la forme

de deux tâches travaillant en pipeline qui est représenté sur la figure 4.7. Sur cette figure on ne représente que 5 fifos de priorité. Le but de cette réduction est de faciliter la lecture du schéma. Dans la version que nous utiliserons dans nos expérimentations, il y a 12 fifos de priorité.

On se rendra également compte que ce découpage en deux tâches est insuffisant pour traiter les paquets à un débit acceptable. On sera donc amené à introduire du parallélisme *taskfarm* dans cette application, ce qui nous conduira au graphe de tâches illustré par la figure 4.8. Sur cette figure on n'a également représenté que 5 fifos de priorité pour des raisons de clarté. De même, on ne représente que 4 tâches de classification/routage et 4 tâches d'ordonnancement, alors que les expérimentations de déploiement sur une architecture matérielle multi-processeurs conduites au paragraphe 7.3 montreront que l'on instancie un nombre de tâches bien plus élevé.

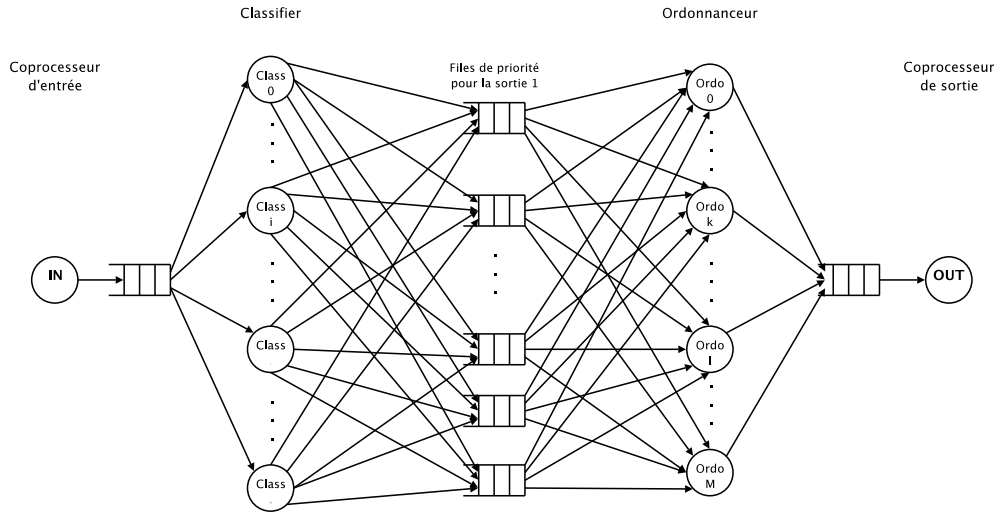

FIG. 4.8 – Le graphe de tâches de l'application, utilisant le parallélisme task farm

4.3.1 La tâche de classification

Le but de cette tâche est de déterminer dans quel flux classer chacun des paquets qui entre par l'interface d'entrée.

La tâche de classification détermine à quel flux appartient un paquet IP. Tous les paquets appartenant à un même flux partagent certaines caractéristiques communes. Cela peut être l'adresse source du paquet, son adresse de destination, ou encore le type de protocole utilisé. Dans notre cas, on distingue les flux sur le seul critère de l'adresse IP de destination de chaque paquet. On distingue donc 12 flux que l'on classe dans 12 fifos de priorité. En fait, seuls 11 flux sont clairement

identifiés comme faisant l'objet d'un contrat de qualité de service, le douzième est le flux par défaut, dans lequel on copie tous les paquets que l'on n'a pu rattacher à aucun des autres flux. La classification consiste, pour chaque paquet, à déterminer dans quelle fifo le copier.

En plus de cette fonction de classification, cette tâche doit également vérifier que chacun des paquets qui transite est bien valide ; ceci pour éviter de propager sur le réseau des paquets dont on sait qu'ils sont corrompus. L'invalidité du paquet peut avoir plusieurs causes que l'on classe en deux familles en fonction de la couche réseau dans laquelle l'erreur est détectée : soit c'est un champ de l'en-tête IP qui indique que le paquet ne doit pas être transmis parce que ne respectant pas certaines conditions, l'erreur se situe alors au niveau IP et donne le plus souvent lieu à l'émission d'un message d'erreur à destination de l'émetteur du paquet. Soit l'erreur a été commise au niveau de la couche transport du réseau (Ethernet dans notre cas). Ce type d'erreur est détecté par la conformité des bits de validité lus avec ceux que l'on calcule. Dans ce cas, le paquet est considéré comme intégralement corrompu, et il est détruit, sans émission de message d'erreur. Il existe de nombreuses raisons de détruire un paquet, mais elles ont toutes une fréquence très faible, et la quasi totalité du flux entrant est acheminée vers les fifo de priorité.

4.3.2 La tâche d'ordonnancement

La tâche d'ordonnancement accorde à chacun des flux identifiés par la classification une priorité de sortie, les flux les plus prioritaires étant traités les premiers. Le but de cette tâche est de garantir une part de la bande passante de sortie aux flux prioritaires. Ce type de tâche sert à assurer la qualité de service (QoS). Le but de cette tâche est de choisir, pour chaque paquet à sortir, dans quelle fifo de priorité on va aller le lire.

Cette tâche vient à la suite de la précédente. La tâche de classification a séparé les paquets en différents flux, et le rôle de la tâche d'ordonnancement est d'arbitrer leur ordre de sortie. Pour cela, chacun des canaux d'entrée de la tâche d'ordonnancement est doté d'une priorité. cette priorité correspond à la part de la bande passante qui est réservée au flux concerné. La somme de toutes les priorités ne peut être supérieure au débit maximal de sortie de la plateforme. La tâche d'ordonnancement conserve en mémoire le volume total de donnée qui a été émis pour chaque interface de sortie, et le volume de données émis pour chaque canal de priorité.

Ces valeurs permettent de déterminer quelles files de priorité ont leur contrat satisfait (leur part du total est supérieure ou égale à ce que garantit leur contrat), et quelles sont celles qui sont en souffrance. Pour chaque paquet émis, on ajoute sa taille au volume total de données émis de l'interface concernée, et au volume de données du canal servi.

4.4 Conclusion

Les spécifications que l'on donne pour les applications candidates sont donc que ces applications doivent pouvoir être représentées sous la forme d'un graphe de tâches qui communiquent entre elles de façon asynchrone. On ne fait pas d'hypothèses sur la nature des tâches, ni sur les données qu'elles s'échangent par ces canaux. On ne pose pas non plus de restrictions quand au nombre de producteurs et de consommateurs qui peuvent être connectés à chaque canal. Ces spécifications nous ont amenés à définir le protocole MWMR qui est l'objet du chapitre suivant. Ce protocole définit les communications entre tâches par mémoire partagée.

L'application de classification ordonnancement que l'on a spécifiée est un exemple d'application qui utilise les deux types de parallélisme que l'on vient de décrire. Cette application utilise des canaux multipoints, à plusieurs lecteurs et/ou plusieurs consommateurs. Il s'agit donc d'un exemple représentatif du type d'application que l'on souhaite pouvoir déployer sur puce en utilisant ces primitives de communication. Cette application sera utilisée au chapitre 7 pour valider le fonctionnement et mesurer les performances des canaux de communications que nous implémenterons.

Chapitre 5

Les canaux de communication MWMR

Sommaire

Les canaux de communication que l'on va définir dans ce chapitre implémentent les spécifications détaillées au chapitre 4. On décrit donc ici un mécanisme de communication asynchrone entre des tâches productrices de données et des tâches consommatrices de ces données. Cet asynchronisme entre production et consommation des données est à l'origine de notre choix d'utiliser un tampon de mémoire partagée organisé suivant le protocole FIFO.

5.1 Le protocole des canaux MWMR

MWMR est un acronyme signifiant Multi Writer, Multi Reader. Cela signifie que c'est un canal de communication qui se comporte comme une FIFO logicielle à laquelle peut être connecté un nombre quelconque de producteurs et de consommateurs. Le protocole des canaux MWMR est donc l'ensemble des règles qui régissent l'accès à un espace de mémoire partagée.

5.1.1 Le tampon en mémoire

Le canal MWMR est un espace de mémoire partagée, muni des structures de contrôle nécessaire pour synchroniser les accès dans un contexte multi-écrivain, multi-lecteurs.

Un canal MWMR est un objet logiciel auquel peuvent accéder des tâches logicielles ou des coprocesseurs matériels. Ni le nombre, ni la nature des tâches productrices et consommatrices connectées à cette fifo ne sont connus. Le canal de communication MWMR est donc principalement composé d'un tampon alloué dans la mémoire du système.

Nous avons besoin d'un protocole d'accès construit au dessus de cette mémoire partagée. Le fait que certaines tâches qui vont utiliser ces fifos seront implémentées en matériel impose une contrainte de simplicité au protocole utilisé, puisque ce protocole devra être mis en œuvre par un automate cablé.

5.1.2 Protéger l'accès aux données

La première chose à faire lorsqu'on établit une zone de mémoire partagée entre plusieurs tâches est de s'assurer de la cohérence du contenu. Ce qui ne peut être

garanti que si l'on protège l'accès à ce contenu : à un instant donné, il ne peut jamais y avoir qu'une seule tâche qui modifie une adresse particulière.

Il faut donc, en premier lieu, identifier quels sont ces contenus dont il faut maintenir la cohérence. Il y a bien sûr le tampon qui contient les données à échanger, et qui doit être protégé contre des accès simultanés en lecture ou en écriture. La protection contre des accès simultanés en lecture est indispensable : si plusieurs tâches lisent une donnée à la même adresse, la donnée est dupliquée, ce qui ne correspond pas à un comportement FIFO. La protection contre des accès simultanés en écriture sert à empêcher l'effet inverse : l'écrasement et par conséquent la perte de données.

On peut en principe envisager une lecture et une écriture au même instant : les deux accès se passent à des adresses différentes. Ces adresses de lecture et d'écriture dans la fifo sont deux pointeurs en mémoire qui sont lus et mis à jours par les tâches concernées, et doivent donc être protégés contre les accès concurrents. La dernière chose à protéger est le statut de la fifo, qui définit le nombre de données présentes dans la fifo à un instant précis. Cette valeur est lue et modifiée par toutes les tâches qui accèdent au canal. Elle ne peut donc supporter qu'un seul accès à la fois.

De ces éléments, il ressort que l'on a trois groupes de données à protéger contre des accès concurrents :
- Le pointeur de lecture et les données pointées : une seule tâche consommatrice à la fois.
- Le pointeur d'écriture et la case vide pointée : une seule tâche productrice à la fois.
- Le statut de la fifo : une seule tâche, productrice ou consommatrice, à la fois.

Un accès à un canal fifomwmr utilise deux de ces groupes ; la lecture et la mise à jour du statut étant communes aux deux types d'accès. Si l'on choisit de protéger chacun de ces trois ensembles par un verrou séparé, on a besoin de trois verrous par fifo, chaque accès en prenant deux. Cela permet deux accès simultanés : une lecture et une écriture. L'algorithme d'accès à la fifo est dans ce cas, pour une lecture :

- prendre le verrou du statut
- lire le statut
- rendre le verrou du statut
- **Si** le statut est bon (il y a assez de données à lire)
  - prendre le verrou de lecture
  - lire le pointeur de lecture
  - lire les données
  - mettre à jour le pointeur de lecture
  - rendre le verrou de lecture
  - prendre le verrou du statut

  - mettre à jour le statut
  - rendre le verrou du statut
- **sinon** : fin

L'algorithme est identique pour une écriture.

On voit que le nombre d'accès à la mémoire nécessaire pour l'implémentation de ce protocole est important, en totale contradiction avec notre objectif initial de définir un protocole simple pour ne pas surcharger les automates des coprocesseurs qui auront à implémenter ces accès.

En conséquence, le bénéfice de pouvoir faire deux accès simultanés à la fifomwmr n'est pas suffisant pour justifier la compléxité du protocole décrit ci-dessus. Bénéfice d'autant plus hypothétique que du point de vue matériel, un banc de RAM ne peut répondre qu'à un seul accès à la fois : s'il y a plusieurs accès concurrents, ils sont séquentialisés et ont lieu les uns après les autres.

Chaque canal fifo est donc protégé par un unique verrou. Ce verrou protège l'ensemble de l'accès à la fifo, pointeurs, statut et données. On perd la capacité à traiter deux accès simultanés, mais on simplifie grandement le protocole d'accès.

5.1.3 Protocole à 5 étapes

Le protocole MWMR se décompose en 5 étapes qui sont autant d'accès à la mémoire :

- Prise du verrou.
- Consultation du statut de la fifo, et lecture de la valeur du pointeur de lecture (ou d'écriture).
- Transfert des données, du tampon local vers le tampon de communication en cas d'écriture, du tampon de communication vers le tampon local en cas de lecture.
- Mise à jour du statut et du pointeur.
- libération du verrou.

5.1.4 Accès non bloquant

Un canal MWMR est caractérisé par sa profondeur et sa largeur. Un élément de données atomique est appelé un item. La profondeur de la fifo est le nombre maximum d'items qu'elle peut contenir. La largeur de la fifo est le nombre de mots de 32 bits contenus dans un item. La bibliothèque fifomwmr fournit deux fonctions pour permettre aux tâches logicielles d'accéder à une fifo : `mwmr_read()` et `mwmr_write()`. Ces deux fonctions prennent chacune trois arguments. Le premier est un pointeur sur le canal fifomwmr lui même, le second un pointeur sur le tampon de mémoire locale, et le troisième est le nombre d'éléments à transmettre. Pour des raisons d'optimisation des temps de transfert, ce dernier para-

mètre est exprimé en mots de 32 bits et il doit correspondre à un nombre entier d'items.

Ces fonctions sont non bloquantes. Cela signifie qu'elles reviennent toujours à l'appelant même si la requête n'est pas satisfaite. Ces fonctions renvoient un entier, qui indique le nombre de mots qui ont été effectivement transférés. Ce nombre de mots transférés correspond toujours à un nombre entier d'items. Si cette valeur est inférieure au nombre demandé, le traitement à effectuer reste à la charge de la tâche demandeuse. Elle peut par exemple tenter de lire dans une autre fifo. Ceci peut être le cas d'une tâche de routage ipv4 qui va essayer un autre canal d'entrée si le premier est vide.

5.1.5 Les accès bloquants

On appelle accès bloquant un appel à une fonction de lecture ou d'écriture qui ne retourne qu'une fois le transfert complètement effectué. C'est un comportement souhaitable pour une tâche qui a besoin d'une certaine quantité de données, issues du même canal avant de pouvoir travailler. C'est un cas fréquent dans les applications multimédia où les données sont traitées par blocs.

Pour pouvoir utiliser les canaux MWMR dans ce type d'applications, on a donc développé des primitives de communications bloquantes. Ces primitives ont été développées au dessus des fonctions non bloquantes `mwmr_read()` et `mwmr_write()`. Elles implémentent une boucle autour de l'appel à la fonction non bloquante correspondante. La boucle est itérée le nombre de fois nécessaire à la résolution du transfert. A chaque tour dans cette boucle, si le transfert n'est pas terminé, la tâche est déchargée du processeur, mais elle reste éligible. On effectue ce déchargement pour permettre à une autre tâche d'être exécutée sur le processeur. Ceci permet, d'une part d'éviter d'utiliser un processeur à faire fonctionner une tâche bloquée en attendant une ressource, et d'autre part, cela peut permettre d'éviter un risque d'interbloquage. Cet interbloquage survient si la tâche effectuant une lecture bloquante est exécutée sur le même processeur que la tâche qui écrit dans ce canal. En l'abscence d'un mécanisme de remplacement de tâche sur le processeur, par exemple après chaque interruption horloge, la tâche en cours d'exécution sur le processeur y demeure indéfiniment, empêchant la tâche productrice de données d'être exécutée.

L'algorithme implanté pour effectuer une lecture bloquante est donc le suivant :

b_mwmr_read(fifomwmr f, char *buf, int n)

- n_lu = 0
- **tant que** (n_lus < n)
  - n_lus += mwmr_read(f, buf + n_lus, n - n_lus)
  - **si** (n_lu < n) **alors**

    - déchargement de la tâche
    - élection d'une nouvelle tâche
  - **finsi**
- **fin tant que**

Dans ce cas, les primitives de communication deviennent complètement déterministes et il est possible d'émuler le comportement et la sémantique des réseaux de Kahn. Les canaux de Kahn sont donc un cas particulier du formalisme fifomwmr : chaque canal ne doit être connecté qu'à un producteur et un consommateur, et les accès doivent utiliser les fonctions bloquantes.

Dans le cas particulier où l'on n'a qu'une tâche par processeur, le mécanisme de déchargement-élection de tâche est inutile : s'il n'y a qu'une tâche prête à être exécutée de disponible, c'est elle qui sera élue, juste après avoir été déchargée.

5.2 Utilisation d'un verrou actif à attente active

Chaque canal fifomwmr est protégé par un verrou unique. Ce verrou est un dispositif permettant de protéger les accès au canal. A un instant donné, il y a au plus une tâche qui a accès à une fifo. La nature logicielle ou matérielle de cette tâche n'est pas connue a priori. Elle n'a d'ailleurs aucune importance pour ce qui est des modifications que cette tâche va effectuer dans la fifo.

Les tâches qui utilisent les fifomwmr sont exécutées soit par des processeurs programmables soit par des coprocesseurs matériels spécialisés. Dans ces deux cas, l'accès à la fifo est entièrement résolu par la tâche. Dans le cas d'un coprocesseur matériel, c'est un automate cablé qui devra réaliser la prise de verrou. C'est un aspect important à prendre en compte lors du choix du type de verrou utilisé.

Il esiste différentes solutions implémentant des verrous : les sémaphores, les mutex et les spinlock. Ces solutions sont la plupart du temps optimisées pour régler des accès concurrents effectués par des tâches logicielles (typiquement, des threads POSIX), qui peuvent s'exécuter sur le même processeur. Mutex et sémaphores sont des structures de données intéressantes pour les applications logicielles multi tâches qui s'exécutent sur un seul processeur grâce à un mécanisme de multiplexage temporel. Le système d'exploitation dispose d'un mécanisme d'ordonnancement permettant de sélectionner la tâche à laquelle affecter un processeur. On parle de tâche élue. Lorsque le système ne dispose que d'un seul processeur, il ne peut y avoir qu'une seule tâche élue à la fois.

Le principe des sémaphores et mutex est d'accorder une ressource à un nombre maximal N prédéfini de tâches. Le mutex est un cas particulier de sémaphore où N=1. Le principe est d'endormir la N+1 ième tâche demandeuse du verrou puisque la ressource n'est pas disponible. Ceci permet de décharger la tâche du

processeur pour laisser du temps de calcul à une autre tâche. La tâche demandeuse est attachée à la ressource mutex, et lorsque le mutex est libéré, la fonction pthread_mutex_unlock() parcourt la liste des tâches en attente sur ce verrou et les replace dans la liste des tâches éligibles de l'ordonnanceur. Notre contexte est différent puisque les tâches implantées sur des coprocesseurs matériels sont en concurence avec les tâches logicielles pour l'accès aux ressources. Il faut donc que ces tâches matérielles puissent accéder aux verrous. Or il n'y a aucune utilité à endormir un coprocesseur matériel puisque le coprocesseur ne peut pas être utilisé pour exécuter une autre tâche. Les sémaphores ne sont donc pas très adaptés au partage de ressources entre des tâches logicielles, dont le fonctionnement est régi par le système d'exploitation, et des tâches matérielles (des coprocesseurs) que le système ne peut pas endormir. Ajoutons que pour les architectures multi-processeurs visées, il arrive fréquemment qu'un processeur n'exécute qu'une seule tâche, ce qui rend inutile le mécanisme d'endormissement.

Il existe un autre type de verrou logiciel, de plus bas niveau : le spin_lock. C'est une structure simple, composée d'une unique adresse mémoire, qui n'a que deux valeurs possibles : libre ou pris. La norme POSIX définit deux primitives pour y accéder : SEM_LOCK et SEM_UNLOCK. La première prend le verrou, en réalisant une attente active et ne retourne qu'une fois celui-ci obtenu. Il s'agit donc d'un appel bloquant. La seconde libère ce même verrou. Il n'y a pas ici de notion de liste de tâches en attente à réveiller lorsqu'un spin_lock est libéré. Ce fonctionnement simple en fait un bon candidat pour les canaux MWMR car ces primitives SEM_LOCK et SEM_UNLOCK peuvant être implémentées sur du matériel sans que cela soit trop coûteux.

5.3 Les grandes lignes de l'architecture matérielle visée

L'architecture matérielle est décrite en détail au chapitre 7.2. On en donne cependant quelques caractéristiques ici car elles aident à comprendre les choix d'implémentation.

L'architecture matérielle visée est composée d'un système d'interconnexion auquel sont connectés tous les modules. Les interfaces entre ces modules et le réseau d'interconnexion respectent la norme VCI [VSI00]. Cette norme définit les ports d'entrée/sortie, ainsi que le protocole à respecter pour établir une communication. La norme VCI reconnait deux types d'agents : les initiateurs et les cibles. Une communication VCI a lieu entre un initiateur et une cible. Cette communication est toujours composée d'une requête, émise par l'initiateur, suivie d'une réponse, fournie par la cible. C'est cette paire requête/ réponse qui constitue une

transaction VCI. Il est à noter que cette transaction peut comporter une ou plusieurs adresses. Dans ce cas, on parle de transaction en rafale, et la réponse comporte autant d'éléments que la requête. L'arbitrage qui accorde le bus à un initiateur est à la charge du réseau d'interconnexion. La norme VCI définit des champs qui permettent d'identifier les émetteurs et destinataires des paquets. Le routage de la requête et de la réponse associée s'appuient sur ces champs.

On a vu que les modules connectés au réseau d'interconnexion VCI étaient de deux types : les initiateurs et les cibles. une cible est désignée comme destinataire d'une requête particulière par la ou les plage(s) d'adresse(s) auxquelle(s) elle est associée. Du point de vue des initiateurs, toutes les cibles sont vues comme des plages d'adresses, contigües ou pas. Cependant, derrière cette uniformité du protocole d'accès se cachent des ressources disparates : de la mémoire ou des registres adressables appartenant à des périphériques.

L'un de ces périphériques mérite une attention particulière, il s'agit du périphérique lock. C'est lui qui fournit le support matériel nécessaire à l'implantantion des verrous sur notre architecture cible. Il associe un verrou binaire à chaque adresse. La valeur de chaque verrou ne peut être que libre ou pris. Une demande de verrou consiste en une lecture à son adresse. Si le verrou est libre, le coprocesseur renvoie la valeur 0 (libre) et change cette valeur à 1 (pris). Si ce verrou est déjà pris, le coprocesseur renvoie la valeur 1 (pris) sans modifier son état interne. Un verrou est libéré en effectuant une écriture à son adresse, ce qui a pour effet de remettre sa valeur à 0 (libre).

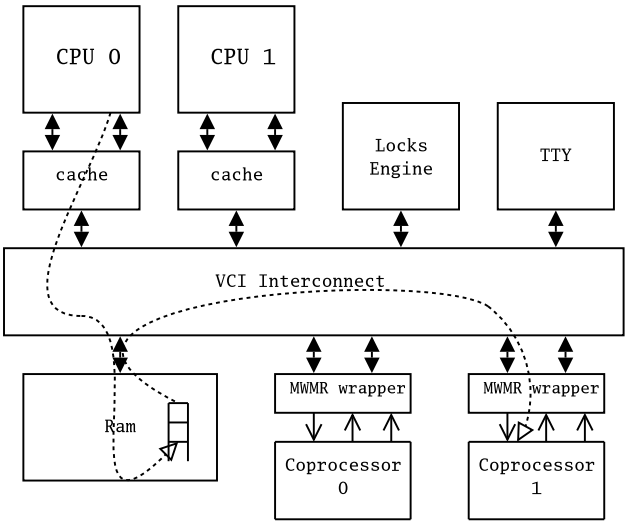

FIG. 5.1 – L'architecture matérielle avec un exemple d'utilisation de canal MWMR partagé entre une tâche logicielle et une tâche matérielle.

La figure 5.1 montre un exemple d'utilisation de canal MWMR avec une tâche

logicielle écrivain, et une tâche matérielle lectrice.

5.4 L'implémentation logicielle des primitives d'accès aux canaux MWMR

5.4.1 Les API des fonctions `mwmr_read` et `mwmr_write`

Comme on l'a vu, on a choisi d'utiliser un protocole non bloquant pour les primitives d'accès `mwmr_read` et `mwmr_write`. L'implémentation que nous avons choisie pour ces deux primitives essaye de lire (écrire) le maximum de données à concurrence de la valeur demandée. Cela signifie par exemple que si l'on essaye de lire 5 mots dans un canal MWMR qui n'en contient que trois, ceux-ci sont copiés, et le code de retour de la fonction est trois. Nous avons fait ce choix pour éviter des bloquages dus à des requêtes plus grandes que ce que la fifo peut contenir.

Les prototypes de ces deux fonctions sont donc :

```
int mwmr_read (fifomwmr *f,
               char * buffer,
               int n)
int mwmr_write (fifomwmr *f,
                char * buffer,
                int n)
```

Où `f` est une structure `fifomwmr`, `buffer` la zone de mémoire locale contenant les données à écrire ou à lire dans cette fifo. `n` est le nombre de mots à transférer.

5.4.2 Les problèmes de cohérence liés à l'utilisation des caches

Les deux fonctions `mwmr_read` et `mwmr_write` doivent implémenter les 5 étapes du protocole MWMR. Chacune de ces 5 étapes contient au moins un accès à la mémoire. Ces accès mémoire se situent dans une zone de mémoire partagée. Or la mémoire partagée entre plusieurs processeurs pose un problème de cohérence.

Tant que l'on utilise un système matériel à base de bus, le mécanisme d'espionnage du bus (snoop) permet de résoudre ce problème. Le mécanisme de snoop, dans un système multiprocesseur construit autour d'un bus, permet à tous les contrôleurs de cache de surveiller toutes les adresses émises. Selon les implémentations, soit ils invalident leur copie d'une donnée partagée lorsqu'ils voient passer son adresse assortie d'une requête d'écriture, soit un des caches dispose

d'une copie plus récente de ladite valeur (parce qu'il vient de la modifier et que sa strategie d'écriture est `write back`), auquel cas, c'est le cache qui fournit la réponse à la requête de lecture, en lieu et place de la mémoire. Avec l'émergence des architectures construites sur des micro réseaux intégrés sur puce (NoC), ces techniques d'espionnage du bus ne fonctionnent plus car il peut y avoir un grand nombre de transactions simultanées.

Une solution a été étudiée pour résoudre ce probléme dans l'architecture multiprocesseur DASH [HHG99]. Elle met en œuvre un mécanisme appelé *Directory based*, initialement proposé par [CF78], dont le principe est d'associer à chaque ligne de cache une zone de N bits, à raison d'un bit par processeur. Chacun de ces bits prend la valeur 1 lorsque le processeur (ou plutôt son cache) possède une copie de cette ligne de cache. Lorsque cette ligne de cache est modifiée par une écriture, tous les caches qui en ont une copie sont alertés et doivent invalider ou mettre à jour l'adresse correspondante. Ce mécanisme fonctionne mais il est très coûteux, puisqu'il faut, d'une part, consacrer une partie non négligeable de la mémoire à stocker des informations de contrôle sur la présence de chaque donnée dans chaque cache. ceci représente une quantité de mémoire qui croît avec le nombre de processeurs, ainsi qu'avec la taille de la zone de mémoire partagée. D'autre part, on a vu que c'est la mémoire qui prévient les caches des modifications effectuées. Cela entraine une complexification des modules matériels : Chaque banc mémoire qui contient des données partagées doit pouvoir envoyer des requêtes et donc se comporter comme un initiateur. Symétriquement, chaque contrôleur de cache doit pouvoir traiter ces requêtes, c'est à dire se comporter comme une cible. Ces solutions sont trop lourdes à mettre en œuvre, et nous proposerons des solutions qui ne nécessitent qu'un petit concours du matériel.

Avant de présenter les deux méthodes mises en oeuvre pour résoudre ce problème de cohérence, nous dressons un inventaire de tous les accès à la mémoire effectués par un processeur lors de l'utilisation d'une des primitives d'accès aux canaux MWMR.

Lorsqu'on détaille les 5 étapes du protocole MWMR, on voit que chacune de ces étapes contient au moins un accès à la mémoire.

Prendre le verrou. Ceci masque en fait deux accès à la mémoire. Le premier consiste à récupérer l'adresse dudit verrou, information contenue dans la structure fifomwmr. Le second accès consite à prendre ce verrou, ce qui nécessite au moins une lecture.

Tester le statut. Comme on l'a dit, cette étape est en fait la contraction de deux étapes : le test du statut proprement dit, suivi de la lecture de l'adresse du pointeur dans la structure fifomwmr. Celle-ci est suivie d'une troisième lecture pour connaitre la profondeur de la fifo, information qui permet de déterminer si le transfert aura lieu à des adresses strictement croissantes ou s'il faut effectuer un modulo. Si on appelle N le nombre de mots à lire, on a l'inégalité : si $readp +$

$N \leq length$ alors les adresses du paquet sont contigües, sinon il faut découper le transfert en deux paquets : le premier paquet fait $profondeur - readp$ mots, et le second $N - (profondeur - readp)$. Cette étape nécessite donc trois accès à la mémoire.

Effectuer le transfert des données. Dans le cas d'une lecture, il s'agit d'aller lire les données dans la zone de données de la fifo et de les recopier dans le tampon local. Chaque mot transféré correspond donc à deux accès mémoire : une lecture et une écriture.

Mettre à jour le statut de la fifo : là encore, il y a deux mises à jour à faire : le statut et le pointeur de lecture, ce qui fait deux écritures.

Libérer le verrou. La libération du verrou est une simple écriture à son adresse.

Comme on le voit, le nombre d'accès faits à la mémoire pour l'implantation de ce protocole est élevé. Le coût des écritures peut être masqué grâce à l'existence dans le cache d'un tampon d'écritures postées, mais le coût des lectures en zone non cachée peut être élevé : quelques dizaines de cycles pendant lesquels le processeur reste gelé.

```
typedef struct fifomwmr{
  void * pointeur;
  LOCK_T lock;
  unsigned int largeur;
  unsigned int profondeur;
  unsigned int status; // nombre de mots occupées
  unsigned int readp;  // nombre de mots déjà lus
                       // (modulo la profondeur)
  unsigned int writep; // nombre de mots déjà écrits
                       // (modulo la profondeur)
  unsigned int *data;
}fifomwmr;
```

FIG. 5.2 – La structure fifomwmr

La structure fifomwmr est présentée dans la figure 5.2. Cette structure comporte huit champs de 32 bits. Le champ `pointeur` n'est pas utilisé. Le champ `lock` est l'adresse du verrou associé à la fifomwmr, les deux champs suivants sont des champs constants initialisés lors de la création du canal fifomwmr. La `largeur` correspond à la taille d'un item contenu dans la fifomwmr, exprimée en mots de 32 bits. La `profondeur` est le nombre maximal d'items que peut contenir la fifomwmr. Le champ `status` contient le nombre de mots présents dans la fifomwmr à un instant donné, il est testé et modifié par tous les accès à cette fifomwmr. Les champs `readp` et `writep` contiennent respectivement le

nombre de mots lus et écrits dans la fifomwmr. Ils sont utilisés pour calculer les adresses des prochaines lectures et écritures dans cette fifomwmr. Le champ `data` est un pointeur sur le début de la zone de données partagées.

5.4.3 Premier mécanisme : implémentation avec préchargement

L'idée directrice est d'implanter les canaux de communication MWMR dans des segments de mémoire non cachable, aussi bien pour les données que pour les descripteurs. Pour minimiser le coût des MISS, il faut exploiter la localité spatiale, et utiliser le mode rafale fourni par la norme VCI-OCP. Pour cela on peut utiliser le mode préchargement du cache. Le préchargement permet d'exploiter la localité spatiale, sans mettre en œuvre la rémanence des données dans la cache. Pour cela on utilise un cache capable d'accéder à la mémoire de trois façons différentes :

- **Cachée** : ce segment mémoire peut être lu et conservé par le cache sans précaution particulière pour maintenir la cohérence. C'est le cas de la mémoire non partagée, c'est à dire utilisée par un seul processeur, comme c'est le cas pour la pile d'exécution d'une tâche si l'on interdit la migration de tâches, ou de la mémoire non inscriptible (là où se trouve le code de l'application). C'est le cas le plus courant. Une lecture effectuée à une adresse appartenant à un segment caché déclenche une transaction en mode rafale : on lit une ligne de cache qui est ensuite rangée dans le cache.
- **non cachée** : ce sont les segments mémoire partagée entre plusieurs processeurs : soit un espace mémoire explicitement partagé entre plusieurs processeurs, comme les ressources du système d'exploitation, qui sont accessibles par tous les processeurs. C'est aussi le cas des périphériques adressables. Une requête de lecture dans un segment non caché déclenche une transaction VCI simple, c'est à dire d'un seul mot.
- **pré-chargeable** : c'est un compromis entre les deux types ci-dessus. Une requête de lecture dans un segment préchargeable déclenche une requête de type rafale : on lit une ligne de cache complète, mais celle-ci n'est pas rangée dans le cache, mais dans un tampon de préchargement. Elle reste disponible dans ce tampon tant que le processeur n'effectue aucun autre accès à la mémoire. C'est dans ces espaces mémoire que l'on implante les canaux MWMR.

Le type d'accès à la mémoire est déterminé par le cache en fonction des bits de poids fort de l'adresse.

La ligne de cache ainsi chargée dans le tampon de préchargement devient invalide dès que le processeur effectue un accès mémoire dans une autre ligne de cache, en lecture comme en écriture. Cette caractéristique nous permet d'éviter de conserver dans le cache des données périmées, tout en bénéficiant de l'accès en rafale.

On vient de voir que la lecture d'une adresse pré chargeable déclenche la lecture de toute la ligne de cache associée. La première lecture du protocole est celle de l'adresse du verrou. Cependant, la deuxième chose que l'on doit faire pour suivre le protocole est de prendre le verrou : c'est à dire faire une lecture dans le périphérique `lock` , qui est dans un autre segment, ce qui invalide toute la ligne précédemment ramenée. On ne peut pas considérer la valeur du statut de la fifo comme valide avant de disposer du verrou. On est donc dans l'obligation de relire ce statut une fois le verrou pris. C'est cette deuxième lecture d'une donnée pré-chargeable que l'on exploite, en copiant en mémoire locale (dans la pile) les valeurs dont on aura besoin : le statut, l'adresse du verrou, le pointeur `readp` (on fait l'hypothèse qu'il s'agit ici d'une lecture), la profondeur de la fifo, le début de la zone de données. Toutes ces valeurs sont copiées dans des variables locales (stockées dans la pile), et ce sont ces variables que l'on manipule lorsqu'on fait un calcul intermédaire.

La deuxième partie de l'optimisation concerne le transfert des données. Pour tirer parti du tampon de préchargement, il faut grouper les instructions de lecture, et pour cela on effectue des rafales de lecture suivant le nombre de mots à transférer. Ces rafales sont de taille décroissante. On commence par des lectures de 16 mots (la longueur d'une ligne de cache), puis 8, 4, 2 et un seul mot. Chaque rafale de lectures est suivie d'une rafale d'écritures de même taille. Ces transferts sont rapides, mais ils sont gourmands en ressources, puisque les 16 mots lus doivent être stockés dans les registres du processeur. C'est également gourmand en taille de code, puisque les boucles de lecture sont largement déroulées.

5.4.4 Second mécanisme : invalidation sélective du cache

La technique de préchargement présentée au paragraphe précédant suppose que tous les contrôleurs de cache du système disposent d'un mécanisme de préchargement. Cette contrainte est très forte, et c'est pourquoi nous avons évalué un autre mécanisme où la mémoire nécessaire aux fifomwmr est allouée dans un segment cachable. Ceci permet de bénéficier du mécanisme de lecture en rafale fourni par le contrôleur du cache.

Le mécanisme assurant la cohérence des données utilise une fonction disponible dans la plupart des processeurs embarqués : la possibilité d'invalider une ligne de cache par logiciel. Dans le cas du processeur MIPS R3000, qui ne possède pas d'instruction d'invalidation on utilise un contournement : toute instruction de lecture en mémoire ayant pour destination le registre $0 est interpretée comme une requête d'invalidation de la ligne de cache correspondante.

Dans les deux fonctions, `mwmr_read` et `mwmr_write`, on a besoin d'invalider les lignes de cache qui contiennent le statut et les pointeurs `readp` et `writep`, leurs valeurs cachées étants considérées comme non valides. Ensuite,

on a besoin de l'instruction d'invalidation dans la fonction de transfert des données : l'adresse à partir de laquelle on lit dans la fifo est peut-être présente dans le cache, et il faut s'assurer que l'on lit les valeurs en mémoire. Il peut y avoir plusieurs lignes de cache à invalider : cela dépend du nombre de mots à lire, de la taille du cache, et de la contiguïté des adresses de lectures.

Cette seconde implémentation met à la charge du logiciel de communication d'assurer la cohérence de ces canaux MWMR, en invalidant les lignes de caches succeptibles de contenir des données obsolètes. Le code de ces fonctions devient très dépendant de l'architecture matérielle utilisée (type de processeur, taille des ligne de cache). Il s'agit d'une contrainte assez lourde contre la portabilité, mais c'est un passage obligé si l'on veut des obtenir performances correctes en terme de bande passante dans les canaux MWMR.

5.5 Le contrôleur matériel MWMR

Pour maximiser les possibilités de réutilisation des coprocesseurs matériels spécialisés, on souhaite séparer clairement les fonctions de calcul des fonctions de communication. On a donc décidé d'encapsuler le protocole d'accès aux canaux de communication MWMR dans un composant matériel générique appelé *contrôleur MWMR*. Il s'agit d'un composant d'interface entre le micro-réseau permettant l'accès à la mémoire et le coprocesseur matériel. Du coté du micro réseau, il se comporte comme un initiateur, puisqu'il est capable de lire ou d'écrire dans un ou plusieurs canaux de communication MWMR. Du coté du coprocesseur, le contrôleur MWMR fournit une interface de communication sans adresse particulièrement simple à utiliser.

Sur la figure 5.3, on voit la place occupée par le contrôleur MWMR : il se situe entre le coprocesseur et l'interconnect VCI. Son rôle est de *traduire* les requêtes de lectures/écritures de données faites par le coprocesseur en accès aux canaux MWMR en respectant le protocole en 5 étapes. Le contrôleur MWMR de la figure 5.3 utilise trois canaux de communication.

5.5.1 L'interface du contrôleur

L'utilisation de ce contrôleur matériel se décompose en deux phases. La première phase est la configuration du contrôleur. Il s'agit de lui envoyer les paramètres correspondant à chacun des canaux MWMR qu'il utilise. Elle est généralement effectuée lors du démarrage du système, en même temps que la création des tâches du système. La seconde phase est l'utilisation fonctionnelle : le contrôleur effectue les accès en lecture et/ou en écriture dans les canaux MWMR pour lesquels il est configuré, à la demande du coprocesseur. On peut dire que

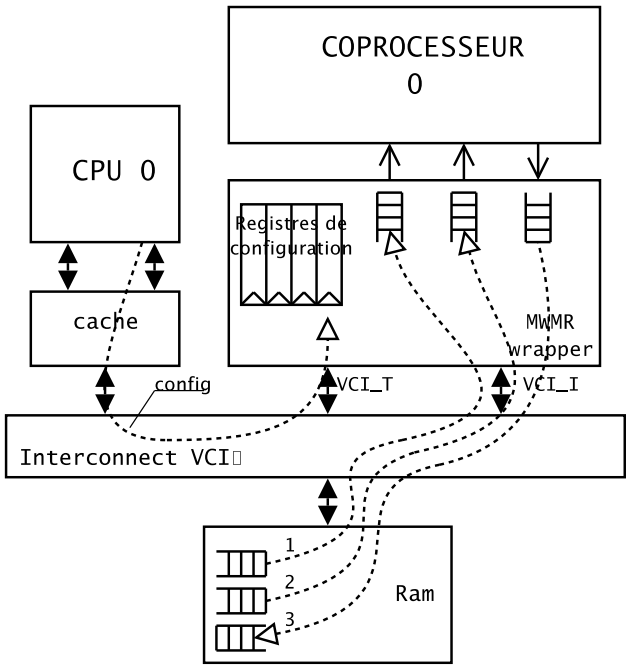

FIG. 5.3 – Position du contrôleur MWMR, entre le coprocesseur et l'interconnexion à interface VCI

le contrôleur implémente les fonctions `mwmr_read` et `mwmr_write` pour les coprocesseurs.

Ce rôle de *traducteur de protocole* se retrouve dans l'interface du contrôleur : du côté du coprocesseur on a une ou plusieurs interfaces fifo en lecture et une ou plusieurs interfaces fifo en écriture. Du côté de l'interconnect VCI, on trouve deux ports, un port initiateur qui accède aux canaux MWMR en lecture et en écriture, et un port cible qui est utilisé pour la configuration du système.

5.5.2 Le protocole de communication avec le coprocesseur

Le protocole de communication du côté du coprocesseur est un protocole sans adresse, à sens unique. Il s'agit du protocole fifo, qui supporte un mécanisme de contrôle de flux extrêment simple à implémenter en matériel.

Les signaux d'interface du protocole FIFO sont décrits sur la figure 5.4. Il y a deux interfaces une pour les lectures, une pour les écritures. Seul le sens du signal DATA change. Dans notre cas, le signal DATA est un signal sur 32 bits. Les signaux de contrôle sont des signaux booléens.

Lorsque le coprocesseur souhaite écrire une donnée, il active le signal W et place la donnée sur le signal DATA. Si le contrôleur est en état de lire cette don-

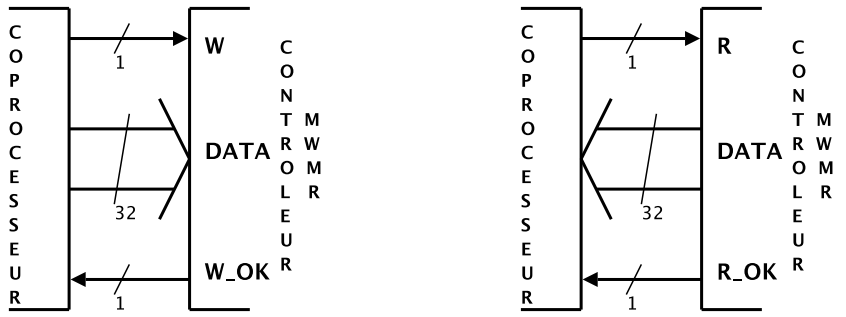

FIG. 5.4 – Les deux interfaces fifo, écriture et lecture (du point de vue du coprocesseur)

née, il active le signal W_OK. Le transfert a lieu lorsque les deux signaux sont actifs. Le transfert peut se poursuivre durant plusieurs cycles, au rythme de un mot transféré par cycle. Si pour une raison ou une autre le contrôleur n'est pas en état d'accepter un mot (par exemple parce que son tampon interne est plein), il baisse le signal W_OK et le transfert s'interrompt. Cette sémantique est illustrée par le chronogramme de la figure 5.5.

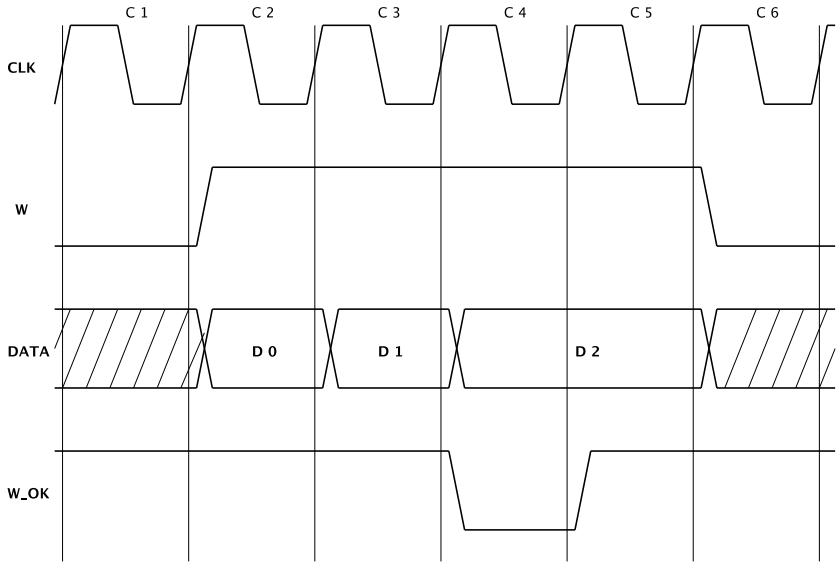

FIG. 5.5 – Chronogramme du protocole fifo pour une écriture

Un coprocesseur peut être composé de plusieurs automates plus ou moins indépendants les uns des autres et qui communiquent avec des canaux MWMR de façon asynchrone. Par conséquent, le contrôleur MWMR fournit plusieurs interfaces FIFO en lecture, et plusieurs interfaces FIFO en écriture. Ces deux nombres sont bornés à 4 pour des raisons d'implantation.

En plus de l'accès aux canaux de communication MWMR, le contrôleur MWMR fournit deux autres services (optionnels) au coprocesseur :

Il arrive fréquement qu'un coprocesseur cablé (donc non programmable) dispose de certains paramètres de configurations qui peuvent être modifiés dynamiquement par logiciel. Le contrôleur MWMR fournit jusqu'à 8 registres de configuration de 32 bits adressables en écriture par le logiciel.

Un coprocesseur cablé doit également pouvoir fournir au logiciel des informations sur l'état d'avancement du calcul. Le contrôleur MWMR fournit jusqu'à 8 registres de statut de 32 bits adressables en lecture par le logiciel.

5.5.3 Implémentation

Du coté de l'interconnect VCI, on trouve deux ports : un port cible et un port initateur.

Le port VCI cible est utilisé par l'application embarquée pour effectuer la configuration du contrôleur lors de l'initialisation de l'application : c'est par ce port que le logiciel vient configurer chacun des canaux MWMR qui vont être utilisés par le coprocesseur. Ce port sert également pour venir configurer le coprocesseur lui-même, ainsi que pour venir consulter son statut ou tout autre information qu'il rend disponible. Le coprocesseur occupe 1 Kilo octets dans l'espace adressable : la destination de la donnée entrante est déterminée en décodant les 10 bits de poids faible de l'adresse à laquelle elle est destinée. Ce décodage s'effectue selon le tableau 5.6

Le second port VCI est un port initiateur. C'est ce port qui est utilisé par le contrôleur MWMR pour transmettre les requêtes du coprocesseur à la mémoire.

L'attribution du port VCI maître à un des canaux fifo du coprocesseur est faite selon un algorithme de priorité tournante (round robin) : cet arbitrage garantit que tous les canaux sont servis. L'automate d'état qui pilote le port VCI implémente le protocole MWMR à 5 étapes décrit plus haut : après avoir sélectionné une requête à servir, le contrôleur essaie d'obtenir le verrou. Ici le fonctionnement est différent de la fonction logicielle pthread_spin_lock() : cette fonction POSIX est bloquante. C'est à dire qu'elle ne retourne que lorsqu'elle a réussi à obtenir le verrou. Tandis que ce que fait le contrôleur MWMR est d'essayer de prendre le verrou : si celui-ci n'est pas libre, l'automate essaie de sélectionner une autre fifo à servir. En choisissant de ne pas faire d'attente active sur le verrou, on permet à un autre canal fifo du coprocesseur d'être servi. On évite ainsi d'introduire un risque d'interbloquage dû à des dépendances artificielles entre les canaux fifo.

Chaque échange entre le coprocesseur et le contrôleur MWMR provoque le transfert d'un ou plusieurs mots. Mais, le contrôleur MWMR ne déclenche pas une lecture ou une écriture pour chaque mot demandé par le coprocesseur. Les

9	8	7	6	5	4	3	2	1	0	Type de registre
0	0	0	0	0	0	0	0	*	*	Soft Reset
0	0	Z	Z	Z	Z	Z	Z	*	*	Reservé quand ZZZZZZ $\neq$ OOOOOO
										Registres de configuration des canaux
0	1	X	X	X	0	0	0	*	*	MWMR_STATE_AD
0	1	X	X	X	0	0	1	*	*	MWMR_OFFSET_AD
0	1	X	X	X	0	1	0	*	*	MWMR_LOCK_AD
0	1	X	X	X	0	1	1	*	*	MWMR_DEPTH
0	1	X	X	X	1	0	0	*	*	MWMR_BASE_AD
0	1	X	X	X	1	0	1	*	*	MWMR_RUNNING
										X X X définit l'index de FIFO :
										- 0 à 3 pour un canal de lecture
										- 4 à 7 pour un canal d'écriture
1	0	X	X	X	*	*	*	*	*	Registres de configuration
										du coprocesseur
										X X X définit l'index du registre
1	1	X	X	X	*	*	*	*	*	registres de statut du coprocesseur
										X X X définit l'index du registre

FIG. 5.6 – L'affectation de la plage d'adresses utilisées par un contrôleur MWMR.

performances seraient bien trop médiocres. Il utilise en fait une petite FIFO matérielle pour chaque canal, qui est rempli de mots en attente d'écriture dans le canal MWMR, ou de mots lus dans ce canal MWMR en attente d'être consommés par le coprocesseur.

La transaction MWMR elle-même s'effectue par paquets d'un certain nombre de mots. Ce nombre est défini par la profondeur de la FIFO associée au canal concerné. Pour un canal de lecture, on ne déclenche de transaction que lorsque le tampon interne du contrôleur MWMR est entièrement vide. Au contraire, pour un canal d'écriture, on ne déclenche de transaction que lorsque le tampon interne est plein. Les transactions sont de la taille maximale : on n'effectue de transfert que si le canal MWMR permet de le faire intégralement. Si le canal MWMR ne permet pas ce transfert, on le libère sans y toucher, pour réessayer plus tard. On peut dire que le contrôleur MWMR effectue des transactions de longueur fixe.

Ceci constitue une contrainte pour le coprocesseur. Le coprocesseur est obligé de transférer des rafales de taille fixe et de faire du remplissage si la quantité de données à transférer est inférieure à la taille d'une rafale.

La figure 5.3 résume les différentes étapes nécessaires à l'utilisation d'un contrôleur MWMR : d'abord sa configuration, effectuée par un processeur. Cette configuration associe chaque port fifo connecté au coprocesseur à un canal MWMR implanté en mémoire, puis Le contrôleur MWMR se comporte comme un initia-

teur sur l'interconnect VCI, ce qui signifie qu'il se comporte comme un contrôleur DMA, fonctionnant en parallèle des autres tâches logicielles.

5.5.4 Surface occupée par le contrôleur MWMR

On a réalisé une vue physique d'une version du contrôleur matériel MWMR. La version que l'on a choisi d'implémenter est celle que l'on trouve sur l'architecture matérielle multi-clusters orientée télécommunications décrite au chapitre 7. On a réalisé le contrôleur MWMR connecté au coprocesseur *Input engine*. Ce coprocesseur utilise trois canaux MWMR, deux en lecture et un écriture.

Les deux fifos matérielles connectées aux ports de lecture ont une profondeur de 16 mots (de 32 bits) chacune. La fifo connectée au port d'écriture a une profondeur de 32 mots.

La réalisation a été effectuée en utilisant les cellules de la bibliothèque *sxlib* de la chaîne de CAO ALLIANCE développée au laboraoire Lip6 [LIP]. Les portes ont été placées et routées en utilisant les outils **seplace** et **seroute** de la chaîne de CAO commerciale *Silicon Ensemble*.

Ce circuit a été routé sans plots car il est destiné à servir d'interface entre un coprocesseur et l'interconnect. Il n'a donc pas de raison d'être conçu hors d'une puce. Le résultat est un circuit de 10 496 000 λ^2, ce qui, en technologie *0,13 micron*, représente une surface de $0,17mm^2$. Ce circuit est représenté sur la figure 5.7.

5.6 Prise en compte des contraintes de temps réel

Cette partie est consacrée aux modifications nécessaires dans les canaux MWMR pour la prise en compte des contraintes posées par l'exécution d'applications temps réel.

5.6.1 Définition du problème

De nombreuses applications sont soumises à des contraintes de temps réel. On appelle contrainte de temps réel le fait d'avoir une durée maximale à ne pas dépasser pour son exécution.

Les contraintes temps réels se divisent en deux catégories : celles qui ne doivent jamais être dépassées et celles qui peuvent l'être dans certaines circonstances.

Pour la première catégorie, on parle de contrainte temps réel dure (hard real time). Dans ce cas, aucun dépassement de la contrainte ne peut être toléré. En

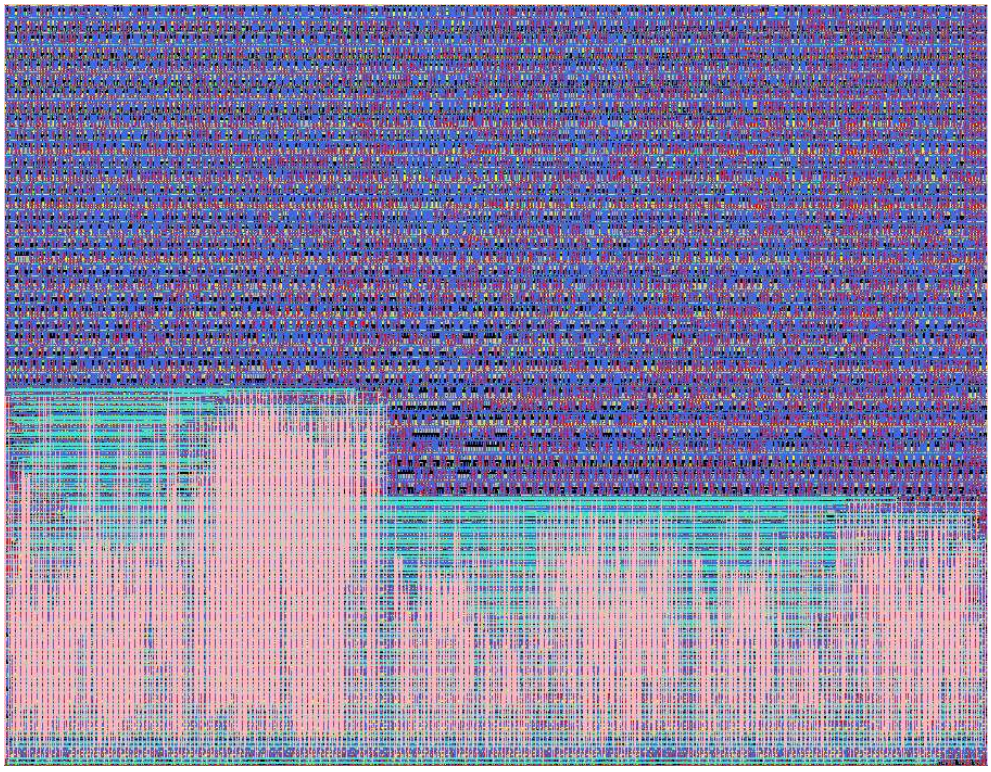

FIG. 5.7 – Vue du contrôleur MWMR avant le routage.

pratique, ce type de contrainte est employé lorsqu'un dépassement amènerait un résultat catastrophique.

Pour la seconde catégorie, on parle de temps réel souple (soft real time). Cette définition englobe toute tâche qui a des contraintes de temps pour s'exécuter, mais qui peut, exceptionnellement, dépasser cette contrainte sans que les conséquences ne soient trop graves pour l'application.

On trouve fréquemment ce type de contraintes dans des applications périodiques telles que le traitement d'un flux vidéo.

Dans le cas du temps réel souple, le dépassement de contrainte peut se produire. Le comportement attendu en cas de dépassement de la contrainte sur un échantillon de données particulier (une image du flux vidéo) consiste à abandonner le traitement en cours, pour pouvoir traiter correctement l'échantillon suivant. Il faut alors, d'une part réinitialiser la tâche qui n'a pas respecté sa contrainte de temps, et d'autre part, vider les canaux de communication auxquels elle est connectée pour les purger des données caduques qu'ils contiennent.

On va voir dans ce chapitre les contraintes imposées par le mode de fonctionnement temps réel souple sur les canaux de communication MWMR.

5.6.2 Mécanisme général

Lorsqu'une date limite est atteinte, une interruption est émise par le dispositif qui l'a détectée. Le gestionnaire de cette interruption envoie des signaux à toutes les tâches concernées. La tâche qui reçoit un signal lui indiquant le dépassement, alors qu'elle n'a pas terminé le traitement en cours exécute alors une séquence de code spéciale destinée à la réinitialiser. Le problème qui se pose alors concerne les canaux de communication qui doivent être réinitialisés.

Le principe est simple : il s'agit de vider chaque fifo concernée des données résiduelles qu'elle contient. Il faut s'assurer qu'aucune tâche pas encore réinitialisée ne vienne y écrire quoi que ce soit.

Cette purge est effectuée par la fonction `fifommrflush`, qui réinitialise une fifo. Comme cette fonction modifie le statut de la fifo, ainsi que les pointeurs `readp` et `writep`, il faut qu'elle prenne le verrou pour le faire. Sinon, on court le risque de rendre le contenu de cette fifo incohérent.

Comme on le voit, cette réinitialisation a besoin de prendre le verrou pour s'effectuer. Il ne faut donc pas qu'une autre tâche garde ce verrou. C'est à dire qu'il ne faut pas qu'une tâche soit déchargée du processeur alors qu'elle possède un verrou. Dans ce cas on court le risque d'un interblocage : une routine d'interruption attend un verrou pour pouvoir faire son traitement et rendre le processeur tandis que la tâche propriétaire du verrou attend d'être réelue pour libérer le verrou.

Ce risque d'interblocage nous conduit à rendre atomique un accès à un canal MWMR : entre la prise et la libération du verrou, les interruptions sont masquées. De ce fait on peut garantir qu'aucune tâche ne sera évincée alors qu'elle possède la propriété exclusive d'un verrou.

5.6.3 Les conséquences sur le matériel

Deux composants matériels sont concernés par la prise en compte des contraintes de temps réel : les coprocesseurs et les contrôleurs MWMR. Comme pour les tâches logicielles le comportement de chaque coprocesseur en cas de dépassement de date limite dépend de l'application. Il n'est donc pas possible d'en présumer ici. Ce qui est sûr, c'est qu'il faut pouvoir signaler à ce coprocesseur que le dépassement à eu lieu. Pour ce faire, on utilise le mécanisme des registres de configuration : l'un deux est utilisé pour déclencher la réinitialisation du coprocesseur. Du côté du contrôleur MWMR, il y a deux choses à faire : bloquer l'utilisation des canaux dès que le signal de dépassement est reçu et jusqu'à ce qu'un signal de réactivation soit reçu, et purger les données encore contenues dans les buffers internes. Le contrôleur MWMR utilise un tampon interne pour chaque canal fifo du coprocesseur. Or en cas de réinitialisation, pour chaque canal, il faut effectuer les actions suivantes :

1) Le contrôleur doit finir la transaction en cours, et ne pas en redémarer une nouvelle avant d'avoir reçu un signal start. La transaction MWMR en cours doit être terminée correctement pour libérer le verrou et éviter un risque d'interblocage.
2) vider le contenu du tampon interne : ses données sont désormais caduques.

Le mécanisme choisi pour cette partie est celui d'un reset retardé : un signal SOFT-RESET est envoyé au contrôleur en cas de dépassement de date limite. Ce signal est enregistré, mais il ne sera traité qu'une fois la transaction MWMR en cours achevée.

Ce mécanisme de reset retardé peut être vu comme le pendant matériel du masquage des interruptions : on doit laisser l'accès en cours à un canal MWMR se terminer avant d'effectuer une remise à zéro.

La conséquence de la prise en compte du temps réel est de rendre les primitives d'accès aux fifomwmr non interruptibles : elles deviennent des sections critiques.

5.7 Analyse des performances

Les expérimentations présentées ici visent à répondre aux deux questions suivantes.

Quelle est la latence minimale (en nombre de cycles du processeur) et quel est le débit maximal (en nombre d'octets transférés par cycle) des fonctions de communication `mwmr_read` et `mwmr_write` ? Pour obtenir des valeurs optimales, on étudiera un scénario sans aucune contention. On analysera plus particulièrement l'impact de la longueur des rafales sur le débit et sur la latence des transactions pour les deux implantations logicielles proposées (cache avec préchargement et invalidation sélective). On effectuera les mêmes mesures de débit et de latence pour les transactions effectuées par le contrôleur matériel MWMR.

L'utilisation de verrous à attente active pour protéger l'accès exclusif simplifie le protocole, mais peut se révéler très coûteux en performances, lorsqu'un grand nombre de tâches sont en compétition pour accéder à un même canal de communication. Comment les performances des communications (latence et débit) dépendent-elles du nombre de tâches ? Combien de tâches peut-on raisonnablement connecter à un même canal ?

5.7.1 Performances hors contention

Dans cette section on évalue les performances des deux impléméntations logicielles des primitives d'accès aux canaux MWMR, ainsi que celles du contrôleur MWMR matériel. Les performances mesurées concernent le nombre de cycles

d'horloge nécessaire pour effectuer une transaction MWMR complète, en fonction de la taille de cette transaction (le nombre de mots transférés).

On s'intéresse aux performances de ces primitives dans le contexte des systèmes embarqués, c'est pourquoi on mesure les temps d'exécution des versions optimisées pour le proceseur MIPS R3000 de ces primitives logicielles. Pour ce qui est du contrôleur matériel, un modèle de simulation précis au bit et au cycle près a été réalisé en langage SystemC. Ce modèle a été intégré dans la plateforme de modélisation Soclib, et c'est cet ensemble qui a servi pour effectuer les mesures du nombre de cycles nécessaire à chaque accès.

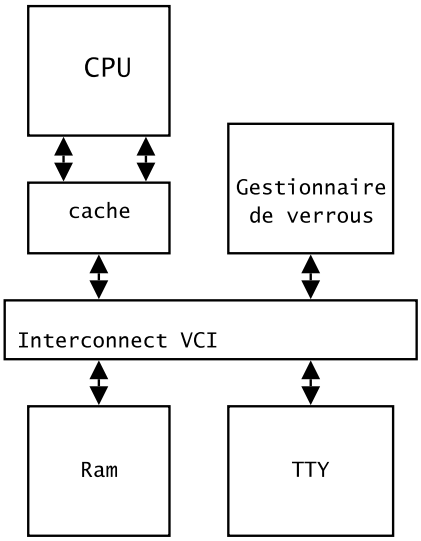

FIG. 5.8 – Architecture mono-processeur utilisée pour les tests de performances à vide des canaux MWMR.

Ces performances sont mesurées sur une plateforme matérielle comportant un seul processeur MIPS R3000. Elle comporte en outre un banc de mémoire, un périphérique gestionnaire de verrous *lock* et un terminal de sortie `TTY` permettant de suivre la progression du déroulement de l'application. Cette architecture mono-processeur est représentée sur la figure 5.8. Cette architecture matérielle exécute une application minimale. Cette application fait successivement des écritures de taille fixe puis des lectures de même taille dans un unique canal MWMR. Le graphe de tâche de cette application est illustré par la figure 5.9.

L'ensemble plateforme + logiciel fonctionne pendant un laps de temps simulé suffisamment long pour effectuer environ 800 écritures et autant de lectures. Les résultats que l'on donne sont donc des moyennes.

Sur les figures 5.10, 5.11 et 5.15 la taille des rafales est exprimée en mots de 32 bits. Les tailles de rafales mesurées varient donc de 1 à 64 mots de 32 bits. Le temps total correspond à la durée totale d'une transaction MWMR ; elle

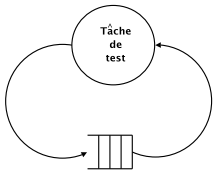

FIG. 5.9 – Graphe de tâche et communication utilisé pour les tests de performances à vide des canaux MWMR.

taille de rafale	read		write	
	Temps total	Période	Temps total	Période
1	324	324	286	286
2	309	154	269	134
4	313	78	271	68
8	323	40	277	35
16	390	24	299	19
32	521	16	342	11
64	808	13	450	7
128	1326	10	602	5

FIG. 5.10 – Les performances en nombre de cycles des primitives `read` et `write` pour l'implémentation utilisant la mémoire prefetch.

taille de rafale	read		write	
	Temps total	Période	Temps total	Période
1	237	237	187	187
2	242	121	186	93
4	249	62	193	48
8	284	36	205	26
16	322	20	234	15
32	378	12	290	9
64	537	8	402	6
100	655	6.5	527	5
128	765	6	628	5

FIG. 5.11 – Les performances en nombre de cycles des primitives `read` et `write` pour l'implémentation utilisant la cohérence par logiciel.

est mesurée en nombre de cycles. La colonne période est le nombre de cycles nécessaire pour transmettre un mot de 32 bits. C'est l'inverse d'un débit.

La figure 5.10 correspond à l'implémentation logicielle utilisant le mécanisme

de pré chargement. Elle nous montre plusieurs points : premièrement, le débit, en lecture comme en écriture, croît sensiblement à mesure que la longueur de la rafale augmente. Ceci est dû au fait que le coût de prise de verrou, lecture de statut et leurs mises à jours prennent un temps fixe, qui est amorti par la longueur de la transaction.

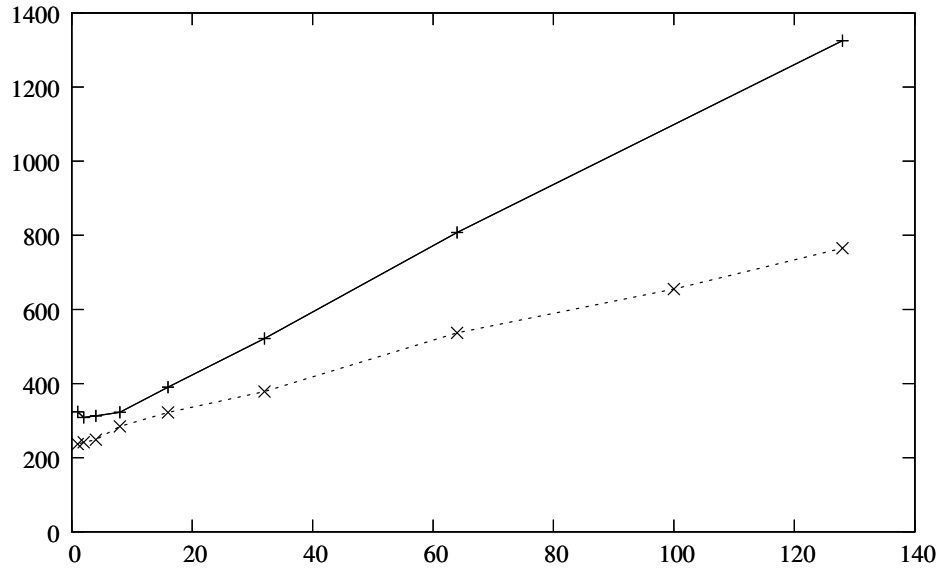

FIG. 5.12 – performances comparées pour les deux implantations prefetch et invalidation des primitives de lecture à vide. L'implantation prefetch est en traits pleins, celle avec invalidation de cache, en pointillés.

On peut ensuite remarquer que les lectures ont une durée supérieure aux écritures, à taille de rafale égale. Cela s'explique par un effet de cache : lors d'une écriture, la donnée que l'on va écrire vient d'être calculée, elle se trouve dans le cache. Au contraire, celle que l'on lit ne s'y trouve pas, ce qui provoque une lecture en mémoire.

La figure 5.11 correspond à l'implémentation logicielle utilisant l'invalidation du cache par logiciel. On y observe les mêmes effets que pour l'implantation prefetch. Le débit augmente quand la longueur des rafales augmente, et les écritures sont plus courtes que les lectures, à longueur de rafale équivalente. Les figures 5.12 et 5.13 montrent l'évolution comparée des primitives de lectures et d'écriture pour les deux implantations prefetch et invalidation. On y voit que la primitive de lecture avec invalidation du cache est nettement plus rapide que son homologue utilisant le mécanisme prefetch, tandis que les primitives d'écriture ont sensiblement la même durée. La raison tient à la meilleure utilisation du cache par la seconde version : les accès aux structures de contrôle de la fifomwmr bénéficient d'accès cachés après la première invalidation.

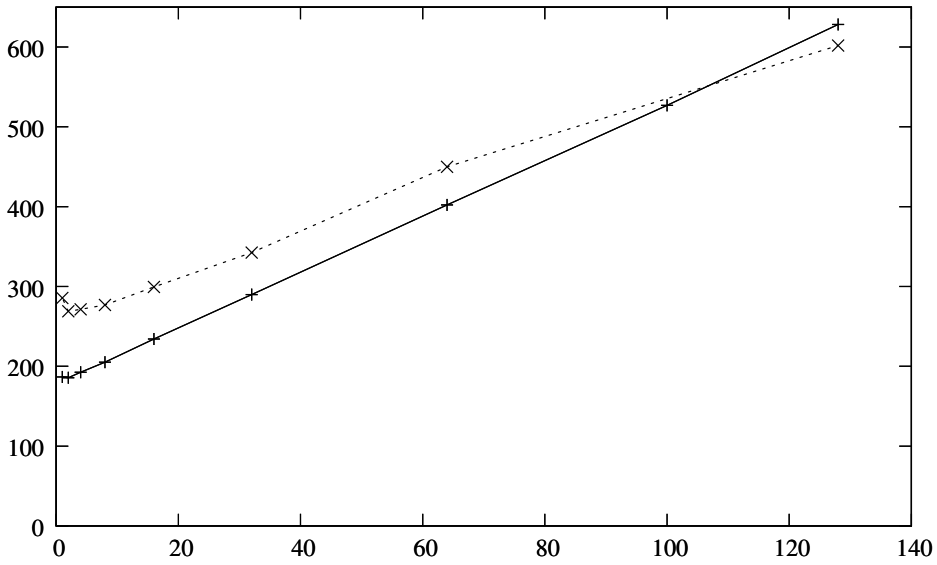

FIG. 5.13 – performances comparées pour les deux implantations prefetch et invalidation des primitives d'écriture à vide. L'implantation prefetch est en pointillés, celle avec invalidation de cache, en traits pleins.

Sur la figure 5.15 sont présentées les performances du contrôleur matériel MWMR. Le protocole expérimental utilise le même graphe de tâche que les expérimentation précédentes. C'est le graphe représenté sur la figure 5.9 : une tâche qui effectue successivement des écritures puis des lectures de longueur fixe dans un unique canal MWMR. La différence est que la tâche est maintenant implantée en matériel dans un coprocesseur qui effectue ses transferts par le biais d'un contrôleur MWMR. L'architecture utilisée est décrite sur la figure 5.14. Cette architecture comporte également un processeur ; il est utilisé pour effectuer la configuration du contrôleur MWMR.

Comme pour les versions logicielles des primitives, le temps moyen nécessaire au transfert d'un mot de 32 bits est plus court à mesure que la longueur de la rafale augmente. On voit également que le temps de la transaction est une fonction affine de la longueur de rafale : $Temps = 151 + rafale$. L'ordonnée à l'origine, 151 cycles, dépend du protocole MWMR (prise de verrou, lecture du statut, etc), ainsi que de la latence de notre réseau d'interconnexion. Pour cette expérience, celle-ci était fixée à 12 cycles. C'est le coût à vide d'une communication unidirectionnelle. Une transaction VCI-OCP coûte donc au moins 24 cycles : 12 cycles pour acheminer le paquet requête et 12 cycles pour le paquet réponse.

En conclusion, on peut dire que ces implantations des canaux de communication MWMR et de leurs primitives d'accès permettent d'utiliser plus de 10% de la bande passante maximale du réseau pour les versions logicielles, et jusqu'au tiers

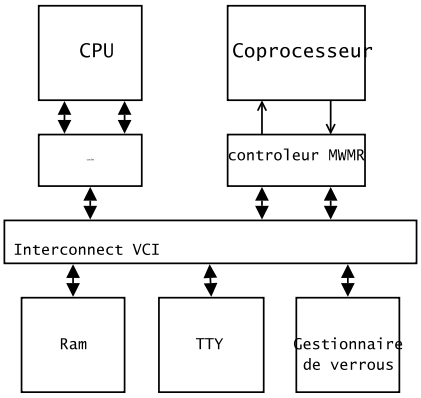

FIG. 5.14 – Architecture mono processeur, avec un coprocesseur utilisée pour les test de performances à vide du contrôleur MWMR.

taille de rafale	read		write	
	Temps total	Période	Temps total	Période
1	152	152	152	152
2	153	76	153	76
4	155	38	155	38
8	159	19	159	19
16	167	10	167	10
32	183	5.7	183	5.7
64	215	3.4	215	3.4

FIG. 5.15 – Les performances du contrôleur MWMR pour les accès `read` et `write` aux canaux MWMR.

de cette bande passante dans le cas d'une version matérielle.

La définition du protocole MWMR, ainsi que la caractérisation de ces implémentations ont fait l'objet d'une publication lors de la conférence RECOSOC 2006 [FGG06].

5.7.2 La résistance à la charge des canaux MWMR

Ici on cherche à déterminer le nombre maximal de tâches que l'on peut connecter à un canal MWMR sans aller jusqu'à saturation. Par saturation, on entend que le temps d'attente dans la boucle d'accès au `spin_lock` devient prohibitif. La capacité d'un canal MWMR à répondre à une requête, qu'il s'agisse d'une lecture ou d'une écriture, dépend du nombre de tâches qui essaient d'accéder au verrou au même instant. Or ce nombre de tâches en concurence à un instant donné est fonc-

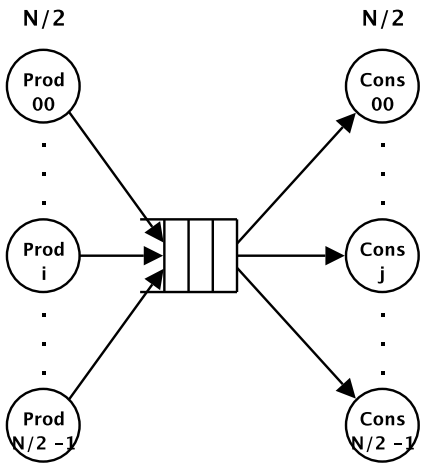

FIG. 5.16 – Graphe de tâches utilisé pour le test de charge des canaux MWMR. Ce graphe comporte 1 canal MWMR auquel sont connectées N/2 tâches productrices et N/ 2 tâches consommatrices.

tion de deux paramètres principaux : le nombre total de tâches connectées à ce canal, et la fréquence de leur accès à ce canal. Ce dernier paramètre peut également s'énoncer comme étant le rapport entre le temps utilisé pour les communications et le temps de calcul de chaque tâche. On définit donc un protocole expérimental au cours duquel on fait varier ces deux paramètres. Le premier est le nombre de tâches logicielles connectées au canal MWMR. Ces tâches se répartissent en deux sous-ensembles de même taille : les tâches productrices et les tâches consommatrices. Le second paramètre est matérialisé par une variable *d* qui représente le nombre de tours effectués dans une boucle de temporisation entre deux accès au canal de communication. Chaque tour de boucle dure 10 cycles. La figure 5.16 représente le graphe de tâches utilisé pour cette expérience, tandis que la figure 5.17 illustre l'architecture matérielle sur laquelle ces tâches sont déployées. Le déploiement consiste à mettre une tâche par processeur. Il y a autant de bancs de mémoire qu'il y a de processeurs. La raison de l'utilisation de cette architecture *idéale* est d'éviter d'introduire accidentellement d'autres points de contention que celui que l'on compte observer. On utilise donc une architecture dotée d'une bande passante énorme pour que les transferts entre les processeurs et la mémoire autres que ceux des accès MWMR ne créent pas de contentions d'accès aux ressources.

Les mesures effectuées sont regroupées dans les 3 courbes de la figure 5.18. En abscisse, on a le nombre de tâches connectées au canal MWMR, en ordonnée on a le rapport entre le temps passé dans la fonction `spin_lock` et le temps total d'exécution de l'application. C'est ce temps passé dans la fonction spin_lock qui correspond au temps *perdu* par l'application. Ces trois courbes ont été tracées à partir de mesures faites pour différentes valeurs du nombre de tours d'attente ef-

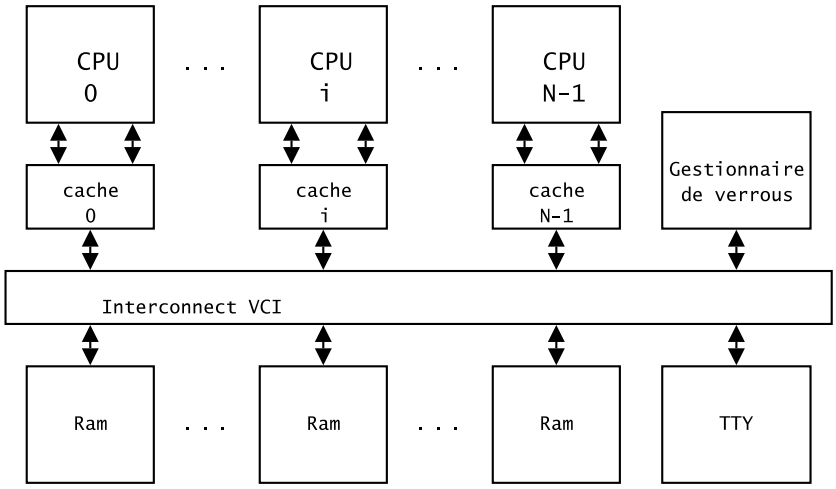

FIG. 5.17 – Architecture multi-processeur, multi-bancs de mémoire utilisée pour le test de charge

fectués entre deux communications : 1000, 20 000 et 50 000 cycles respectivement pour les trois courbes à gauche, à droite et en bas de la figure 5.18. La variation de ce paramètre permet de simuler différents rapports entre le temps de calcul de la tâche et son temps de communication. Sur ces 3 courbes, on voit que le temps passé dans la fonction `spin_lock` augmente avec le nombre de tâches connectées. Lorsqu'il y a peu de tâches, ce rapport est presque nul (moins de 1% dans tous les cas pour 4 tâches). Il correspond au temps d'exécution de cette fonction sans attente. Pour ces trois courbes, on voit que le comportement est le même : lorsqu'il y a peu de tâches connectées au canal MWMR, la durée des accès au verrou demeure modeste. Par contre, lorsqu'il y a contention pour l'accès au verrou ce rapport tend à s'accroitre considérablement. En fonction de la fréquence des communications, la saturation du canal MWMR intervient pour un nombre plus ou moins important de tâches connectées. Pour un temps de calcul de 1000 cycles, la saturation intervient pour un nombre de tâche compris entre 15 et 20. En revanche pour un temps de calcul de 20 000 cycles, le canal MWMR n'est saturé qu'à partir d'une quarantaine de tâches. Et il faut environ 70 tâches pour saturer le canal lorsque le temps de calcul est de 50 000 cycles entre deux communications. Le seuil de saturation d'un canal est le nombre de tâches à partir duquel le temps d'attente du verrou devient trop pénalisant. Lorsqu'il atteint 50% du temps d'exécution, cela signifie que chaque processeur passe en moyenne un cycle sur deux à attendre le verrou associé au canal MWMR. Quand ce seuil est franchi, on peut dire qu'on a dépassé les limites de parallélisation de l'application. Cette limite de parallélisation est atteinte lorsque l'ajout d'une tâche pour exécuter l'application n'accélère plus la vitesse de celle ci mais, soit n'apporte rien, soit même ralen-

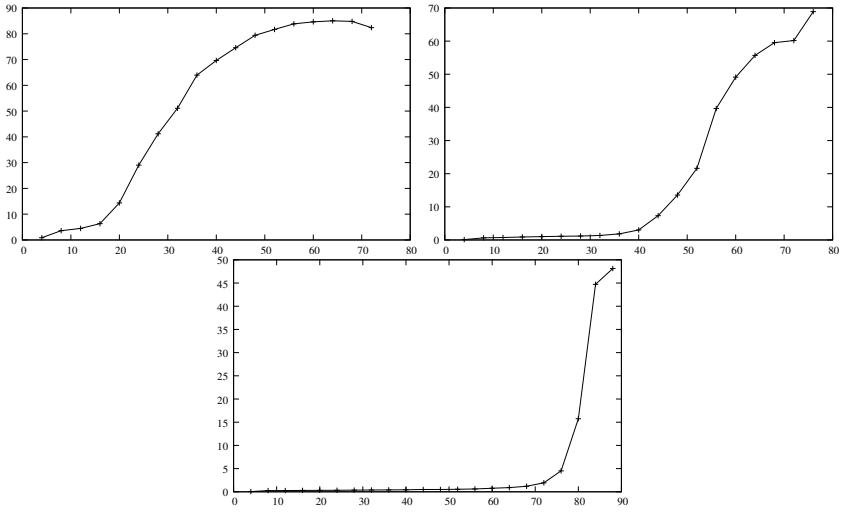

FIG. 5.18 – Pourcentage de temps passé en attente active du verrou, en fonction du nombre de tâches connectées au canal MWMR. Pour 1000 (à gauche), 20 000 (à droite) et 50 000 (en bas) cycles de calcul entre deux communications.

tit l'application. La mesure de cet optimum peut-être évaluée grâce à la formule suivante : $N_{optim} = MAX(N_{taches} * (100 - pourcentage_{spin\_lock})/100)$. Ceci donne le nombre de processeurs qui exécutent effectivement l'application. La différence entre le nombre de processeurs présents sur la puce et ceux exécutant l'application est le nombre de processeurs en attente du verrou. Tant que le nombre de processeurs en train d'exécuter l'application augmente, on peut considérer que l'ajout de matériel est justifié puisqu'il apporte un gain en vitesse d'exécution.

Les courbes de la figure 5.19 montrent ce nombre de processeurs effectivement en train d'exécuter l'application en fonction du nombre total de processeurs sur la puce, et ce pour les trois séries de mesures déjà effectuées. On remarque que cette courbe s'apparente d'abord à une droite. Chaque processeur ajouté apportant presque toute sa capacité de calcul au traitement de l'application. Autour du seuil de saturation, la courbe s'aplatit, à mesure que les nouveaux processeurs se retrouvent en concurence pour l'accès au canal MWMR. Cet aplatissement, suivi du rapide effondrement correspond à l'augmentation de la part de la fonction *spin_lock* dans l'exécution de l'application. le maximum de chacune des courbes indique le nombre maximal de tâches pouvant se partager le canal MWMR sans se gêner les unes les autres. On remarque que ce maximum est plus élevé si le temps de traitement est long.

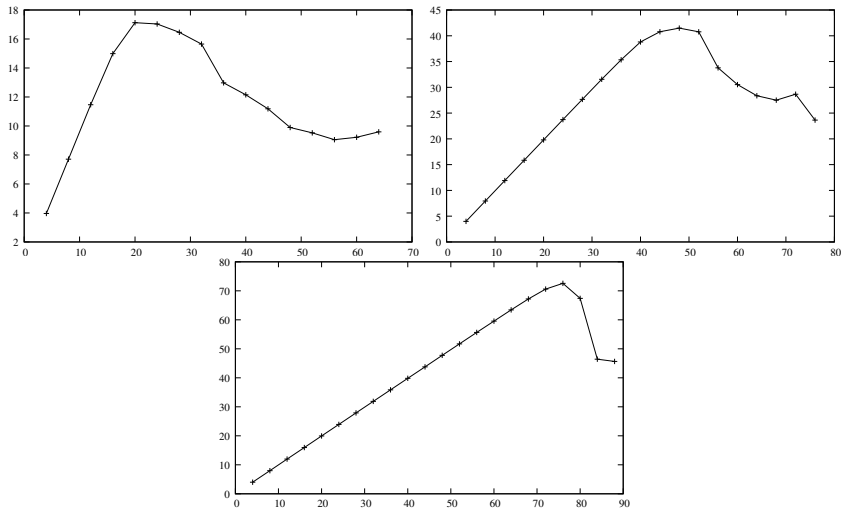

FIG. 5.19 – Nombre de processeurs disponibles pour l'exécution de l'application pour 1000 (à gauche), 20 000 (à droite) et 50 000 (en bas) cycles de calcul entre deux communications, en fonction du nombre total de processeurs.

5.8 Conclusion

Les canaux MWMR sont des structures logicielles placées dans la mémoire du système. Chacun d'eux est protégé par un verrou unique. Ce verrou garantit la cohérence des données. Chaque canal peut avoir plusieurs écrivains et lecteurs, dont la nature matérielle ou logicielle n'est, a priori, pas connue.

Pour accéder à ces canaux, on a développé des primitives de lectures et d'écritures, chacune d'elles ayant fait l'objet de plusieurs implantations. Ces différentes implantations avaient pour objet d'apporter une réponse au problème posé par la cohérence des caches. Les deux implantations nécessitent un concours du matériel (segments typés par le cache dans le premier cas et instruction d'invalidation de ligne de cache pour le second cas). Les mesures de performances effectuées sur ces implantations montrent que l'implantation utilisant l'invalidation explicite de lignes de caches par le processeur est plus efficace en toutes circonstances. En conséquence, c'est la seule pour laquelle nous avons caractérisé la résistance à la charge. La résistance à la charge est le nombre maximal de tâches que l'on peut connecter à un canal MWMR avant que le temps d'accès au verrou associé ne deviennent trop important. Cette notion n'est pas facile à mettre en évidence, car elle ne dépend pas que du nombre de tâches connectées à un canal MWMR, mais aussi de la fréquence à laquelle elles effectuent leurs communications avec ce canal. En conséquence, on a effectué cette mesure de résistance à la charge pour des durées croissantes de calcul dans les tâches. De ces mesures, il ressort que la saturation du canal MWMR intervient pour un nombre de tâches qui lui sont

connectées d'autant plus important que le temps de traitement entre deux communications est long. On a également pu établir une formule permettent de mesurer le nombre maximal de tâches pouvant se partager un canal MWMR à partir de ces mesures de saturation.

Chapitre 6

Les canaux MWMR pour implanter des graphes de Kahn

Sommaire

Dans ce chapitre, on va montrer que les canaux MWMR sont efficaces pour implanter les graphes de tâches qui respectent le formalisme de Kahn. On sait que les canaux MWMR de par leurs spécifications peuvent être utilisés pour implanter les communications de ces graphes, ce que l'on souhaite mesurer c'est leur efficacité dans cette tâche.

Pour mesurer cette efficacité, on a recours à une comparaison : on utilise une application décrite comme un graphe de Kahn, utilisant des canaux ad hoc, et on remplace ces canaux par des canaux MWMR. On compare les temps d'exécution de l'application pour ces deux types de canaux.

Cette application est le décodeur d'image JPEG.

6.1 Le codage JPEG

Le format de fichier JPEG est un format de compression avec perte utilisé pour la compression d'images. Il permet de réduire de 90% la taille nécessaire pour stocker une image, avec une perte de qualité minime. Le codage JPEG repose sur la compression par la méthode de Huffman et la représentation en fréquences de

l'image. Pour effectuer la compression, l'image est divisée en blocs carrés de 8 pixels de coté. Les données de l'image sont toujours traitées par paquets de 64 pixels.

La compression d'une image au format JPEG est effectuée en trois étapes :

La première calcule la transformée en cosinus discrète (DCT) qui permet d'obtenir une représentation en fréquences de l'image.

La seconde est une étape de seuillage. Les fréquences les plus hautes de l'image, auxquelles l'oeil est peu sensible sont filtrées. C'est cette étape qui provoque une perte de qualité. La valeur des seuils utilisés détermine la qualité de l'image encodée : des seuils bas permettent de conserver beaucoup des qualités de l'image initiale, tandis que des seuils élevés permettent un taux de compression important, au prix d'une perte de qualité sensible.

La troisième étape est le codage de Huffman. L'image est compressée en utilisant une table de transcodage entropique (table de Huffman). Ce codage est réalisé de manière à ce que les séquences les plus utilisées soient codées avec le minimum de bits possible.

Une image JPEG est composée de trois parties, séparées par des marqueurs particuliers appelés balises :

La description de l'image : sa taille, le nombre de tables utilisées, l'auteur.

Les tables de Huffman utilisées pour la compression, ainsi que les valeurs des seuils utilisés pour le filtrage, qui font partie du flux JPEG.

Les données compressées représentant l'image.

Le décodage d'une image JPEG consiste à effectuer l'inverse de ces trois étapes dans l'ordre inverse :

Le décodage de Huffman (VLD Variable Length Decoder), puis

le seuillage inverse (IQ pour Inverse Quantization), enfin

La transformée en cosinus inverse (IDCT Inverse Discrete Cosinus Transformation)

6.2 Programmation parallèle du décodage JPEG

Dans cette section on décrit les spécifications d'une application parallèle multitâche réalisant le décodage JPEG. On a vu que le décodage d'une image JPEG nécessitait trois étapes. Chacune de ces étapes est réalisée par une tâche dediée. Ceci constitue un pipeline à trois étages. Les communications entre ces tâches sont réalisées en utilisant des canaux FIFO.

On a donc trois tâches qui effectuent le décodage JPEG :

La tâche VLD qui effectue le décodage de Huffman

La tâche IQ pour le seuillage inverse

La tâche idct qui réalise la transformée en cosinus inverse.

Une image JPEG est composée de plusieurs parties, et ces différentes parties sont séparées par des balises. On a besoin d'une tâche supplémentaire pour identifier ces balises et envoyer les différentes composantes du flux JPEG aux tâches concernées : tables de Huffman et données compressées à la tâche VLD, table de seuillage à la tâche IDCT. Cette nouvelle tâche, chargée du démultiplexage du flux JPEG est appelée DEMUX.

On a vu que lors de la décompression JPEG, l'image était manipulée par blocs de $8 * 8$ pixels, or l'affichage d'une image est effectué ligne par ligne. On a donc besoin d'une nouvelle tâche chargée de réordonner les pixels pour former des lignes à partir des blocs reçus. Cette tâche est appelée LIBU (LIne BUilder).

La spécification comprend en outre deux tâches chargées des entrées sorties du système : une tâche TG (Traffic Generator) qui lit le ficher d'entrée JPEG et le copie dans la mémoire du système, et une tâche RAMDAC (RAM Digital Analog Converter) qui effectue l'affichage de l'image une fois celle-ci décompressée.

La spécification parallèle de l'application de décodage JPEG comprend donc au total sept tâches. Elle est illustrée par la figure 6.1. Sur cette figure, on voit le pipeline des sept tâches, ainsi que les canaux de comunications qui les relient. Les première et dernière tâches, qui réalisent les entrées/sorties, sont toutes les deux implantées par des coprocesseurs matériels spécialisés. Les cinq autres tâches sont réalisées en logiciel. La tâche DEMUX a deux canaux de communication vers la tâche VLD, ceci pour séparer les différents types de données qui sont envoyées : les tables de Huffman et les images compressées. De même, la tâche IQZZ reçoit ses données de deux canaux : par le canal connecté à VLD arrivent les blocs compressés de l'image, par le canal connecté à DEMUX arrivent les tables de seuillage.

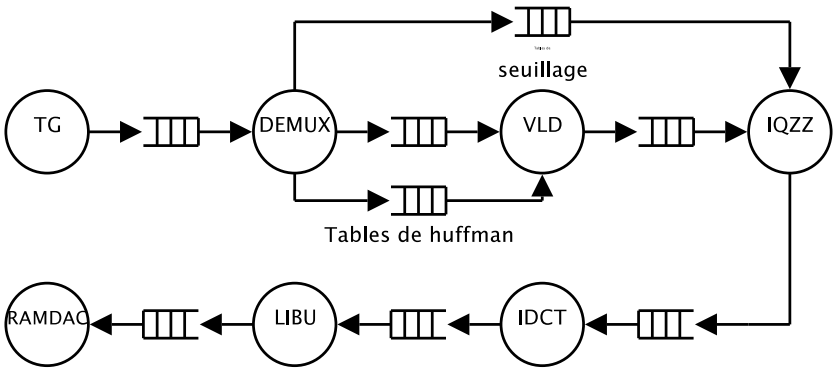

FIG. 6.1 – Graphe de tâches des spécifications du décodeur JPEG

6.3 Les canaux de communications : objet de la comparaison

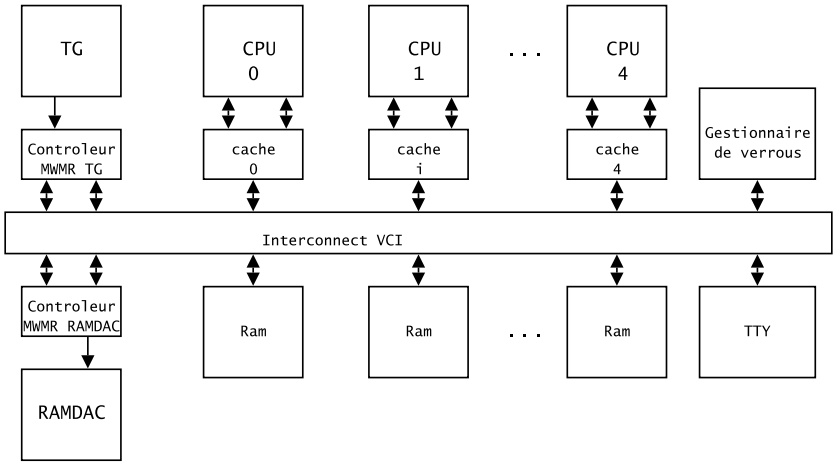

FIG. 6.2 – L'architecture matérielle utilisée pour l'exécution du décodeur parallèle JPEG

Dans cette spécification, on voit que toutes les communications se font point à point entre deux tâches. Il est de plus interdit de jeter un paquet si le canal devant le recevoir est plein. Il faut au contraire attendre que celui-ci se vide. Les communications sont donc bloquantes et on voit par là que ce graphe de tâche respecte les contraintes nécessaires pour être un graphe Kahn. C'est ce qui nous intéresse ici. Le but de cette expérience est de montrer que les canaux MWMR sont également efficaces pour implémenter un graphe de Kahn.

Pour confirmer nos dires, on exécute l'application parallèle du décodeur MJPEG spécifiée ci-dessus avec deux types de canaux de communication. D'abord avec les canaux DPN (Disydent Process Network) issus de l'environnement Disydent. Ces canaux de communications ont été conçus et optimisés pour des graphes de Kahn. On exécute ensuite la même application en remplaçant les canaux DPN par des canaux MWMR.

La différence principale entre les deux types de canaux de communications porte sur le type de verrou utilisé. On a vu que les canaux MWMR utilisaient des verrous à attente active *spin_lock*. En revanche, les canaux DPN utilisent des verrous à exclusion mutuelle de type Mutex. La particularité des verrous Mutex est qu'une tâche qui cherche à acquérir le Mutex lorsque celui-ci est indisponible est endormie. Elle n'est réveillée, c'est à dire ne redevient éligible pour l'exécution, que lorsque le verrou est libéré. À ce moment, toutes les tâches en attente du verrou se retrouvent en concurrence pour y accéder, une tâche l'obtient, les autres sont rendormies. Dans le cas des canaux, les Mutex ne sont partagés qu'entre deux

tâches, une tâche productrice et une tâche consommatrice.

Une autre différence entre les deux types de canaux concerne le fonctionnement des primitives de lectures et d'écriture lorsque le contenu de la FIFO ne permet pas d'exécuter complètement le transfert demandé. Dans les canaux DPN, les tâches s'endorment en demandant à être réveillées lorsque le statut de la FIFO DPN a été modifié. Dans les canaux MWMR, lorsqu'on effectue un accès bloquant, la tâche se décharge du processeur si elle ne peut pas faire la totalité du transfert prévu, mais elle reste éligible. L'algorithme utilisé par les primitives d'accès bloquantes aux canaux MWMR est celui présenté au paragraphe 5.1.5, tandis que celui utilisé par les canaux DPN a été détaillé au paragraphe 3.7.3.

On peut supposer que ces différences entre les deux types de verrous va jouer en faveur des canaux DPN lorsqu'on a plusieurs tâches sur le même processeur. Dans ce cas, il semble plus efficace de ne pas élire une tâche dont on est sûr qu'elle ne dispose pas de données pour travailler. Au contraire, lorsqu'on dispose d'autant de processeurs qu'il y a de tâches à exécuter, l'utilisation d'un mécanisme de synchronisation complexe, les Mutex, sera pénalisant : on n'a aucune tâche à élire lorsqu'on endort la tâche en attente, et le processeur reste donc inactif pendant ce temps d'attente.

On fait deux séries de mesures. Les deux portent sur le nombre de cycles nécessaires pour décoder dix images de 48 * 48 pixels. On compte dans ce temps de décodage le temps nécessaire au démarrage et à l'initialisation du système.

Pour la première série, on utilise les canaux MWMR, pour la seconde, on utilise les canaux DPN de Disydent. L'architecture matérielle utilisée est illustrée sur la figure 6.2. Sur cette figure, on voit les deux coprocesseurs TG et RAMDAC connectés au système par des contrôleurs MWMR. Ces contrôleurs MWMR restent les mêmes pour les deux séries de mesures. La mesure ne porte donc que sur les canaux utilisés par des tâches logicielles.

Pour chaque série de mesures, on fait varier le nombre de processeurs utilisés sur la plateforme, et donc le déploiement de l'application parallèle sur la plateforme matérielle. On utilise successivement un, puis deux et enfin cinq processeurs.

Lorsqu'on ne dispose que d'un unique processeur, celui-ci exécute les 5 tâches logicielles.

Avec deux processeurs, on partitionne les tâches en deux groupes, en essayant de répartir la charge entre les deux processeurs. Il s'avère que dans notre application parallèle, la tâche IDCT prend autant de temps que les quatre autres tâches logicielles réunies. Le partitionnement est donc, sur un processeur la tâche IDCT, les tâches DEMUX, VLD, IQZZ et LIBU sur l'autre. Pour ne pas introduire de problème de cohérence de cache, on interdit la migration de tâche.

Avec les cinq processeurs, on place une tâche par processeur.

Le tableau 6.3 donne les temps d'exécution du décodage JPEG, pour les deux

NB CPU	1	2	5
Canaux DPN	35 343 000	26 487 583	17 870 622
Canaux MWMR	26 567 973	19 682 116	11 273 471
Surcoût lié aux canaux DPN	29%	33%	63%

FIG. 6.3 – Durée d'exécution du decodage de 10 images JPEG en fonction du type de canaux et du nombre de processeur utilisés

types de canaux, DPN et MWMR, et pour les trois cas d'utilisation des processeurs, un , deux et cinq.

Dans tous les cas, les canaux MWMR s'avèrent plus performants que les canaux DPN. Le bénéfice attendu d'élire la bonne tâche lors d'un changement de contexte pendant l'exécution sur un seul processeur ne suffit pas à compenser le surcoût lié à la gestion des Mutex et à la signalisation associée. Ceci vient de ce qu'un changement de contexte prend environ 300 cycles, et que le surcoût d'une communication DPN par rapport à une fifo MWMR est de 700 cycles environ. De ce fait, avec 5 tâches fonctionnant sur le même processeur, on a le choix parmi 4 tâches lorsqu'on change de contexte. Ce qui fait que l'on fait au maximum 3 mauvais choix avant de charger la tâche qui peut s'exécuter. Ces trois mauvais choix nous ont coûté 3 * 300 = 900 cycles, soit pas beaucoup plus que le surcoût lié à une utilisation d'un canal DPN. Lorsque le nombre de tâches par processeur diminue (deux processeurs, ou cinq processeurs), l'avantage en faveur des canaux MWMR est encore plus marqué.

6.4 L'influence de la profondeur des fifos

Le nombre d'accès réussis aux canaux de communication dépend aussi de la profondeur de ceux-ci. Si chaque canal ne peut contenir que peu de données, les tâches seront plus souvent bloquées pour cause de canaux pleins ou vides. Pour notre première série de tests, la taille des canaux de communication avait sans doute peu d'influence puisqu'ils faisaient 1024 mots chacun, soit 4Ko par canal. C'est beaucoup. On essaie donc de réduire cette taille, et on mesure l'influence que cela a sur le temps nécessaire au décodage.

Les deux tableaux 6.4 et 6.5 présentent les résultats des durées d'exécution du décodage JPEG de dix images en fonction de la taille et du type des canaux utilisés.

La taille minimale des canaux doit être de 64 mots de 32 bits car c'est la taille des items qui sortent de la tâche IQZZ.

On voit que la réduction de la profondeur des canaux de communication a pour première conséquence d'augmenter le temps de traitement de l'application.

NB CPU	1	2	5
fifo MWMR 1024 mots	26 567 973	19 682 116	11 273 471
fifo MWMR 128 mots	28 984 049	21 528 502	10 120 618
fifo MWMR 64 mots	31 589 538	23 447 556	9 975 284

FIG. 6.4 – Durée d'exécution du décodage de dix images JPEG en utilisant les canaux MWMR, pour trois tailles de ceux-ci, et pour trois cas d'utilisation des processeurs.

NB CPU	1	2	5
fifo DPN 1024 mots	35 343 000	26 487 583	17 870 622
fifo DPN 128 mots	40 474 264	30 210 580	19 276 764
fifo DPN 64 mots	45 860 146	34 017 271	19 893 882

FIG. 6.5 – Durée d'exécution du décodage de dix images JPEG en utilisant les canaux DPN, pour trois tailles de ceux-ci, et pour trois cas d'utilisation des processeurs.

Cet allongement est la conséquence des accès infructueux faits dans ces canaux. Lorsqu'une tâche essaie de lire dans un canal vide ou d'écrire dans un canal plein, elle est déchargée du processeur. Ce déchargement provoque l'appel de la fonction d'élection d'une nouvelle tâche. Si la liste des tâches éligibles est non vide, une nouvelle tâche est élue, sinon, le processeur reste en attente. Lorsqu'une tâche est élue sur le processeur, elle y demeure jusqu'à ce qu'elle soit bloquée sur une opération d'accès à un canal de communication. En pratique, soit ses canaux d'entrées sont vides, soit ceux de sorties sont pleins. Avec ce mode de fonctionnement, des fifos de grandes profondeurs ne provoquent que peu de changement de tâche élue. Au contraire, des fifo de faible profondeur forcent à souvent élire une nouvelle tâche. C'est le surcoût lié aux changement de tâches qui explique la différence de temps d'exécution entre les différentes versions de l'applications où seule la profondeur des fifo change.

Dans tous les cas, les canaux MWMR sont plus efficaces que les canaux DPN.

6.5 Conclusion

Dans ce chapitre, on a montré que les canaux MWMR peuvent être utilisés pour implanter des graphes de Kahn. De par leur construction, on savait que les canaux MWMR pouvaient réaliser des graphes de Kahn, mais nous avons souhaité comparer les performances obtenues avec MWMR aux performances d'une implantation existante respectant la sémantique des réseaux de Kahn.

La bibliothèque des canaux de communication DPN, fournie par l'environnement Disydent utilise des verrous de synchronisation de type Mutex : une tâche bloquée sur un accès à un canal de communication est endormie et ne redevient

éligible que lorsque la condition de bloquage disparait. Il ressort de ce test que les canaux MWMR sont plus efficaces que les canaux DPN ayant servi de référence. Le gain est de l'ordre de 60% lorsqu'on n'a qu'une tâche par processeur, ce qui est le cas le plus favorable aux canaux MWMR, et il est aussi de presque 30% lorsque toutes les tâches fonctionnent sur le même processeur, ce qui est le cas le plus favorable aux canaux DPN. Ce dernier résultat est tout à fait intéressant, puisqu'il montre que les canaux MWMR peuvent être utilisés pour implanter des graphes de Kahn sans perte de performances.

En conclusion, et bien que les canaux MWMR aient été conçus pour répondre aux besoins des applications télécoms, cette bibliothèque de fonctions de communication se révèle également très efficace pour déployer des applications de type vidéo ou multi-média respectant la sémantique KPN.

Chapitre 7

Application Télécommunication

Sommaire

On décrit dans ce chapitre une application télécom de classification et d'ordonnancement de paquets IP. Cette application est implantée sous la forme d'un graphe de tâches parallèles et elle est déployée sur une plateforme matérielle multi processeurs qui est également décrite en détails.

L'objet de ce chapitre est d'étudier les performances de cette application, en terme de débit sortant, et de voir comment ces performances évoluent en fonction du placement des tâches de l'application sur les ressources de la plateforme matérielle (ses processeurs).

7.1 L'application logicielle

L'application logicielle a pour fonction de lire des paquets sur un canal d'entrée et de les sortir sur son canal de sortie. Cette application est composée de deux parties : une première qui effectue la classification des paquets lus sur les entrées. Et une deuxième partie qui effectue l'ordonnancement des paquets en fonction de listes de priorités pré-établies. L'architecture de cette application est illustrée par la figure 7.1. Sur cette figure, on a représenté 5 fifos de priorité. Ces fifo correspondent à différents contrats de qualité de service que l'ordonnanceur doit remplir.

On a déjà décrit la plupart des spécifications fonctionnelles de cette application au paragraphe 4.3. Dans cette partie, on détaille plus avant ces spécifications, et on donne quelques indications sur l'implémentation.

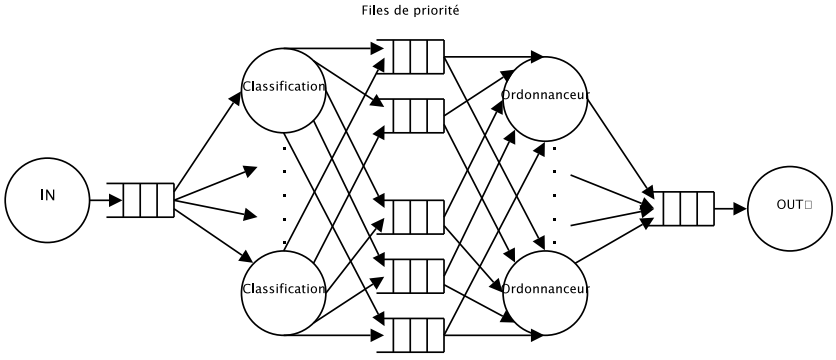

FIG. 7.1 – Schéma de l'architecture logicielle, avec les deux types de tâches séparés et 5 fifos de priorité

7.1.1 La tâche de classification

La tâche de classification est celle qui lit les paquets depuis les canaux d'entrée et qui, pour chaque paquet, détermine son niveau de priorité.

En plus de cette opération de classification, cette application effectue également des vérifications sur l'intégrité des paquets. Ceci pour éviter de propager des paquets endommagés. La suite séquentielle des opérations effectuées par la tâche de classification est la suivante :

On teste ensuite la validité de l'en-tête IP. On fait ceci en vérifiant la validité des bits de checksum, ou CRC (Contrôle de Redondance Cyclique) contenus dans l'en-tête du paquet. Pour effectuer cette vérification, on recalcule ce cheksum et on compare le résultat obtenu avec la valeur fournie dans l'en-tête. Une erreur à ce niveau témoigne d'une erreur lors de la transmission du paquet et son contenu est désormais considéré comme incohérent. Pour cette raison, le paquet est jeté sans qu'aucune signalisation d'erreur ne soit émise, l'adresse source du paquet n'étant pas considérée comme valide.

On examine ensuite l'adresse IP de destination du paquet, car c'est sur cette partie que porte la classification de nos différents flux.

On compare pour cela le préfixe de l'adresse (l'adresse du réseau de destination) avec les adresses de réseaux contenues dans les tables des flux. Ces tables de flux peuvent être volumineuses et porter sur de nombreux autres critères que l'adresse IP de destination des paquets. Dans notre cas, les tables de classification n'excèdent pas quelques dizaines d'entrées, et la recherche de l'appartenance d'un paquet à un flux est faite en parcourant itérativement la liste des flux.

Cette application de classification distingue 12 flux différents : 11 flux correspondent à un critère de sélection : l'adresse IP de destination correspond à tel ou tel masque de réseau, et chacun de ces flux se voit attribuer une fifo de priorité. Le douzième flux est le flux par défaut : il regroupe tous les paquets IP qui ne sont à destination d'aucun des douze sous-réseaux mentionnés. Ce dernier flux est en fait le résultat du multiplexage de tous les flux qui ne font l'objet d'aucun contrat de qualité de service.

On jette ensuite les paquets dont l'adresse source est une adresse de réseau. Ce type d'adresse est interdit pour une adresse source.

On met à jour l'adresse source des paquets fabriqués par l'application (les paquets d'erreur et les paquets ARP Queries) : il s'agit de l'adresse IP du port de sortie.

On décrémente ensuite le champ IP TTL (Time To Live). Si celui-ci devient nul, alors cela signifie que le paquet a atteint sa limite de vie. Chaque dispositif que traverse un paquet décrémente ce champ TTL. Lorsqu'il atteint 0, cela signifie que le paquet circule depuis trop longtemps sur le réseau sans parvenir à atteindre

sa destination. Le protocole RIP (Routing Information Protocol)[Mal98] utilisé par les routeurs pour s'informer entre eux des distances les séparant considère que le nombre maximal de sauts nécessaires entre deux routeurs est 15. Donc, un paquet IP peut atteindre sa destination normalement n'importe où sur Internet en 15 sauts au maximum. Le champ TTL des paquets est stocké sur 8 bits, sa valeur maximale est donc de 255. Son initialisation est faite par l'émetteur du paquet. Cette décrémentation constitue une modification de l'en-tête du paquet, par conséquent, il faut recalculer le checksum de cet en-tête.

La dernière vérification effectuée par cette tâche de classification concerne la taille du paquet : la taille maximale autorisée par le protocole Ethernet est de 1500 octets + 14 octets d'en-tête Ethernet. Les paquets IP d'une taille supérieure doivent être fragmentés. Notre tâche n'effectue pas cette fragmentation elle-même. A la place elle renvoie un message d'erreur vers l'émetteur du paquet fautif.

7.1.2 Mise à jour des tables de routage

Les tables de routages sont des tables associatives qui associent une interface de sortie à chaque adresse de réseau. Ces tables sont potentiellement très grandes (plusieurs dizaines de milliers d'entrées selon la recommandation RFC1812).

Les routeurs se communiquent entre eux des informations sur l'état du réseau. Ces informations conduisent parfois le routeur à modifier ses tables de routage, soit parce qu'un chemin est devenu inutilisable, soit parce qu'un chemin plus court entre deux réseaux vient de s'ouvrir. Ces modifications sont rares au regard du fonctionnement normal du routeur, mais elles existent.

Dans notre implantation, les tables de routage sont des variables partagées du système, puisque chaque tâche de classification les utilise. Cependant, bien qu'il s'agisse de variable partagées, elle sont stockées dans un segment de mémoire cachable afin que chacune des tâches de classification puisse bénéficier d'une copie locale dans son cache. Tant que ces données ne sont pas modifiées, tout va bien, leur accès uniquement en lecture ne nécessite pas de protection contre les accès concurents. Par contre lorsqu'une information nécessitant la mise à jour de ces tables parvient au routeur, plusieurs copies de ces tables existent, et il faut les mettre toutes à jour.

Lorsqu'un paquet de ce type arrive, il est lu par une tâche de classification. Cette tâche va effectuer la mise à jour dans les tables de routage. La stratégie d'écriture de nos caches étant *write through*, on a l'assurance que cette mise à jour est immédiatement effectuée en mémoire, pas seulement dans le cache du processeur concerné. Cette mise à jour est à ce moment invisible aux autres tâches qui continuent d'utiliser une version obsolète contenue dans leur cache.

Après avoir modifié la variable globale, la tâche de classification envoie un signal (grâce à l'appel system `kill`) à toutes les autres tâches de classification.

Le traitement de ce signal consiste en l'invalidation de toutes les lignes du cache de données.

Ce dispositif n'a pas été implémenté dans nos simulations, mais ne semble pas poser de problème théorique.

7.1.3 La tâche d'ordonnancement

Le but de la tâche d'ordonnancement est de déterminer dans quel ordre les paquets, précédemment triés par la tâche de classification doivent être transmis à l'interface de sortie.

Une fraction de la bande passante est garantie à chaque canal de priorité. Chacune de ces parts garanties est le résultat d'un contrat de qualité de service passé entre l'application de classification et l'émetteur ou le récepteur du flux de paquets concerné. Le but de la tâche d'ordonnancement est de faire respecter ces contrats, en s'assurant que tous les flux ayant passé un contrat de qualité de service disposent de la part de bande passante qui leur revient.

Les contrats de qualités de service et les valeurs de priorité qui en découlent sont des constantes du système. C'est à dire qu'il n'est pas possible de les modifier dynamiquement pendant l'exécution de l'application.

Chacun de ces contrats correspond à une fifo de priorité, soit une entrée pour les tâches d'ordonnancement.

Lorsqu'une tâche d'ordonnancement doit émettre un paquet, elle choisit de servir une de ses fifo d'entrée. Le choix de la fifo à servir se fait de façon à servir celle qui est le plus loin de remplir son contrat de priorité. Le choix de cette fifo est fait en comparant la quantité totale de données émise sur l'interface de sortie avec la quantité de données émise pour chaque fifo. Chaque fifo de priorité est affecté d'une part de la bande passante, exprimée en pourcentage du flux sortant. On considère la fifo de priorité f. On appelle D_f la quantité de données émise pour la fifo de priorité f, P_f la part, en pourcentage, de bande passante accordée à f, et D_t la quantité totale de données émise par l'interface de sortie. Si $D_f < P_f \times D_t$, alors, le contrat de qualité de service n'est pas rempli pour la fifo f. La tâche d'ordonnancement choisit donc d'émettre un paquet de la fifo de priorité f. Ceci fait, les valeurs D_f et D_t sont mises à jour avant le choix d'un nouveau paquet.

Chacune des instances de la tâche d'ordonnancement utilise ses propres variables pour les valeurs de D_f et D_t.

7.1.4 L'encapsulation des paquets : la structure slot

La copie des paquets qui arrivent de l'extérieur vers la mémoire du système est une tâche qui est remplie par un coprocesseur matériel spécialisé : le coprocesseur

Input Engine. Son but est de lire des paquets sur un lien Ethernet et de les copier dans la mémoire du système.

En raison du manque de mémoire embarquée sur la puce, il n'est pas possible de copier l'ensemble du flux sur la puce. La mémoire se remplirait trop vite, et on aurait trop souvent à jeter des paquets, faute de pouvoir les stocker. Il faut donc envoyer une part de ce flux vers la mémoire externe.

Pour déterminer quelle part du flux doit être conservée sur la puce, et quelle part peut être reléguée dans la mémoire externe, nous avons fait la constatation que la plupart des applications réseau, et en particulier celle de routage et de classification, ne travaillent que sur l'en-tête du paquet. Les données elles-mêmes sont en général transférées sans avoir été modifiées, ni même lues.

Dès lors, notre choix a été de copier l'en-tête de chaque paquet dans la mémoire embarquée, tandis que le corps du paquet était placée dans la mémoire externe. Ceci nous garantit que toutes les trames Ethernet arrivant ont leur en-tête facilement accessible pour l'application.

Ca choix a évidemment des conséquences sur les performances d'applications qui travaillent sur la totalité du paquet. Par exemple des applications de cryptage, ou de contrôle de contenu.

Ce choix rend également le fonctionnement du coprocesseur *Input engine* (ainsi que celui du coprocesseur output engine) un petit peu plus compliqué que ce que l'on avait initialement expliqué. Il doit donc, en plus de la copie des paquets, découper ceux-ci en blocs de taille fixe à copier dans la bonne zone de mémoire (interne ou externe), et en plus lier ces blocs entres eux de façon à pouvoir reconstituer le paquet en fin de traitement. (Cette reconstitution sera du ressort du coprocesseur *Output engine*.) La solution que nous proposons est d'encapsuler ces tranches de paquets dans une structure de données créée dans ce but. Le format de cette structure appelée structure slot est le suivant :

```
typedef struct slot{
  DESC desc;
  unsigned char data[120];
}slot;
```

Le premier champ de cette structure, le champ `desc` a une longueur de 8 octets. C'est également une structure qui contient des informations sur le paquet dans son ensemble, le slot courant et l'adresse du slot suivant, s'il y en a un. Le format de cette structure DESC est le suivant :

```
typedef struct descriptor{
        unsigned int address;
        unsigned int TT:        11;
        unsigned int TS:        7;
        unsigned int OFFSET:    7;
```

```
        unsigned int INTERNAL:   1;
        unsigned int DATE:       6;
}DESC;
```

Cette structure DESC fait exactement 8 octets de long, ce qui porte la taille totale de la structure slot à $120+8=128$ octets. La signification de chacun des champs de cette structure est résumée dans le tableau ci-dessous.

Nom	Signification
address	Adresse du slot suivant de ce paquet (0 s'il s'agit du dernier slot).
TT	**T**aille **T**otale du paquet, exprimée en octets.
TS	**T**aille de la partie utile du **S**lot, c'est à dire le nombre d'octets utilisés de ce slot. Exprimée en octets.
OFFSET	Décalage. Il s'agit du nombre d'octets non utilisés situés avant le premier octet du paquet.
INTERNAL	Valeur binaire valant 1 si ce slot est situé en mémoire interne, 0 sinon. Ce champ est utilisé pour la gestion des slots.
DATE	Numéro du paquet. C'est un compteur modulo 2^6. Il est utilisé dans le cas où les paquets doivent être réordonnés avant leur copie sur le lien de sortie.

Une remarque sur le champ `OFFSET` : Sa valeur standard est 40 (octets) pour le premier slot, 0 pour les suivants. Avoir un espace vide de 40 octets à l'avant du paquet permet à l'application d'ajouter des données devant le paquet sans avoir à déplacer celui-ci. La plupart du temps, l'ajout de données de la part d'une application réseau consiste en l'apposition d'un nouvel en-tête qui se situe devant le paquet.

Cette modification d'en-tête est le but de l'application MPLS (Multi Protocol Label Switching)[RVC01].

7.1.5 Le graphe de tâches complet

La figure 7.2 montre le graphe de tâches tel qu'il est implanté. Le coprocesseurs *Output Engine* utilise deux canaux MWMR pour communiquer au coprocesseur *Input engine* les adresses de slots ré-utilisables quand ceux-ci sont libérés. Il y a deux canaux, un pour les adresses internes (correspondant à de la mémoire embarquée sur puce) et un pour les adresses externes, correspondant à la mémoire externe à la puce. Les tâches de classifications sont également connectées à ces deux canaux : lorsqu'elles détectent une erreur dans un paquet, celui-ci doit être détruit, mais les slots qu'il occupait doivent pouvoir être réutilisés. Il faut donc

que la tâche recopie les adresses des slots concernés dans les canaux. Symétriquement, lorsqu'une erreur est détectée, dans la plupart des cas, un paquet est envoyé pour informer l'expéditeur. Il faut donc pour cela disposer d'un slot libre. Pour des raisons de clarté du schéma, seules les liaisons d'une des tâches de classification vers les fifos contenant les adresses de slots libres sont représentées.

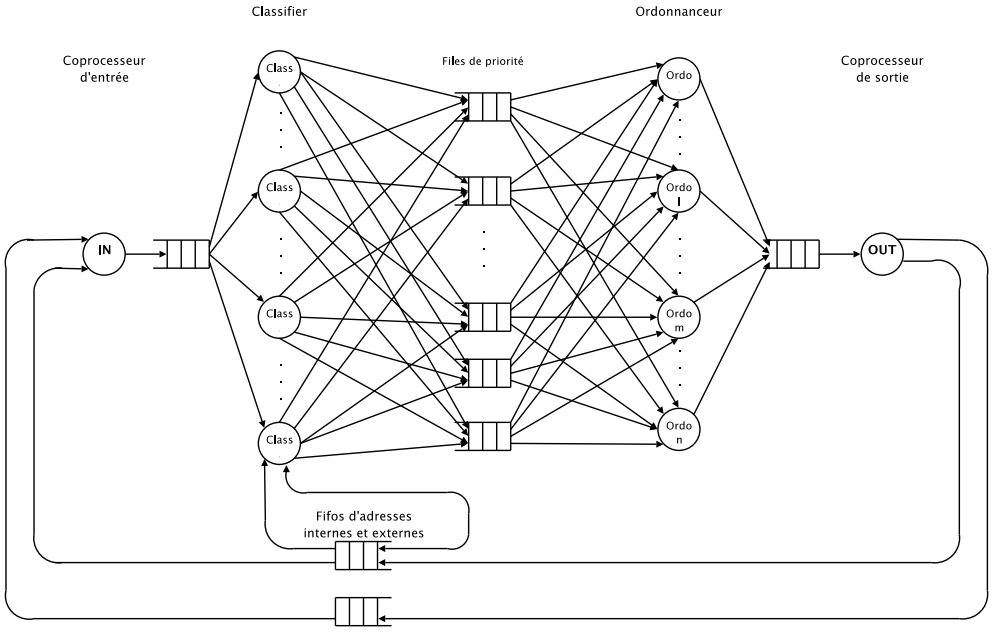

FIG. 7.2 – Le graphe de tâches de l'application, utilisant les deux types de parallélisme : task farm et pipeline. Les Canaux MWMR servant à la gestion des slots libres sont représentés en bas du schéma.

7.2 Plateforme matérielle

7.2.1 Introduction

La plateforme matérielle est la puce sur laquelle on va faire s'exécuter l'application embarquée. Cette plateforme est composée de sous-systèmes, appelés clusters, connectés entre eux par un micro réseau embarqué sur puce (NoC pour Network on Chip). On détaille son architecture de façon hiérarchique, d'abord du point de vue multi cluster, puis chaque cluster, puis les composants de chaque cluster.

Le nombre de clusters, le nombre de processeurs et de bancs mémoire sur chaque cluster, ainsi que la présence de coprocesseurs spécialisés sont des paramètres de la plateforme matérielle.

Cette plateforme matérielle est modélisée en systemC, en utilisant les modèles de simulation contenus dans la bibliothèque SOCLIB [SOC03]. Les différents composants de cette plateforme matérielle sont connectés entre eux en utilisant la norme VCI [VSI00].

7.2.2 La norme VCI

Au niveau local, on utilise la norme VCI avancée [VSI00] pour connecter les composants au réseau d'interconnexion.

La norme VCI définit le protocole de communication utilisé entre les composants et l'interconnect. Une transaction VCI est un couple requête/réponse. Un initiateur émet une requête (Une adresse + une commande Read/Write + une donnée en cas d'écriture), celle ci est acheminée à la cible correspondante, qui émet à son tour un paquet réponse, paquet qui est renvoyé à l'initiateur. Un paquet requête peut contenir un ou plusieurs mots, on parle dans le deuxième cas de transmission en rafale, ou *burst*. Une rafale est constituée d'une suite d'adresses consécutives. Le paquet réponse doit contenir autant de mots que le paquet requête.

La norme VCI sépare les requêtes de leur réponses : paquets requêtes et paquets reponses sont acheminés par des chemins différents, indépendamment l'un de l'autre.

L'interface VCI permet de séparer complètement le fonctionnement des IP (Intellectual Properties) de leur interface d'entrée/ sortie. Ce qui facilite la réutilisation de ces IP. Ceci permet de découpler le design d'un composant de son interface de communication.

7.2.3 Interconnect à plusieurs niveaux

L'un des principaux traits de l'architecture matérielle choisie est le parallélisme matériel élevé qu'elle offre. C'est ce parallélisme matériel qui répond aux importants besoins en puissance de calcul des applications réseau. En conséquence, le nombre de processeurs présents sur la puce peut atteindre plusieurs dizaines. A ce nombre déjà important, viennent s'ajouter les coprocesseurs (au moins pour les entrées sorties) et les bancs de mémoire. Ce nombre important de composants à connecter sur la puce influe sur la topologie retenue pour leur interconnexion.

Du fait que l'on compte mettre un nombre d'initiateurs potentiellement elevé, le choix d'un bus se révele inadapté. [ACG+03] montre qu'à partir de 4 processeurs, le PI-bus est saturé et ne permet plus de tirer parti de l'ajout de nouveau processeur. On s'oriente donc vers un micro réseau sur puce. Un tel interconnect présente l'avantage de voir sa bande passante croître avec le nombre d'agents que l'on y connecte. De ce fait, le problème de bande passante est résolu. Le revers

de la solution micro réseau est que la latence de l'interconnect augmente avec le nombre d'agents qu'on y connecte.

L'idée est alors d'utiliser deux niveaux d'interconnects. On fabrique de petits sytèmes construits autour d'un bus local ou du crossbar, et on les connecte entre eux au moyen d'un autre interconnect : le micro réseau DSPIN [MPGS06]. DSPIN est un micro réseau à commutation de paquets. DSPIN a été conçu pour connecter entre eux des sous-systèmes, ou clusters. Quel que soit le nombre de ces sous-systèmes connectés, la surface occupée par le micro réseau DSPIN demeure modeste. DSPIN peut fournir au concepteur du système des garanties à la fois sur la latence et sur le débit des communications. La topologie utilisée par DSPIN est du type Mesh. C'est à dire que chaque nœud du réseau est connecté à un sous-système local et aux 4 nœuds voisins (voir fig. 7.3). De cette façon on résoud le problème de bande passante posé par le bus, et celui de surface posé par le crossbar. Cette solution n'est cependant pas une panacée, et on verra au paragraphe 7.3, consacré au placement, qu'elle amène des problèmes quand au placement des objets en mémoire.

La plateforme matérielle se présente alors sous la forme d'un réseau d'interconnexion global qui connecte entre eux de petits sous-systèmes appelés clusters. Une illustration de ce type de système est donnée par la figure 7.3. Ces clusters contiennent chacun des processeurs, de la mémoire, et éventuellement des coprocesseurs spécialisés. Il s'agit d'une architecture de type NUMA (Non Uniform Memory Access) puisque les temps d'accès à la mémoire sont différents suivant qu'un processeur adresse la mémoire locale au cluster, la mémoire d'un autre cluster, ou la mémoire extérieure à la puce.

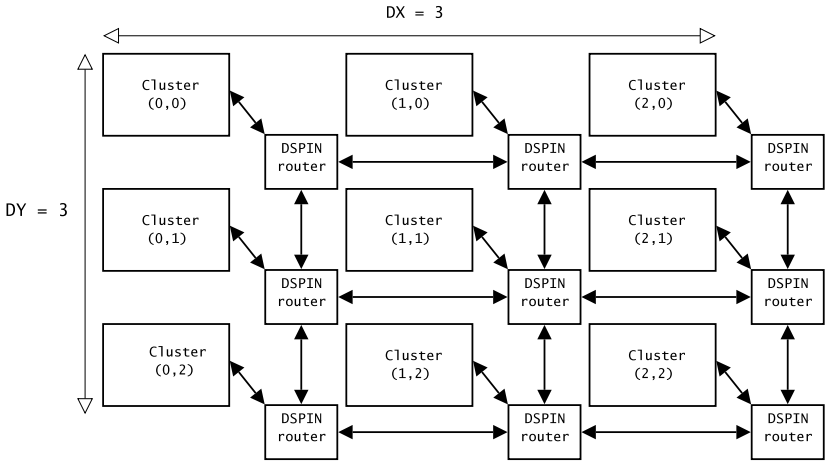

FIG. 7.3 – Un exemple de mesh à 9 clusters.

L'architecture que l'on va présenter ici est composée d'un assemblage régulier

de clusters. Ces clusters sont de deux types distincts : les clusters de calcul, qui contiennent des processeurs comme initateurs, et les clusters d'entrées/ sortie qui contiennent les coprocesseurs chargés de gérer les entrées/ sorties avec l'extérieur du système, ou l'accès à la mémoire externe.

7.2.4 Les clusters de calcul

On appelle cluster un sous-système articulé autour d'un interconnect local. Il s'agit en quelque sorte d'une "brique" élémentaire du système plus vaste que l'on souhaite décrire.

On appelle ces clusters clusters de calcul car ils comportent des processeurs destinés à exécuter des tâches logicielles (des tâches de calcul). L'interconnect choisi est un interconnect à faible latence, et à bande passante élevée : c'est un crossbar complet dans lequel tous les initiateurs sont connectés à toutes les cibles. Il y a devant chacune de ces cibles un multiplexeur destiné à selectionner la requête à présenter à la cible dans le cas où plusieurs se présentent simultanément. Cet interconnect présente l'avantage d'une bande passante élevée : tous les initiateurs sont connectés à toutes les cibles, et donc aucune contention ne peut avoir lieu dans l'interconnect. L'inconvénient corollaire est que sa surface croît très vite avec le nombre de composants à connecter. C'est pour cette seconde raison que la taille de ce type de réseau d'interconnexion est limitée à 4 ports initiateurs et 4 ports cibles.

Plus on a de processeurs sur le sous-système, et plus on peut y placer de tâches de traitement, mais il faut aussi intégrer les éléments cibles nécessaires. Ces éléments sont la mémoire, pour stocker le code des tâches et leurs données, ainsi que les canaux de communication. Une autre cible est le périphérique matériel qui gère les verrous utilisés pour protéger l'accès aux canaux de communication.

Lors de la mise au point de cette plateforme, on s'est aperçu que le fait de n'avoir qu'un seul banc de mémoire par cluster constituait un goulot d'étranglement, puisque les quatre processeurs du cluster essayaient parfois d'y accéder simultanément. Ce banc mémoire est également sollicité pour servir les requêtes formulées par d'autres processeurs situés sur d'autres clusters. Ces requêtes arrivent par le port de l'interconnect dédié aux communications avec les autres clusters. On a donc 5 initiateurs (les quatre processeurs du cluster et le port VCI qui reçoit les requêtes de tous les autres processeurs et coprocesseurs de la puce) qui cherchent à accéder à une unique cible, ce qui crée un phénomène de contention. On a donc ajouté un second banc de mémoire afin de répartir les accès effectués par les processeurs.

En outre, chaque cluster dispose d'un périphérique "timer", connecté à chaque processeur du cluster par un fil d'interruption.

Ces clusters se composent donc de 4 processeurs et de quatre cibles, 2 bancs

de mémoire, un gestionnnaire de verrous, et un timer, chargé d'émettre une interruption tous les 10 000 cycles. Il y a en plus un port initiateur et un port cible par lesquels le cluster communique avec les autres clusters du système. Le port initiateur sert à envoyer à l'extérieur les requêtes des 4 processeurs du sous-système, et le port cible sert à router dans le sous-système les requêtes des autres initiateurs (processeurs ou coprocesseurs) du système. L'architecture de ces clusters est représentée sur la figure 7.4.

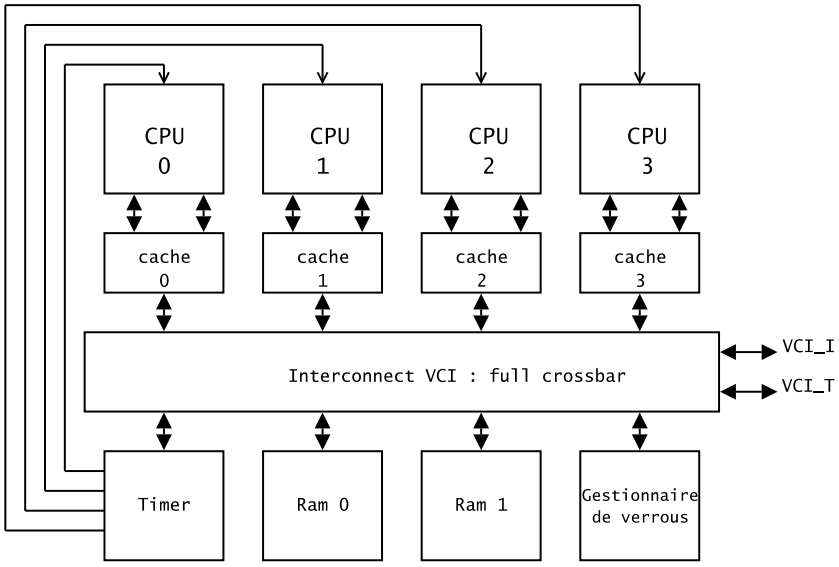

FIG. 7.4 – Architecture d'un cluster de calcul.

7.2.5 Les clusters d'entrée/sortie

Pour assurer les Entrées/sorties de la puce, et son interconnexion avec le monde extérieur, on a besoin de coproceseurs spécialisés. Ces coprocesseurs reçoivent des données sous la forme d'un flux de trames Ethernet. De la même façon, ils émettent en sortie un flux de trames Ethernet. On dispose de deux types de coprocesseurs : des coprocesseurs d'entrée, et des coprocesseurs de sortie. L'application que l'on déploie sur cette puce est une application de classification ordonnancement. Il s'agit d'un dispositif que l'on branche *en série* sur un lien Ethernet. Par conséquent, l'architecture matérielle que l'on utilise dispose d'un coprocesseur d'entrée et d'un coprocesseur de sortie.

On a donc un cluster d'entrée, qui comprend un coprocesseur *Input Engine*, connecté à l'interconnect par l'intermédiaire d'un contrôleur matériel MWMR. Le schéma de ce cluster est représenté sur la figure 7.5. De façon symétrique, on construit le cluster de sortie en remplaçant le coprocesseur *Input engine* par un

coprocesseur *Output Engine*. Ce cluster est représenté sur la figure 7.6. Sur ces deux clusters, on place un banc de mémoire principalement destiné à contenir les canaux MWMR avec lesquels les coprocesseurs communiquent.

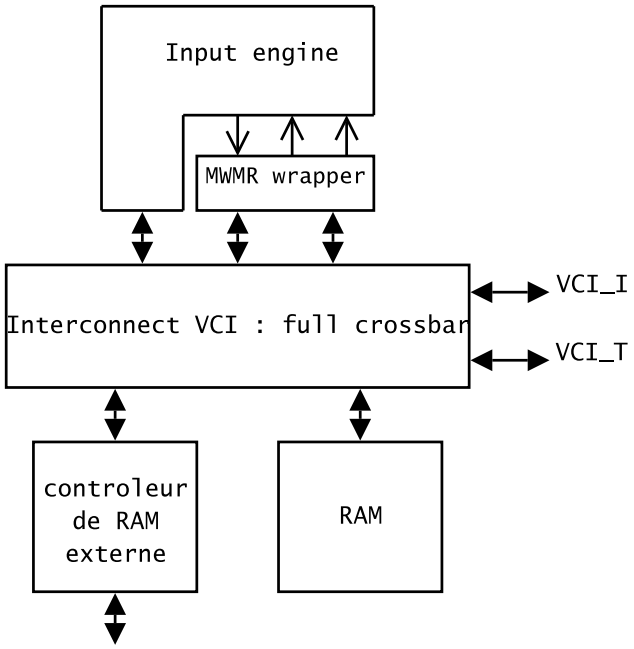

FIG. 7.5 – l'architecture d'un cluster d'entrée. Le coprocesseur *Input Engine* utilise un canal fifo en écriture et 2 canaux en lecture connectés au contrôleur matériel MWMR et un port VCI Maître connecté au réseau d'interconnexion local.

Les coprocesseurs copient la plus grande partie du flux entrant dans la mémoire externe à la puce. On place également un contrôleur de RAM externe sur le cluster D'entrée.

Les coprocesseurs *input engine* et *output engine* ont des interfaces similaires : un port VCI initiateur avec lequel ils effectuent la copie des paquets, et plusieurs port fifo par lesquels ils communiquent avec un contrôleur matériel MWMR, décrit au paragraphe 5.5. Les architectures de ces deux coprocesseurs sont détaillées aux paragraphes 7.2.10 et 7.2.11.

Ce sont ces clusters qui assurent l'ensemble des entrées/sorties de la puce. Par conséquent, il faut les placer sur les bords de celle-ci, de façon à pouvoir facilement les connecter aux plots de la puce.

7.2.6 L'espace adressable

La norme VCI que nous avons choisie stipule que l'espace adressable est partagé. Ce qui signifie que tous les initiateurs voient toutes les cibles. La désignation

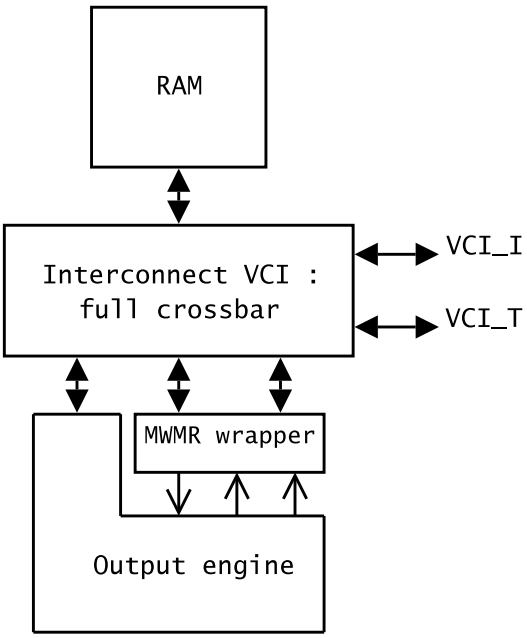

FIG. 7.6 – l'architecture d'un cluster de sortie. Le coprocesseur *Output Engine* utilise 2 canaux fifos en écriture et un canal en lecture connectés au contrôleur matériel MWMR et un port VCI Maître connecté au réseau d'interconnexion local.

d'une cible est faite par les bits de poids fort de l'adresse : chaque cible est désignée par une plage d'adresses. Et réciproquement, chaque adresse désigne une cible unique. Le routage d'une requête d'un initiateur vers une cible est le fait de l'interconnect. Les initiateurs n'ont aucune idée de la répartition physique des bancs de mémoire. La désignation d'une cible se fait en deux étapes : on désigne d'abord le cluster qui contient la cible, puis la cible elle-même au sein de ce cluster. Ce routage est effectué en décodant les bits de poids fort de l'adresse. Ces X bits de poids forts (X augmente avec le nombre de cibles) se décomposent en deux champs : les X_1 bits de poids fort, et les $X_2 = X - X_1$ bits de poids faible. Ils sont notés respectivement MSB et LSB pour Most Significant Bits et Less Significant Bits. Le champ MSB permet de désigner le cluster cible, tandis que le champ LSB indique quelle cible est concernée dans le cluster. La répartition de ces champs dans l'adresse est illustrée par la figure 7.7.

De la même manière le paquet réponse est routé vers l'initiateur en décodant un champ du paquet VCI : le champ SRCID (IDentificateur SouRCe). Le champ SRCID de la norme VCI se décompose également en deux : des bits MSB et des bits LSB. Les premiers servant à identifier le cluster d'où est issue la requête, le second désignant l'initiateur au sein du cluster. Bien que les champs MSB et LSB du champ SRCID et de l'adresse fonctionnent de façon complètement identique

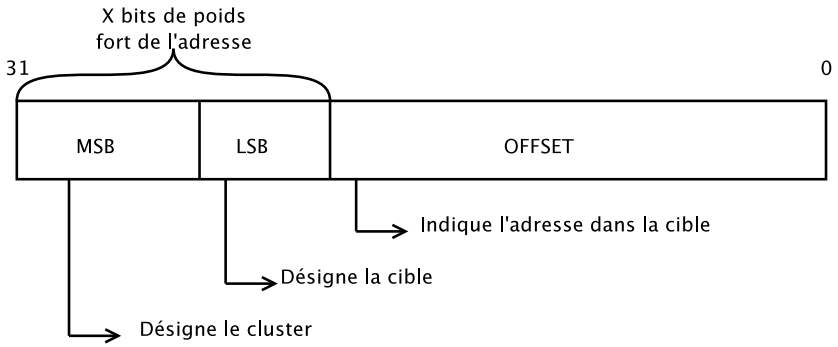

FIG. 7.7 – La répartition des différents champs dans une adresse VCI.

et remplissent le même rôle, les uns pour la désignation des initiateurs, les autres pour celle des cibles, ils convient de noter qu'ils n'ont rien en commun ; la valeur des MSB d'adresse d'un cluster n'a aucune influence sur celle des MSB de SRCID.

Une autre caractéristique de l'espace adressable est la caractérisation des segments de mémoire. En effet, cette mémoire est principalement destinée à être lue par les processeurs. Or, en fonctions du type de données que l'on va lire, on souhaite pouvoir bénéficier du mode de lecture en rafale que la norme VCI permet.

Cette notion de typage des segments ne concerne, du point de vue du matériel, que les caches des processeurs. En effet, ce sont eux qui, en fonction de l'adresse demandée par le processeur, vont construire la requête VCI correspondante. Et la forme de cette requête (un mot ou une rafale) est liée au caractère cachable ou non de l'adresse émise. L'espace adressable est divisé en segments, chaque segment étant un fragment continu de l'espace. Un segment est caractérisé par sa taille, son adresse de base, et son type d'accès. Le type d'accès indique la façon dont on va pouvoir y acceder. Il y a deux types d'accès aux segments mémoire : caché et non caché. Les segments cachés contiennent de la mémoire non partagée (pile d'une tâche) ou non modifiable (segment de code). Le cache les lit par rafale de N mots (N = 8 ou 16) et n'a pas besoin de consulter l'original en mémoire pour s'assurer de la cohérence de son contenu. Les segments non cachés contiennent les données partagées entre plusieurs tâches. C'est ici que l'on trouve les structures du noyau, les données explicitement partagées (les variables globales), ainsi que les canaux de communication.

Le type de segment pré-chargeable décrit au paragraphe 5.4.3 n'est pas utilisé dans cette architecture. Il s'agissait d'une extension des segments non cachables. Il s'agit de segments fondamentalement non cachés, mais que l'on va lire par ligne. Ils servaient au stockage des buffers des canaux MWMR.

7.2.7 La mémoire

La taille de la mémoire ainsi que sa répartition sur les bancs physiques ont une influence sur l'efficacité du système. La quantité disponible est fonction des caractéristiques de la puce. D'une manière générale, pour des raisons de surface disponible et donc de coût, la quantité de mémoire embarquée est faible.

7.2.8 Les processeurs et leurs caches

Les processseurs sont des processeurs généralistes de type RISC (Reduced Instruction Set Computer).

Il s'agit de processeurs MIPS R3000.

Ces processeurs ne sont pas connectés directement au reste du système. Au lieu de cela, ils passent par un cache pour effectuer leur accès aux périphériques (mémoire et coprocesseurs).

Les deux caches instruction et données sont regroupés au sein du même composant. Le cache utilisé est un cache à correspondance directe,chaque adresse ne pouvant occuper qu'une unique place dans la mémoire, et sa strategie d'écriture est *write through*, ce qui signifie que les écritures sont effectuées immédiatement en mémoire, *à travers* le cache. La taille de ces caches est paramétrable : on peut faire varier le nombre de ligne de chacun des deux caches instructions et données indépendemment, de même que l'on peut faire varier la taille de ces lignes.

7.2.9 Le périphérique gestionnaire de verrous *Ramlock*

Il s'agit d'un périphérique chargé de la gestion matérielle des verrous utilisés par le système d'exploitation et l'application embarquée. Ce périphérique fournit un banc de verrous binaires valant 0 (libre) ou 1 (pris). Lorsque le périphérique *Ramlock* reçoit une requête de lecture, il met l'adresse concernée à 1 (pris) et renvoit la valeur précédement contenue à cette adresse. Lorsqu'une requête d'écriture est reçue, la valeur contenue à l'adresse cible est remise à 0 (libre). Le périphérique ne vérifie pas que l'initiateur VCI qui rend le verrou est celui qui l'avait pris.

7.2.10 L'architecture du coprocesseur Input Engine

La tâche du coprocesseur *Input Engine* est de copier les trames Ethernet dans la mémoire du système.

Cette copie est le résultat de plusieurs étapes effectuées par le coprocesseur *Input Engine* qui transforment une trame Ethernet en une liste chainée de structures slots.

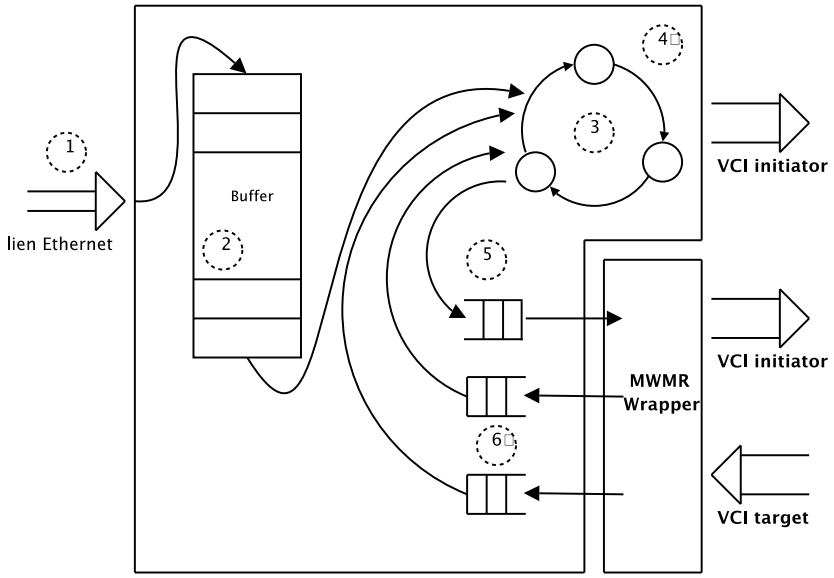

FIG. 7.8 – Schéma de l'architecture du coprocesseur *Input engine*

pour chaque paquet arrivant il faut déterminer le nombre de structures slots nécessaires, et les remplir avec le paquet. Ensuite, ces structures de données doivent être copiées dans les zones de mémoire appropriées. Une fois ceci accompli, il faut signaler aux tâches de traitement où se trouve le premier slot de la chaîne. Cette dernière opération est réalisée en copiant une structure DESC décrivant le premier slot dans une fifomwmr prevue à cet effet. Cette structure DESC contient l'adresse du premier slot, ainsi que des informations relatives à ce slot (TS, OFFSET) et au paquet (TT, DATE).

Comme on l'a vu, les structures `slots` sont des structures de données de 128 octets de long, et elles sont utilisées et gérées en premier lieu par les coprocesseurs Input engine.

La figure 7.8 montre l'architecture interne du coprocesseur *input engine*. La légende de ce schéma est expliquée ci-dessous.

1. Les trames Ethernet arrivent sur le lien 1. Elles sont stockées temporairement dans le tampon 2 en attendant d'être copiées dans la mémoire du système. Pour les besoins de la simulation, le débit de cette copie peut être ajusté pour chaque coprocesseur : on précise le nombre de cycles d'attente entre la copie de 2 mots de 32 bits. La valeur que l'on spécifie n'est donc pas un débit, mais une période, exprimée en nombre de cycles pour un mot de 32 bits. Pour passer de l'un à l'autre, on doit faire des suppositions sur la fréquence de fonctionnement du système : Soit D le débit, F la fréquence et T la période, on a l'égalité $D = F \times 32 \div T$. En supposant une fréquence de

fonctionnement de 700 MHz, on obtient, pour une valeur de 21, un débit de $700.10^6 \times 32 \div 21 \approx 1GB/s$.

2. Il s'agit du buffer d'entrée, où les paquets sont temporairement stockés avant d'être chargés dans des structures `slot`, puis copiés dans la mémoire du système. Lorsque ce buffer est plein ,les trames Ethernet qui arrivent par le lien sont simplement jetées. Lorsque cela se produit, cela signifie que le coprocesseur *Input Engine* n'est pas en mesure de copier ces trames dans la mémoire du système à la même vitesse qu'elles n'entrent. Cela peut avoir de multiples causes comme un débit d'entrée trop élevé, ou une pénurie de slots libres.
3. Cet automate copie les paquets du buffer d'entrée vers la mémoire du système. séquentiellement, il effectue les étapes suivantes :
   (a) Lire un paquet dans le buffer d'entrée.
   (b) lire sa taille.
   (c) Calculer le nombre de slots nécessaires :
   1 slot interne.
   $N \geq 0$ slots externes.
   (d) remplir le premier slot. Si on a calculé qu'il fallait au moins un deuxième slot pour stocker le paquet, on ne quitte cet état qu'après avoir obtenu une deuxième adresse de slot.
   (e) Envoyer les requêtes VCI correspondant à l'écriture du slot en mémoire.
   (f) Tant que (le paquet Ethernet n'est pas terminé)
   – obtenir une adresse de slot externe.
   – remplir ce slot. De même que pour l'état (d), si le slot courant n'est pas le dernier pour ce paquet, on attend de disposer de l'adresse du slot suivant avant de quitter cet état.
   – écrire ce slot en mémoire.
   (g) Envoyer le descripteur du premier slot à la fifo 5.
4. Ce dispositif reçoit les paquets VCI réponse. Comme le coprocesseur *Input Engine* n'émet que des commandes d'écritures, les réponses reçues ne sont que des acquittements. Au cas où un message d'erreur serait reçu, ce système émet un message d'erreur.
5. Une fifo de 32 mots de long sur 32 bits de large. Cela permet de stocker jusqu'à 16 descripteurs de slots en attendant leur traitement par le contrôleur MWMR.
6. Deux fifos de 16 mots chacunes, dans lesquelles on vient ranger des adresses de slots libres, prêts à l'emploi. Il y a une fifo pour les slots internes, et une

pour les slots externes.

Chacune des deux fifos 5 et 6 est connectée à un port du contrôleur MWMR et communique avec un canal MWMR.

7.2.11 Le coprocesseur Output Engine

Le coprocesseur Output Engine est en charge de l'émission des paquets sur le lien Ethernet après leur traitement. Les paquets circulent dans l'application encapsulés dans des structures slots, et c'est naturellement dans ces structures que le coprocesseur va trouver les paquets.

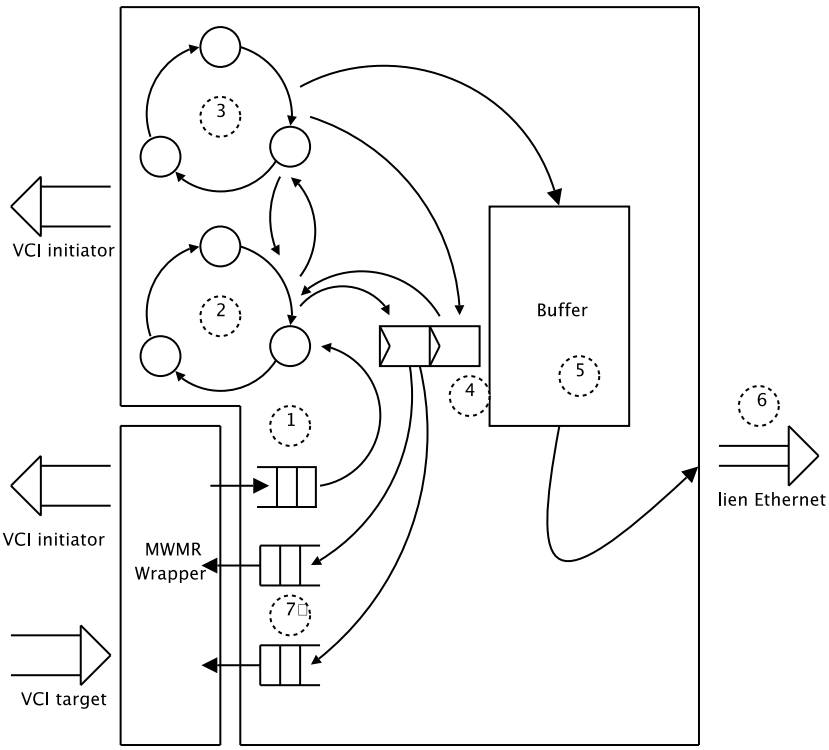

FIG. 7.9 – Schéma de l'architecture du coprocesseur *Output engine*

La figure 7.9 montre l'architecture interne de ce coprocesseur. La légende de ce schéma est donnée ci-dessous.

1. L'émission d'un paquet commence par la lecture d'un descripteur de slot. Les descripteurs de slots sont rangés dans la fifo 1.
2. L'automate 2 sort de l'état IDLE si la fifo 1 contient au moins un descripteur. Dans ce cas, l'automate lit ce descripteur et émet à partir de celui-ci les

commandes nécessaires pour lire ce slot. L'automate tire parti de la valeur de OFFSET du descripteur pour n'envoyer que le minimum de commandes de lectures VCI : les données de ce slot sont stockées à partir de l'adresse pointée par OFFSET.

3. l'automate 3 reçoit les réponses VCI. Ces réponses contiennent d'une part l'en-tête du slot, qui contient l'adresse du slot suivant s'il y en a un. Si c'est le cas, cette adresse est stockée dans le registre 4. Les données du paquet qui arrivent dans les réponses VCI suivantes sont copiées dans la mémoire tampon 5.
4. il s'agit de deux registres de 32 bits. L'un est utilisé par l'automate 2 pour y stocker l'adresse d'un slot à libérer. Cette adresse est accompagnée d'un bit qui signale si cette adresse doit être envoyée dans la fifo des slots internes ou dans celle des slots externes. C'est ici que sert le bit INTERNAL de chaque slot : au moment de renvoyer l'adresse du slot libéré vers une des fifos 7, on teste la valeur du bit INTERNAL. Le deuxième partie de ce registre est utilisée par l'automate de réception de réponse 3 pour y copier l'adresse du slot suivant d'un paquet. Ce registre est assorti d'un bit de validité qui signale à l'automate 2 que l'adresse qu'il contient est valide et doit être traitée.
5. Le tampon de mémoire 5 sert à stocker les données des paquets en attendant d'avoir un paquet complet pour pouvoir l'émettre sur le lien Ethernet de sortie 6.
6. Il s'agit du lien Ethernet vers lequel les paquets sont émis. Dans le cadre de nos modèles de simulation SOCLIB, ce lien est simulé par l'utilisation d'un fichier texte dans lequel on copie les paquets.
7. Ces deux fifo de 16 mots de 32 bits servent à recevoir des adresses de slots dont le contenu a été récupéré. Ces adresses doivent être renvoyées vers les canaux MWMR adéquats.

7.3 Impact du placement sur les performances

Dans cette partie, on étudie le déploiement de l'application de classification/ordonnancement sur une plateforme multi processeur. La plateforme choisie contient 2 clusters d'entrée / sortie, et 6 clusters de calcul, comportant chacun 4 processeurs. Il y a un dernier cluster, qui ne contient que des cibles VCI : il comporte un banc de mémoire qui contient le code de l'application, ses variables globales et ses constantes. Ce cluster contient également un périphérique contrôleur de sortie multi-TTY ; ce qui permet d'utiliser la fonction `printf` dans l'application et

suivre ainsi sa progression. Ce cluster contient également un périphérique contrôleur de verrous matériels *Ramlock*.

On utilise donc une plateforme matérielle avec $6 \times 4 = 24$ processeurs et 2 coprocesseur : 1 coprocesseur d'entrée *Input Engine* et un coprocesseur de sortie *Output engine*. L'architecture de ce système est représentée sur la figure 7.10.

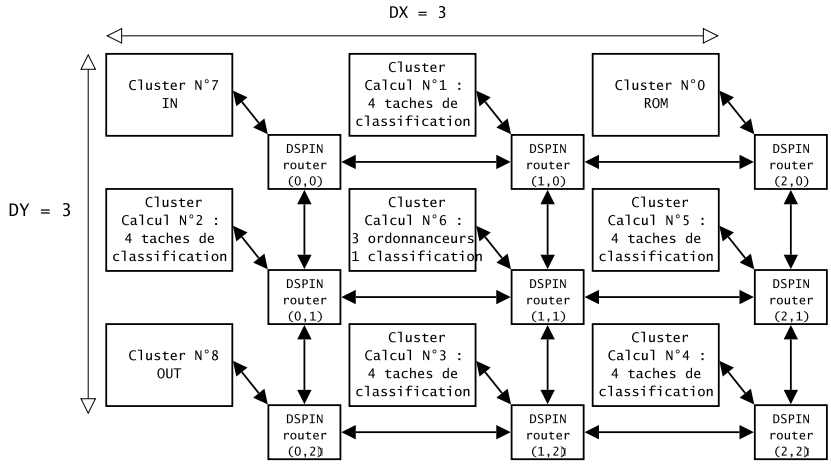

FIG. 7.10 – Répartition des clusters sur l'architecture mesh 3×3.

Le déploiement de l'application sur cette architecture multi-processeurs clusterisée consiste à choisir pour chaque tâche le processeur qui l'exécutera, ainsi que pour chaque objet logiciel (les canaux de communication principalement), le banc de mémoire qui le contiendra. Comme cette architecture multi cluster a des caractéristiques NUMA (Non Uniform Memory Access), il est important de regrouper sur un même cluster les tâches et les canaux de communication qui sont connectés dans le graphe de tâches. La figure 7.11 montre de tels exemples de groupes de tâches candidats pour le déploiement dans un cluster.

Le but de ce déploiement est que l'application soit capable de tenir le débit imposé par le coprocesseur d'entrée. Pour contrôler que c'est le cas, on mesure le débit sur le coprocesseur de sortie.

7.3.1 Exploration architecturale et premier déploiement

L'application que l'on déploie comporte 24 tâches logicielles et 2 tâches matérielles. Le graphe de tâches de cette application est représenté sur la figure 7.12.

On a vu que la tâche de classification est bien plus longue que celle d'ordonnancement. On va donc a priori utiliser plusieurs tâches de classification pour chaque tâche d'ordonnancement.

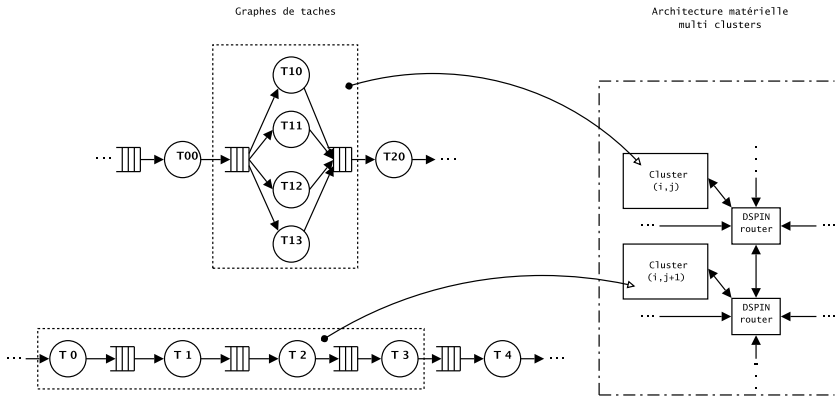

FIG. 7.11 – 2 exemples de déploiement de tâches sur un cluster de calcul. Sur chacun des deux graphes de tâches, on sélectionne un sous ensemble de 4 tâches qui sont placées sur le même cluster.

On évalue les performances de ce système en mesurant le débit obtenu sur l'interface de sortie. Le protocole expérimental consiste à effectuer une simulation de 3 000 000 cycles et de compter le nombre de paquets traités pendant ce laps de temps. Ce temps de simulation est mesuré à partir de la mise sous tension du système. C'est à dire qu'il contient le temps d'initialisation du système. Or pendant cette phase d'initialisation, le débit en sortie est nul. De plus, le démarrage des threads est synchronisé grâce à la fonction POSIX `pthread_cond_wait`, donc tous les threads commencent à travailler en même temps. En fait, seules les tâches de classification commencent à travailler, celles d'ordonnancement doivent attendre que les tâches de classification aient traité leurs premiers paquets avant de pouvoir démarrer.

Pour compenser cet effet d'amorçage du pipeline, on effectue la mesure de performance à partir de la date de sortie du premier paquet.

Pour cette expérience, tous les paquets ont une longueur IP de 40 octets, ce qui est la taille minimale pour ce type de paquets. A ces 40 octets , il faut ajouter 14 octets d'en-tête Ethernet. La taille totale des paquets entrants est de 54 octets.

Il s'agit du pire cas pour notre application de classification car le nombre d'en-tête à vérifier et le nombre de paquets à classer est maximal par rapport au volume total de données transféré.

Si on appelle N_0 le nombre de paquets récupérés sur l'interface de sortie, la mesure du débit de l'application est obtenue par le calcul suivant :

$D_{out} = 54 \times 8 \times N_0 \div (T_{fin} - T_{inits/cycle})$

Pour ces mesures, on fixe T_{fin} à 3 000 000 cycles. T_{ini} est la date, mesurée en cycles, à laquelle le premier paquet est récupéré sur l'interface de sortie du coprocesseur *Output engine*. $54 \times 8 = 432$ est le nombre de bits contenus dans un

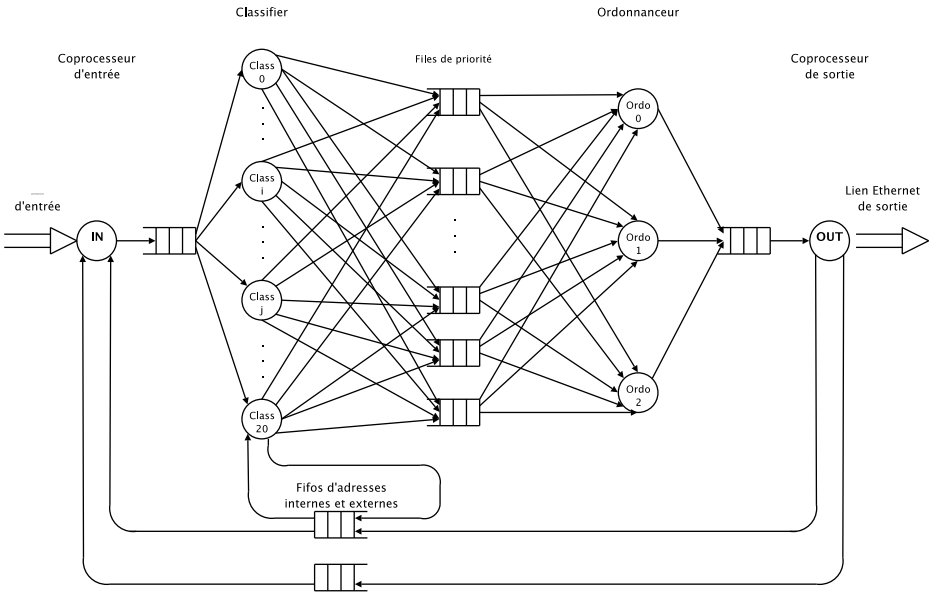

FIG. 7.12 – L'application de classification/ordonnancement telle qu'elle est initialement déployée sur une architecture à 24 processeurs.

paquet.

Dans cette exploration, on fait varier le nombre de tâches de classification par rapport au nombre de tâches d'ordonnancement. La somme des deux restant égale à 24. On fait également varier la taille des rafales lues ou écrites dans les canaux MWMR.

Le meilleur rapport que l'on obtient pour la répartition des tâches d'ordonnancement et de classification est de 21 tâches de classification pour 3 tâches d'ordonnancement.

Les tailles des rafales de communication sont les suivantes, de la gauche vers la droite de la figure 7.12 : la tâche IN lit les adresses de slots internes et externes par rafales de 16 mots. Les descripteurs sont envoyés dans le canal d'entrée par rafales de 32 mots (16 descripteurs). les tâches de classification lisent ces descripteurs par rafales de 8 mots (4 descripteurs) et les écrivent dans les fifos de priorité un par un. Les tâches d'ordonnancement lisent ces descripteurs un par un et les reécrivent un par un dans la fifo de sortie. La tâche OUT lit ces descripteurs par rafales de 32 mots (16 descripteurs), et renvoient les adresses des slots libérés par rafales de 16 mots. Lorsqu'une tâche de classification détruit un paquet, les slots de celui-ci sont renvoyés un par un dans les canaux MWMR correspondant. De même lorsque la tâche de classification créé un paquet (de notification d'erreur le plus souvent), elle lit les adresses de slot dont elle a besoin une par une.

Sur ce déploiement de l'application sur la plateforme matérielle, on a égale-

ment fixé la taille des canaux MWMR utilisés. Ceux-ci doivent être suffisament profonds pour accepter un flux de nombreux petits paquets en attente. Les tailles des canaux obtenues sont les suivants :
Une fifo d'entrée : 1024×2 mots de 32 bits, soit **8 Ko**.
12 fifos de priorité de 128×2 mots de 32 bits, soit **12 Ko**.
Une fifo de sortie : 1024×2 mots de 32 bits, soit **8 Ko**.
Une fifo d'adresses de slots internes de 1024 mots soit **4 Ko**.
Une fifo d'adresses de slots externes de 1024 mots soit **4 Ko**.

Pour obtenir une estimation de la mémoire nécessaire, on peut ajouter que l'application utilise $6 \times 64 = 384$ slots de 128 octets alloués dans la mémoire sur puce, soit **48 Ko** supplémentaires ainsi que 1024 slots de 128 octets alloués dans la mémoire externe, soit **128 Ko**. La quantité totale de mémoire disponible pour le stockage des paquets est de **84 Ko** de mémoire embarquée sur puce et **128 Ko** de mémoire externe. À cela on peut ajouter que la taille du code embarqué est de **44 Ko** et que chaque thread dispose d'une pile de **8 Ko** pour son exécution.

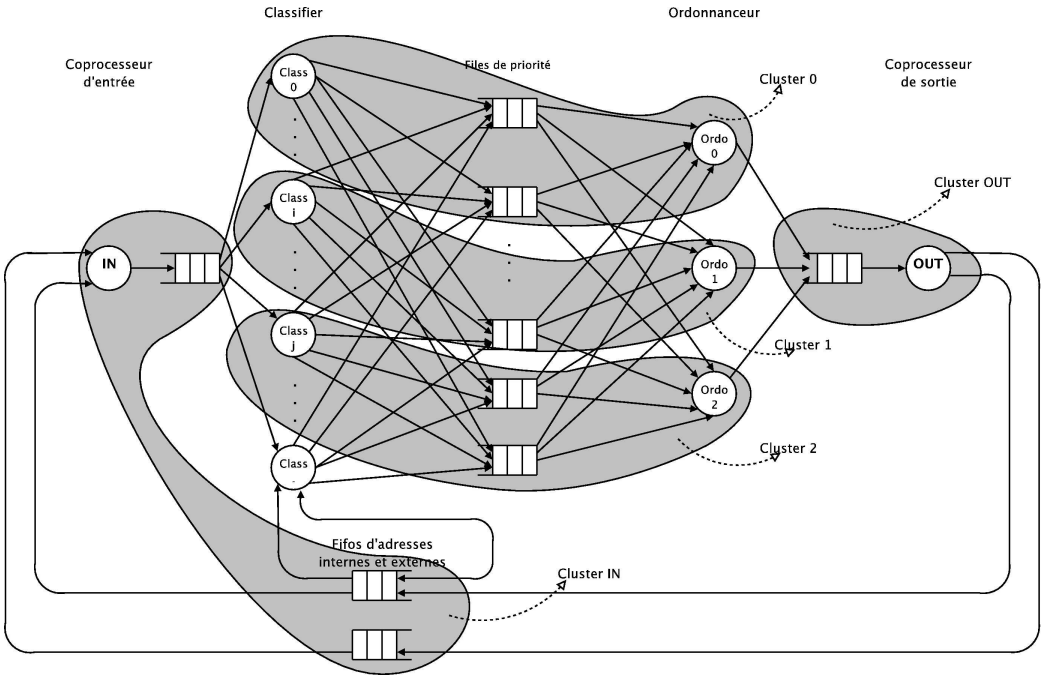

FIG. 7.13 – Le placement initial des tâches et des canaux MWMR

Le placement des tâches et des canaux MWMR sur la plateforme matérielle est illustré par la figure 7.13. Sur cette figure on voit que la tâche IN implantée par le coprocesseur *InputEngine* est placée sur le même cluster que les canaux MWMR contenant les adresses de slots libres, ainsi que le canal MWMR d'entrée.

Le coprocesseur *Output Engine* est situé sur le même cluster que le canal MWMR de sortie des paquets après leur traitement par les tâches d'ordonnancement. Les tâches d'ordonnancement et de classification sont réparties sur les clusters, de même que les fifos de priorités. Cette répartition est effectuée comme suit : les trois premier cluster 0, 1 et 2 comportent chacun trois tâches de classification, une tâche d'ordonnancement et deux fifos de priorité. Les clusters 3, 4 et 5 comportent chacun quatre tâches de classification et deux fifos de priorité.

Lorsqu'on exécute l'application dans cette configuration, le premier paquet sort au bout de 592 000 cycles. Au terme des 3 000 000 cycles, on obtient un total de 6624 paquets. Le débit de sortie obtenu est donc :

$D_{out} = 54 \times 8 \times 6624 \div (3000000 - 592000) = 1.18 bit/cycle$

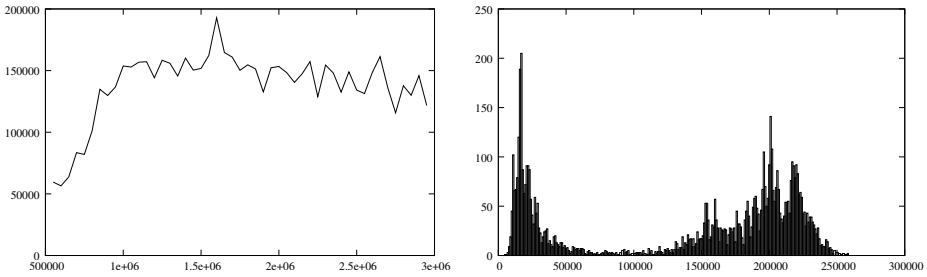

FIG. 7.14 – Évolution de la latence de traitement des paquets en fonction du temps (à gauche) et répartition des paquets par latence de traitement (à droite) pour le premier placement.

Lors de cette expérimentation, on avait fixé le débit d'entrée du coprocesseur *Input engine* à 1.6 bit/cycle (Un mot de 32 bits copié dans la mémoire du système tout les 20 cycles). La différence entre le débit d'entrée et celui de sortie indique que la mémoire du système se remplit plus vite que notre application n'est capable de la vider. Ce constat est confirmé par la courbe des latences de la figure 7.14, à gauche. Sur cette figure, on représente l'évolution de la latence de traitement des paquets en fonction du temps.

Cette courbe montre deux modes : le premier montre la latence qui augmente linéairement avec le temps, ce qui indique que l'application n'a pas encore atteint un régime stable. Ce mode dure pendant 400 000 cycles, de la date du premier paquet sorti, jusque vers le temps t=1 000 000 cycles. Au delà de ce moment, l'application a atteint un régime stable et la latence de traitement reste constante.

La forme de cette courbe montre que la latence augmente entre les premiers paquets traités et les suivants. Ceci s'explique par le fait que l'application ne parvient pas à tenir le débit imposé par le coprocesseur d'entrée. Ceci a pour conséquence que le coprocesseur *Input Engine* remplit la mémoire plus vite que l'application ne la vide. Les paquets étant traités en ordre fifo, ils passent de plus en plus de temps sur la puce en attendant qu'une tâche puisse les traiter.

Sur la figure 7.14, à droite, on a représenté la répartition des paquets par latence de traitement. Ce graphique permet de constater que derrière la relative uniformité de temps de traitement constatée sur la figure 7.14, à gauche, on voit apparaitre une forte disparité, illustrée par la présence de deux pics dans la répartition.

Les paquets qui composent le premier pic sont ceux qui subissent les tâches de classification et d'ordonnancement dans le même cluster, en ayant de plus été stockés dans une fifo de priorité située dans ce même cluster. Les paquets qui forment l'autre pic sont ceux qui ont subi ces trois étapes sur trois clusters différents. À cette différence s'ajoute le fait qu'au fil du temps, le coprocesseur *Input Engine* remplit le canal MWMR d'entrée plus vite que les tâches de classification ne parviennent à traiter les paquets. En conséquence ceux-ci restent pendant un temps de plus en plus long en attente dans cette première fifo.

7.3.2 Amélioration du placement

Ce placement initial a pour caractéristique que certains paquets bénéficient d'une bonne vitesse de traitement puisqu'ils subissent la classification, le stockage dans une fifo de priorité et l'ordonnancement dans le même cluster, tandis que d'autres paquets subissent ces trois étapes de classification, stockage et ordonnancement dans trois clusters différents, ce qui allonge nettement le temps nécessaire à leur traitement.

Un autre source de ralentissement concerne les communications entre les tâches d'ordonnancement et le canal MWMR de sortie. Ces communications ont toujours lieu entre deux clusters distincts, pour tous les paquets. Or ici les écritures sont effectuées paquet par paquet, ou plutôt descripteur de paquets par descripteur de paquet, soit par rafale de 2 mots. De l'autre coté de ce canal, la tâche OUT lit dans ce canal des rafales de 16 descripteurs. On souhaite donc favoriser les écritures, qui sont plus nombreuses, que les lectures qui on lieu par rafales.

Le second placement consiste à regrouper les trois tâches d'ordonnancement dans le même cluster avec le canal MWMR de sortie.

Ce placement est illustré par la figure 7.15. Sur cette figure on voit que les tâches d'ordonnancement sont placées sur le même cluster, avec le canal de sortie. Le quatrième processeur du cluster exécute une tâche de classification. Les tâches de classification sont déployées sur les clusters restant, de même que les fifos de priorité. On a donc 5 clusters (dont trois seulement sont représentés) qui exécutent quatre tâches de classification chacun et qui contiennent deux ou trois fifos de priorité.

On exécute à nouveau cette application avec ce nouveau déploiement, sur la même architecture matérielle. Tous les paramètres sont inchangés par rapport aux conditions de l'expérimentation du paragrahe 7.3.1.

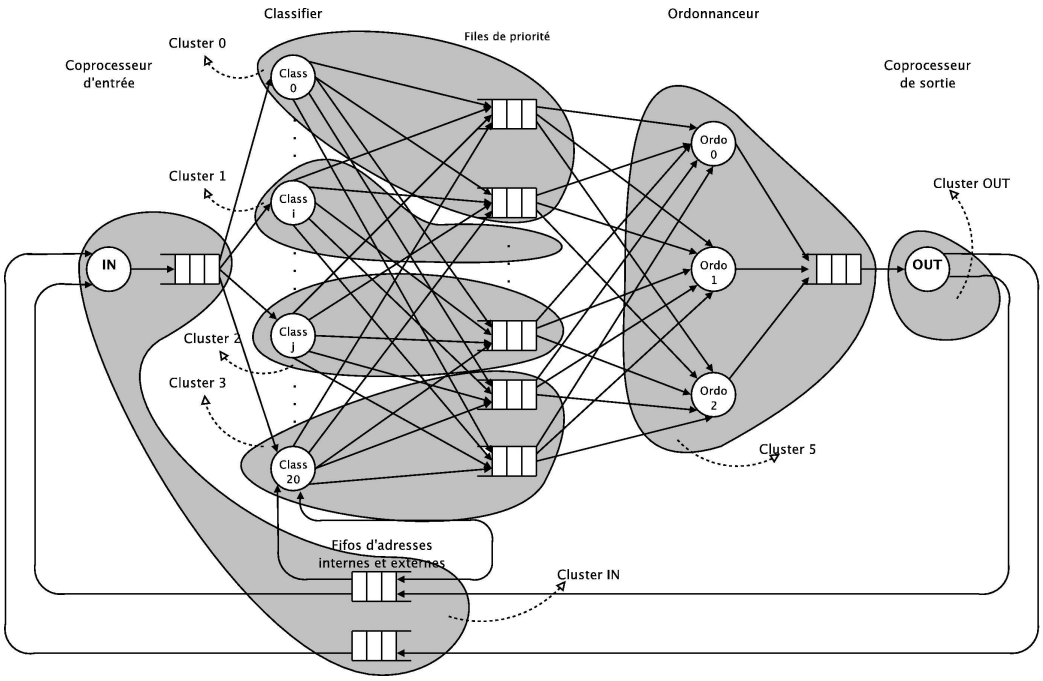

FIG. 7.15 – Le second placement des tâches et des canaux MWMR, où toutes les tâches d'ordonnancement sont regroupées dans le même cluster avec le canal MWMR de sortie.

Pour cette seconde expérience, le premier paquet est recueilli sur la sortie après 571 000 cycles et le nombre total de paquets reçus à la fin de la simulation est de 7408 paquets.

Le débit obtenu pour ce second placement est de :
$432 \times 7408 \div (3000000 - 571000) = 1.32 bit/cycle$
On constate une amélioration, mais celle ci est faible et ne suffit pas à atteindre la contrainte de débit imposée par le flux d'entrée, à savoir 1.6 bit/cycle. Ce résultat est confirmé par la courbe d'évolution de la latence de traitement au cours du temps (fig 7.16). On constate la même évolution sur cette courbe que pour le placement précédent, c'est à dire une forte augmentation de la latence pendant une période transitoire qui dure environ 400 000 cycles, suivie d'un régime stable caractérisé par une latence constante mais élevée. On constate néanmoins que ce second placement réduit la latence moyenne de traitement des paquets. Cela s'observe mieux sur la figure de droite, où l'on constate que la latence maximale de traitement est de 220 000, avec un second pic à 180 000 cycles. À comparer avec les valeurs obtenues pour le premier placement : une latence maximale de plus de 250 000 cycles et un second pic vers 200 000 cycles.

Ce second placement, bien que meilleur que le premier, est encore insuffisant pour respecter les contraintes fixées.

7.3.3 Placement final

Pour les deux précédents placements, on a réparti les fifos de priorité sur les clusters pour éviter de créer un phénomène de contention lié à leur accès. Cette façon de placer ces fifos fait que certains paquets sont stockés dans une fifo de priorité située sur le même cluster que la tâche de classification qui l'a traité tandis que d'autres sont ajoutés à une fifo située sur un cluster distant. L'inconvénient de cette répartition est que statistiquement, la majeure partie des paquets doit être stockée sur un autre cluster que celui où la classification a été faite. La raison en est simple : il y 12 fifo de priorité réparties sur 5 clusters, cela fait deux ou trois chances sur douze que la tâche de classification se trouve dans le même cluster que la fifo de priorité. Cette communication longue distance est systématique pour les tâches d'ordonnancement puisque celles-ci sont toutes groupées sur un cluster où il n'y a aucune fifo de priorité.

Le bilan de ces communications autour des fifos de priorité est que environ 20% des paquets font l'objet d'un transfert à l'intérieur d'un même cluster pour l'écriture de la tâche de classification vers la fifo, tandis que 80% passent par l'interconnect global. Et 100% des paquets passent par l'interconnect global pour être traités par une tâche d'ordonnancement

Pour améliorer ces communications autour des fifos de priorité, l'idée retenue est d'essayer de réduire la latence de tous les paquets. Pour cela, une solution est de placer toutes les fifos de priorité sur le même cluster que les tâches d'ordonnacement. De cette façon, le temps des communications entre les tâches de classification et ces fifos change peu, puisque seulement 20% de ces communications étaient locales à un cluster. L'effet bénéfique est que toutes les communications entre les tâches d'ordonnancement et ces fifo de priorité sont désormais locales à un cluster.

Ce nouveau placement est illustré par la figure 7.17. Sur cette figure, on voit le cluster d'ordonnancement qui contient toutes les fifos de priorité, ainsi que la

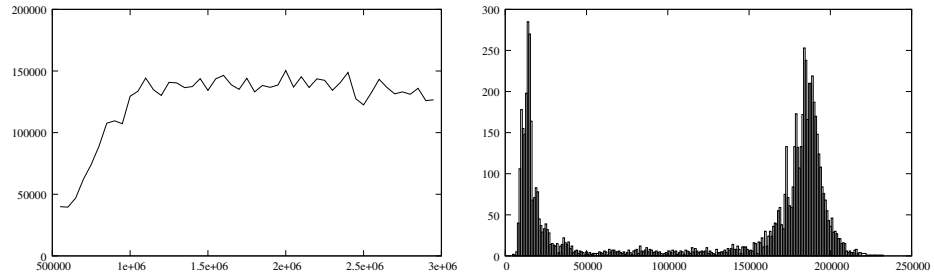

FIG. 7.16 – Évolution de la latence de traitement des paquets en fonction du temps (à gauche) et répartition des paquets par latence de traitement (à droite) pour le second placement.

fifo de sortie. Les 5 autres clusters de calcul contiennent chacun quatre tâches de classification.

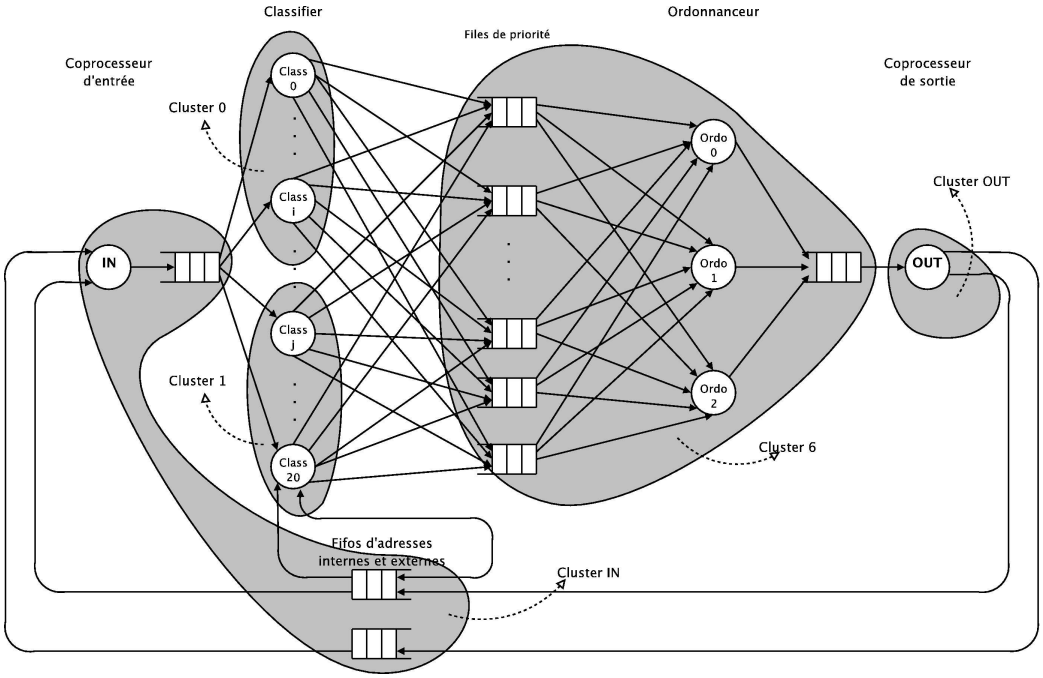

FIG. 7.17 – Le placement final des tâches et des canaux MWMR, où toutes les fifos de priorité sont regroupées dans le même cluster, avec les tâches d'ordonnancement.

On utilise le même protocole que lors des deux expériences précédentes, c'est à dire une simulation de 3 000 000 cycles au terme de laquelle on comptabilise le nombre de paquets traités.

Le résultat est que le premier paquet est reçu après 570 000 cycles et que le total de paquets reçus à la fin de la simualtion est de 9392.

Le débit obtenu en sortie est de :
$432 \times 9392 \div (3000000 - 570000) = 1.67 bit/cycle$

Ce résultat est nettement meilleur que ceux obtenus pour les deux précédents placements. Ce résultat permet de tenir la contrainte de débit qui était imposée. Le débit d'entrée est de 1.6 bit/cycle et le débit de sortie lui est légèrement supérieur (1.67 bit/cycle). Ceci s'explique par la méthode utilisée qui utilise comme temps de référence la date de sortie du premier paquet, ce qui introduit un léger biais puisqu'à cette date, plusieurs paquets sont déjà en fin de traitement, stockés dans la fifo de sortie.

Ce bon résultat est confirmé par la courbe montrant l'évolution de la latence en fonction du temps. Cette courbe (fig 7.18) montre que cette latence est stable

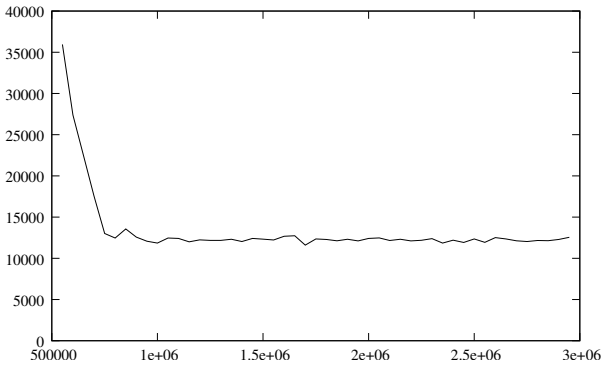

FIG. 7.18 – Évolution de la latence en fonction du temps pour le troisième placement. La latence est très stable en fonction du temps, ce qui indique que le temps passé en attente dans les fifos est constant.

dans le temps, indiquant que l'application a atteint son régime stable. Cette courbe indique une latence moyenne de l'ordre de 14 000 cycles. Les premiers paquets traités ont une latence de traitement plus longue que les suivants. La raison est que le coprocesseur commence à copier des paquets dans la mémoire du système alors que les tâches n'ont pas encore démarré. La latence de l'application est très stable dans le temps, ce qui indique que le temps passé en attente dans les fifos est à peu près constant. En fait, l'application est capable de traiter les paquets à un débit supérieur à celui imposé par le coprocesseur d'entrée. Par conséquent, les fifos internes sont presque vide, et les paquets n'y restent pas longtemps.

La figure 7.19 montre la répartition des latences de traitement. On y voit nettement que la plupart des paquets ont un temps de traitement compris entre 7 000 et 19 000 cycles. On constate surtout la disparition du second pic de latences. Cela confirme qu'il n'y a plus de paquets qui restent pendant des temps très long sur la puce.

On a pu, uniquement en jouant sur le placement des tâches et des canaux de communication, obtenir un gain de performances de plus de 30%, et ainsi respecter la contrainte de débit imposée.

7.4 Impact du placement sur la consommation d'énergie

Après avoir observé l'importance sur les performances du placement des tâches et des canaux MWMR sur les ressources offertes par l'architecture matérielle, on s'intéresse aux effets du placement sur l'activité du micro réseau. On s'intéresse à cet aspect car il s'agit d'un indicateur de la consommation énergétique de la puce.

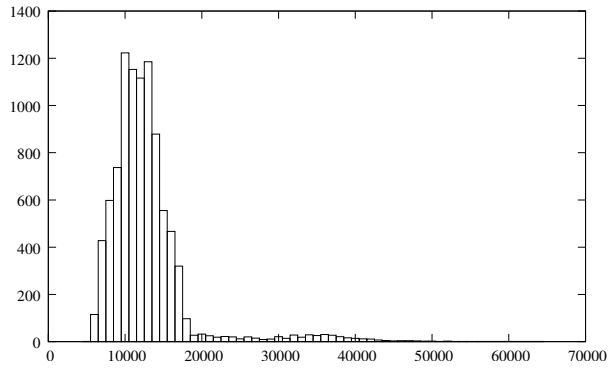

FIG. 7.19 – Répartition des paquets par latence de traitement. Presque tous les paquets ont une latence de traitement comprise entre 7 000 et 19 000 cycles.

7.4.1 Mesures d'activité pour le placement final

L'indicateur que l'on utilise pour la mesure de l'activité du micro réseau est le nombre de mots de requêtes qu'il achemine. Le nombre de commandes n'est pas un bon indicateur, car le nombre de mots qu'elle contient est variable. Il n'est pas nécessaire de mesurer le nombre de mots de réponses car celui-ci est égal au nombre de mots de requêtes.

On considère le placement des clusters sur le micro réseau comme illustré sur la figure 7.20. Pour ce placement, on a placé le cluster qui contient toutes les fifo ainsi que les tâches d'ordonnancement en position centrale. On a placé les clusters avec lesquels il fait beaucoup d'échanges, les clusters qui contiennent les tâches de classification au plus près. On place finalement les clusters d'entrée/sortie et de ROM sur les places restantes, sur les bords de la puce. On veille à ne pas trop éloigner le cluster d'entrée de celui de sortie, la part du flux stockée en mémoire externe devant être relue par le coprocesseur *Output Engine*.

Pour caractériser le trafic sur le micro réseau, on défini le protocole expérimental suivant : on laisse fonctionner l'application en mode normal pendant un certain temps, afin d'atteindre un régime stable. A ce moment on déclenche l'instrumentation du micro réseau : on comptabilise toutes les requêtes émises par un port initiateur à destination d'un port cible. On comptabilise le nombre de mots émis, pas uniquement le nombre de requêtes. Le micro réseau contenant 9 clusters, on obtient un tableau de 9×9 éléments. On effectue ce comptage pendant la période nécessaire au traitement de 2 000 paquets, puis on l'arrête. Ce protocole est illustré par la figure 7.21. Sur cette figure, on a indiqué les dates de début et de fin des mesures effectuées. La date de début de mesure est indiquée par le moment où le coprocesseur *Output engine* sort le 4 000 ième paquet. La date de fin est donnée par ce même coprocesseur à la sortie du 6 000 ième paquet. L'intervalle entre ces deux dates est de 521 000 cycles.

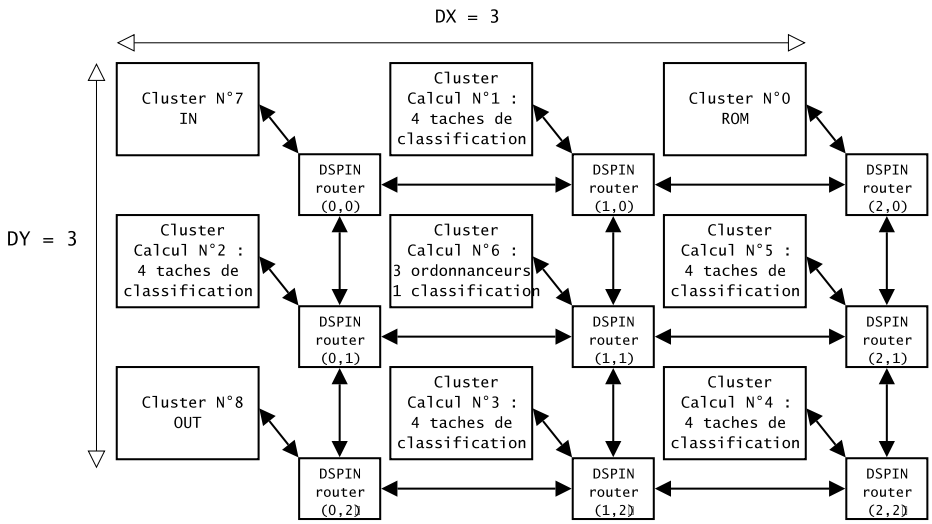

FIG. 7.20 – Répartition des clusters sur l'architecture mesh 3×3.

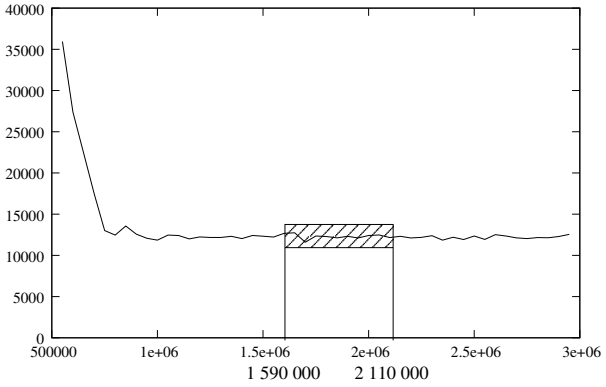

FIG. 7.21 – Période d'instrumentation, avec indication des dates de début et de fin.

Pour estimer la consommation énergétique sur le micro réseau, il faut prendre en compte Le nombre de transactions qui y ont lieu, mais également la distance parcourue par ces transactions. La distance est la distance de Manhattan entre les deux clusters qui participent à la transaction. La notion de distance est en fait le nombre de routeurs traversé moins un : une transaction entre le cluster 6 (au centre) et le cluster 1 (en haut) traverse deux routeurs (celui de son cluster, plus celui du cluster d'arrivée). c'est une transaction de longueur 1. Un transaction entre le cluster 2 et le cluster 3 traverse 3 routeurs, elle est donc de longueur 2.

La disance maximale pour une transaction est donc 4, pour les transactions s'effectuant entre deux clusters situés sur des coins opposés.

On considère que l'énergie nécessaire pour effectuer une transaction est pro-

portionnelle à la distance parcourue par cette transaction.

Si on appelle E_i l'énergie nécessaire au transfert d'un mot sur une distance i (compris entre un et quatre) entre deux clusters, on a $\forall i \leq 4, E_i = i \times E$, où l'on appelle E la quantité d'énergie nécessaire au transfert d'un mot entre deux clusters voisins immédiats. La mesure de l'activité du micro réseau revient alors, pour chaque cluster à faire le compte de ses transactions de longueur 1 puis 2, 3 et 4, puis à effectuer leur somme. On a alors, en appelant N_i le nombre de transaction total de longueur i l'égalité suivante :

$E_{total} = \sum_{i=1}^{4} i \times N_i \times E$

On obtient $E_{final} = 627000E$ pour un nombre total de transactions de 320000. Les valeurs ayant servi à effectuer ce calcul sont représentées sur les graphiques de la figure 7.22. Le cluster 0 ne contient aucun initiateur, par conséquent, il n'émet pas de trafic, il ne fait que répondre à des requêtes. On ne représente donc pas la répartition de son trafic. Dans les graphiques de la figure 7.22, on peut noter

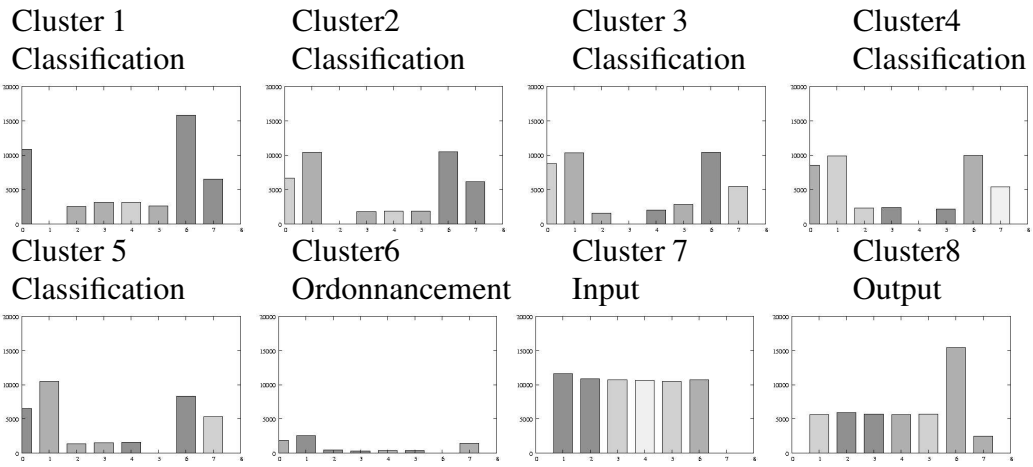

FIG. 7.22 – graphiques de l'activité du micro réseau cluster par cluster. Les barres sont teintées du plus sombre pour les transactions de longueur 1 au plus clair pour celles de longueur 4.

quelques éléments intéressants : d'une part, on constate que le cluster 8 du coprocesseur *Output Engine* effectue de nombreuses requêtes à destination du cluster 6, qui contient la fifo de sortie des paquets. D'un manière générale, on voit que tous les clusters de classification (numéros 1 à 5) font beaucoup de requêtes à destination du cluster 6. L'autre cluster qu'ils sollicitent beaucoup est le cluster 1, qui contient le verrou de la fifo d'entrée. Ce qui indique que les tâches de classification sont en pénurie de paquets à traiter.

7.4.2 Mesures d'activité pour le placement initial

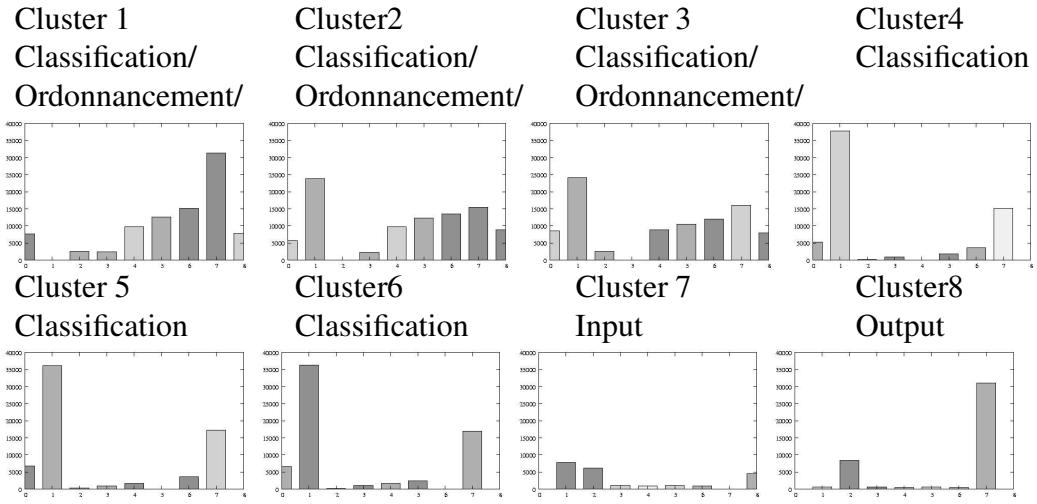

FIG. 7.23 – graphiques de l'activité du micro réseau cluster par cluster. Les barres sont teintées du plus sombre pour les transactions de longueur 1 au plus clair pour celles de longueur 4.

On effectue la même expérience avec l'application déployée en utilisant le placement initial. On effectue donc les mesures d'activité du micro réseau entre la sortie du 4 000 ième paquet et le 6 000 ième. On obtient les graphiques d'activité de la figure 7.23. Le laps de temps nécessaire au traitement de ces 2 000 paquets est de 746 000 cycles, soit environ 40% de plus que le placement étudié précédement.

En utilisant la même méthode que pour le premier placement, on obtient l'energie utilisée par ce second placement : $E_{initial} = 1014000E$, pour un total de 532 000 transferts recensés. Ceci permet de dire que le placement final réduit la consommation de 38% $(E_{final} - E_{initial}) \div E_{initial} \times 100 = 38\%)$ pour effectuer le même travail.

De même que pour les performances de l'application, le placement des tâches et des canaux MWMR sur les ressources de la plateforme matérielle ont une influence sur la consommation électrique de celle-ci. Dans l'exemple étudié, ce surcoût de consommation est lié à une augmentation du nombre de mots de requêtes qui passent par l'interconnect global, plus qu'à la longueur de ces requêtes qui reste sensiblement la même pour les deux expériences. Cette longueur des requêtes est estimée en divisant le nombre de transferts élémentaires utilisé pour évaluer la consommation énergétique par le nombre total de requêtes effectuées pendant la durée de l'expérimentation. On obtient $L_{moyen} = \frac{627000}{320000} = 1,96$ pour le placement final ; et $L_{moyen} = \frac{1014000}{532000} = 1,9$ pour le placement initial.

7.4.3 Remarque

Dans le placement des tâches, on ne prend pas en compte le facteur de proximité du code : dans notre plateforme, le code de l'application est présent une unique fois dans la mémoire du cluster 0. Tous les processeurs accèdent au même banc de mémoire pour charger leurs instructions, ce qui peut créer un goulot d'étranglement pour l'accès à cette mémoire. Cet effet est corrigé par l'utilisation de gros caches d'instruction de 16Ko par les processeurs, ce qui réduit considérablement le taux de MISS des caches pendant l'exécution.

Sur une plateforme matérielle réelle, il faudrait probablement utiliser des caches instruction plus petits et répliquer le code des tâches dans chaque cluster.

7.4.4 Conclusion

On a analysé dans ce chapitre l'importance du placement des tâches et des canaux de communications dans cette architecture NUMA. On s'est particulièrement intéressé au placement des canaux de communication par rapport aux tâches qui les utilisent. On a vu qu'en plaçant judicieusement ces canaux, on peut améliorer sensiblement les performances de l'application. Cette amélioration nous a permis d'atteindre les objectifs que l'on s'était fixé. Le gain obtenu demeure néanmoins modeste (environ 30%), puisqu'on ne peut réduire que le temps utilisé par les communications. Or le temps passé dans les communications appelle plusieurs remarques. Premièrement, il recouvre au moins deux choses : la première partie est le temps *réellement* nécessaire au communications, c'est à dire le temps nécessaire à la résolution du protocole en 5 étapes. Cette durée est succeptible d'être réduite par un placement optimisé des canaux MWMR. La seconde partie du temps de communication est en fait le temps nécessaire à la synchronisation des tâches : c'est le temps qui est consommé par une tâche pour lire dans une fifo vide, ou le temps que cette tâche passe à attendre que le verrou soit libéré. Ce temps là est passé en pure perte. Il importe de le réduire également, mais ceci est du ressort de l'exploration architecturale, c'est à dire pour notre application, fixer le nombre de tâches de classification et d'ordonnancement optimal, ce n'est pas du ressort du placement.

L'optimisation du placement des canaux de communication permet également de réduire la consommation énergétique. Nos expériences confirment qu'un bon placement du point de vue des performances est également un bon placement du point de vue de la consommation énergétique parce que dans un cas comme dans l'autre, on cherche à rapprocher des entités qui communiquent entre elles.

Chapitre 8

Conclusion

Dans cette thèse nous avons présenté une méthode de communication par mémoire partagée pour des applications logicielles multi-tâches destinées à être exécutées sur des architectures matérielles multi-processeurs intégrées sur puce (MPSOC).

La première question que nous nous sommes posés lors de l'énoncé de la problématique était la suivante :

Comment peut-on aider le concepteur à expliciter le parallélisme de l'application logicielle pour lui permettre d'utiliser le parallélisme matériel offert par les systèmes multi-processeurs sur puce ?

L'exploitation efficace des nombreux processeurs disponibles sur l'architecture matérielle impose la parallélisation de l'application en autant de tâches. Dans les applications orientées traitement de paquets, cette parallélisation ne peut pas être uniquement basée sur le modèle *pipeline*. En conséquence on utilise aussi le parallélisme *task farm*, dans lequel on crée plusieurs instances de la même tâche qui effectuent le même travail, mais sur des données différentes. L'utilisation de ce type de parallélisme nous a amenés à regrouper dans un même canal les communications entre un ensemble de tâches productrices et un ensemble de tâches consommatrices. Le but de cette factorisation est d'éviter la multiplication des canaux de communication.

Nous avons donc défini un modèle de communication par mémoire partagée où un nombre quelconque de tâches productrices émet des données à destination d'un nombre quelconque de tâches réceptrices.

La prise en compte de ces contraintes particulières nous a amené à définir un protocole robuste en 5 étapes, qui garanti la cohérence des échanges entre les tâches. Ce protocole a été implanté dans un intergiciel de communication pour MPSOC à mémoire partagée appelé MWMR (Multi-writer Multi-Reader).

L'utilisation de mémoire partagée entre plusieurs tâches pour leurs communications pose le problème de la cohérence des caches. Ce problème avait donné

lieu à la formulation de la seconde question de notre problématique :

Comment résoudre le problème de la cohérence des caches dans des architectures multi-processeurs à base de NoC, tout en utilisant des tampons de mémoire partagée ?

Plusieurs solutions ont été évaluées, la plus efficace repose sur l'utilisation des fonctionnalités du cache, la capacité à effectuer des rafales de lectures, et la capacité à invalider une ligne de cache pour forcer la lecture en mémoire.

Cet intergiciel de communication a été testé dans différents contextes, en utilisation point à point et en mode multi-points où il s'est avéré fiable et efficace.

Dans le contexte de communications point à point, les communications basées sur la bibliothèque MWMR ont remplacé avec succès la bibliothèque DPN issue de l'environnement Disydent. Cette expérience a montré que le protocole MWMR en 5 étapes peut être déployé pour émuler le fonctionnement des réseaux de Kahn et que l'implémentation qu'on en a fait s'avère être plus rapide que celle des canaux de communication DPN puisqu'elle amène, par exemple, un gain de performances allant jusqu'à 60% sur l'application de décodage d'image MJPEG.

On s'était également intéressé à définir un protocole de communication qui ne soit pas dépendant de la nature matérielle ou logicielle des tâches qui l'utilisent.

Peut-on définir un modèle de communication homogène qui permette des communications entre des tâches indifféremment matérielles ou logicielles ?

On a répondu à cette question en mettant au point un protocole de communication en 5 étapes qui est suffisamment simple pour pouvoir être implanté dans un contrôleur matériel. Ce contrôleur matériel est destiné à transformer les requêtes de lecture/écriture faites par un coprocesseur matériel en l'ensemble des accès mémoire nécessaires à la résolution du protocole MWMR. Ce contrôleur matériel implémente les fonctions `read` et `write` pour l'utilisation par un coprocesseur matériel.

Lors du développement de cet intergiciel de communication, on s'est attaché à en optimiser les performances, notamment en exploitant les capacités du cache. Sur les architectures de type NUMA, cette optimisation du code doit aller de pair avec la recherche d'un bon placement des tâches et des canaux de communications sur les ressources de la plateforme matérielle.

Quel est l'impact du placement des tâches sur les processeurs et des objets logiciels sur les bancs mémoire sur les performances du système considéré ?

Au cours de ces expérimentations visant à mesurer l'efficacité de l'implémentation de cet intergiciel de communication, on a mesuré l'importance du placement des canaux logiciels sur les bancs de mémoire physique. On a ainsi pu constater que pour une application de référence, on pouvait obtenir un gain de performances de 30% uniquement en jouant sur le placement des tâches et des canaux de communications. Ce gain de perfomances se retrouve également sur l'activité de l'interconnect et donc sur la consommation électrique.

8.1 Perspectives

8.1.1 Utilisation dans un système dynamiquement reconfigurable

Les canaux MWMR ont été définis dans la perspective d'une utilisation statique, c'est à dire avec un nombre de tâches productrices et consommatrices défini lors de l'initialisation du système, le schéma des communications ne variant pas durant l'exécution. Cependant, comme le canal de communication est défini indépendamment des tâches, on peut l'utiliser dans le cadre d'une reconfiguration dynamique de l'application. Une nouvelle tâche productrice ou consommatrice peut être ajoutée en cours d'exécution, sans que cela n'altère le fonctionnement du canal MWMR, ni des autres tâches. Pour cela, il faut juste que le mécanisme qui décide de la création de cette nouvelle tâche dispose de l'adresse en mémoire du canal MWMR à passer à la tâche. Cette information est suffisante, le reste des informations de contrôle du canal MWMR étant disponible dans sa structure de contrôle.

8.1.2 Automatisation du placement

Le placement des canaux de communication sur les bancs de mémoire est une des étapes de l'optimisation du déploiement d'une application sur une plateforme matérielle donnée. Dans les exemples que l'on a vu, on est parvenu a trouver un maximum local par essais successifs. Cependant, il serait souhaitable de mettre en évidence les ressorts sur lesquels jouer. Dans notre exemple la variable que l'on a cherché à minimiser est le nombre de communications passant par le micro réseau. Cette approche a donné de bons résultats sur cette application. Il faudrait en vérifier la validité en essayant de la généraliser à d'autres applications.

Il serait souhaitable de pouvoir extraire automatiquement du graphe de tâches un placement qui minimise le nombre de communications inter-clusters.

Annexe A

Manuel de l'utilisateur de la bibliothèque MWMR

A.1 Introduction

Le présent document a pour but d'exposer le fonctionnement et l'utilisation de la bibliothèque logicielle fifomwmr.

A.2 Principes généraux

La bibliothèque fifomwmr fournit des primitives de communication pour permettre à plusieurs tâches de communiquer entre elles. le terme fifomwmr est un acronyme qui signifie First In, First Out, Multiple Writers, Multiple Readers. Cet acronyme est également une définition de ce type de canaux. Il s'agit de canaux de type fifo, ce qui signifie que les données sont lues sur la sortie dans le même ordre qu'elles ont été écrites sur l'entrée, et d'autre part, il peut y avoir un nombre quelconque de tâches productrices ou consommatrices de données connectées à chaque canal.

A.3 Fonctionnement

Les canaux fifomwmr sont des objets logiciels qui comprennent un espace de mémoire qui peut être partagé entre plusieurs tâches. Ces différentes tâches n'ont aucune connaissance a priori les unes des autres, et c'est à la fifomwmr elle même de proceder à l'arbitrage entre des accès concurents.
C'est pour cette raison qu'à chaque fifomwmr est associé un verrou. Ce verrou a pour but de s'assurer qu'à un instant donné, il y a au plus une tâche en train

d'acceder à une fifomwmr donnée.
L'accès à une fifomwmr peut se décomposer en 5 étapes :

- Prendre le verrou
- Tester l'état de la fifo (non vide ou non pleine)
- Effectuer le transfert (lire ou écrire)
- Mettre à jour l'état de la fifo
- Rendre le verrou

On voit par là que ce canal MWMR contient deux parties : une première partie qui contient les données à communiquer, et une seconde qui contient les informations de contrôle nécessaires au fonctionnement de la fifo. Un canal MWMR est une strucuture de donnée, qui contient les champs suivants :

```
typedef struct fifomwmr{
  void * pointeur;
  LOCK_T lock;
  unsigned int largeur;
  unsigned int profondeur;
  unsigned int status; // nombre de mots occupées
  unsigned int readp;  // nombre de mots déjà lus
                       // (modulo la profondeur)
  unsigned int writep; // nombre de mots déjà écrits
                       // (modulo la profondeur)
  unsigned int *data;
}fifomwmr;
```

Le premier champ, `pointeur` n'est pas utilisé, il sert à aligner la structure sur une zone de 8 mots, pour faciliter son alignement en mémoire. Le second champ est un pointeur sur une structure de verrou. L'implantation de cette structure est dépendante de l'architecture matérielle cible. Dans le cas d'un système monoprocesseur de mise au point, typiquement une station Linux, il s'agit d'un verrou à exclusion mutuelle *Mutex* ; tandis que dans le cas d'une architecture multiprocesseurs sur puce, il s'agit d'un verrou à attente active *spin-lock*.

La première étape de l'utilisation d'un canal fifomwmr est donc l'allocation et l'initialisation de l'espace mémoire nécessaire.

A.3.1 Initialisation

L'initialisation du canal fifomwmr se fait en appelant la fonction `fifomwmrinit`. Cette fonction réclame trois paramètres : Le premier est le numéro du segment mémoire où l'on veut placer cette fifo. Ceci n'a de sens que si l'on est en mesure de choisir effectivement ce banc mémoire. C'est donc une valeur qui dépend de l'architecture matérielle utilisée. Le paramètre suivant indique la taille des objets que l'on souhaite faire passer dans cette fifo. Cette taille est

exprimée en mots (de 32 bits) et elle est fixe pour chaque fifo. On s'y réfère sous le nom de taille d'item. Le troisième paramètre est la profondeur de la fifo, c'est à dire le nombre maximal d'items qu'elle peut contenir. Cette valeur est exprimée en nombre d'items.

```
struct fifomwmr *
fifomwmrinit(unsigned int pool,      // memory segment id
             unsigned int lockpool, // lock segment id
             unsigned int l,         // item size (word)
             unsigned int p);        // channel depht (items)
```

Les deux premiers paramètres permettent d'indiquer explicitement le banc de mémoire et le gestionnanire de verrous que l'on souhaite utiliser. Ceci n'est possible que si le système d'exploitation sur lequel l'application est exécutée le permet. Dans le cas contraire, ces deux paramètres sont ignorés.

Les deux paramètres suivants permettent de spécifier la taille de l'espace de stockage du canal. Par exemple, si l'on souhaite créer un canal, par lequel on veut transmettre des données de 4 octets et dont on veut limiter la taille à un kilo octet, on écrira la séquence de code suivante :

```
struct fifomwmr *f;
if (!(f = fifomwmrinit(0, 0, 1, 256)))
  printf("Erreur: pas assez de memoire disponible\n");
```

A.3.2 Utilisation

Une fois le canal fifomwmr initialisé, la première chose que l'on souhaite faire, c'est y écrire des données.
On dispose pour cela de la fonction `mwmr_write` .Cette fonction a le prototype suivant :

```
int mwmr_write (struct fifomwmr *f,      // address of
                                         // the channel
                void * data,             // address of
                                         // the data
                unsigned int nb);        // number of 32 bits
                                         // words to
                                         // be written
```

Ces trois paramètres sont dans l'ordre, un pointeur sur le canal où l'on souhaite faire l'écriture, un pointeur sur la zone de données à recopier, et la quantité de données à copier. Cette dernière valeur s'exprime en mots de 32 bits. Cette fonction est non bloquante, ce qui signifie qu'elle retourne, que l'écriture ait réussi ou pas. En fait ce qui se passe est que la fonction acquiert le verrou, teste l'état de la fifo

et en fonction de cet état, elle calcule la quantité de données qui va effectivement être écrite.

Par exemple, supposons que l'on veuille écrire 2 blocs de 200 mots dans la fifo definie précedement (d'une capacité de 256 mots). On écrira

```
burst1 = mwmr_write (f , &data_zone1 , 200);
burst2 = mwmr_write (f , &data_zone2 , 200);
```

Si aucune tâche ne vient lire de données entre ces deux écritures, la fifo va être pleine. De ce fait, la deuxième écriture ne peut pas être effectuée complètement. Dans ce cas la statégie adoptée consiste à dire que la fonction `mwmr_write` transfère autant de données que possible et sa valeur de retour vaut le nombre de mots effectivement copiés. On aura donc pour l'exemple précédent :

```
burst1=200
burst2=56
```

La lecture des données s'effectue de façon analogue à l'aide de la fonction `mwmr_read`. Cette fonction utilise également trois paramètres :

```
int mwmr_read (struct fifomwmr *f,        // address of
                                          // the channel
               void * d,                  // address of
                                          // the free space
               unsigned int nb);          // number of 32 bits
                                          // words to be read
```

Ces trois paramètres sont respectivement un pointeur sur la fifomwmr où lire, un pointeur sur une zone mémoire où copier les données, ainsi que la taille du transfert en mots.
ATTENTION À CE QUE LA ZONE POINTEE PAR *D SOIT ASSEZ GRANDE POUR RECEVOIR LES DONNEES !
De la même manière, le code de retour de la fonction est le nombre de mots qui ont effectivement été lus.

A.4 Fonctions supplémentaires

Les trois fonctions `fifomwmrinit, mwmr_read` et `mwmr_write` constituent la base de la bibliothèque et sont réellement le minimum nécessaire pour s'en servir si l'on utilise uniquement des tâches implémentées sous la forme de threads. Cependant, d'autres fonctions sont disponibles.

A.4.1 Accès bloquants

Le premier couple de fonctions supplémentaires permet d'accéder aux canaux MWMR de façon bloquante. Un accès bloquant se caractérise par ce que la fonction ne retourne que lorsque la totalité du transfert a été effectué. Le prototype de ces fonctions est le suivant :

```
void b_mwmr_read(struct fifomwmr *f,      // address of
                                          // the channel
                 void *d,                 // address of
                                          // the free space
                 unsigned int nb);        // number of 32 bits
                                          // words to be read

void b_mwmr_write(struct fifomwmr *f,     // address of
                                          // the channel
                  void *data,             // address of
                                          // the data
                  unsigned int nb);       // number of 32 bits
                                          // words to
                                          // be written
```

Les prototypes de ces fonctions sont semblables à ceux des fonctions d'accès décrites précedemment. La seule différence concerne le type de retour puisque ces fonctions ne retournent à l'appelant qu'une fois le transfert effectué dans son intégralité. En ce qui concerne l'implémentation, il est à noter que ces fonctions sont des surcouches des fonctions `mwmr_read` et `mwmr_write`. On peut également signaler que l'ensemble des données n'est pas nécessairement transféré d'un seul coup. Si le canal MWMR ne permet pas au transfert de s'effectuer complètement, la fonction demande le déchargement de la tâche du processeur par l'appel à sched_yeld(). La tâche est déchargée du processeur, mais elle reste éligible. En particulier, elle peut être réélue sur le processeur alors que le statut du canal MWMR ne permet toujours pas le fin de la transaction. Il est également possible qu'une fois que le statut du canal a été modifié par une lecture ou une écriture de données faite par une autre tâche, une tâche tierce vienne effectuer un transfert avant la fin de celui qui a été suspendu. La spécification des primitives `b_mwmr_read` et `b_mwmr_write` ne permet pas de garantir qu'un transfert de données aura lieu de manière contigüe. Le système des items garantit en revanche que chaque item de données sera bien copié dans le canal de façon atomique.

A.4.2 Autres fonctions

Une autre fonction est disponible, il s'agit de la fonction `mwmr_flush`

```
int fifomwmrflush(struct fifomwmr *f);
```

Cette fonction a pour objet de vider un canal MWMR de tous les éléments qu'il contenait. Ceci est utile dans le cadre d'une réinitialisation du logiciel, particulièrement dans le contexte d'une utilisation pour une applicatin avec des contraintes de temps réel.

Cette fonction remet à zéro les pointeurs de lecture et d'écriture dans le tampon de la fifo, ainsi que son statut.

La dernière fonction de la bibliothèque est la fonction mwmrHinit.

```
volatile int mwmrHinit(struct fifomwmr *f,
                       unsigned long adr);
```

Cette fonction sert à configurer un des canaux du contrôleur matériel **soclib_vci_mwmr** pour lui permettre d'utiliser un canal MWMR. Le premier argument est un pointeur sur un canal MWMR, le second une adresse. Cette adresse sert à identifier le contrôleur MWMR que l'on souhaite configurer, et également à désigner le canal de ce contrôleur que l'on souhaite configurer.

Les bits de poids fort servent à indiquer la cible visée tandis que ceux les bits 7, 6 et 5 déterminent le canal MWMR à configurer.

La configuration d'un canal MWMR dans un contrôleur MWMR consiste à écrire certaines valeurs dans des registres internes du contrôleur MWMR. Ces valeurs sont les suivantes :
MWMR_STATE_AD, qui reçoit l'adresse du statut du canal MWMR.
MWMR_OFFSET_AD, qui est l'adresse du pointeur `readp` ou `writep`, en fonction du type d'accès effectués par ce controleur à ce canal.
MWMR_LOCK_AD est l'adresse du verrou.
MWMR_DEPTH, est la taille en mots de 32 bits de la zone de données.
MWMR_BASE_AD, qui reçoit l'adresse du début de la zone de données du canal MWMR.
MWMR_RUNNING est un indicateur booléen sui est mis à un lorsque la configuration est terminée : le canal est prêt à servir.

La fonction `mwmrHinit` englobe ces 6 accès aux registres du contrôleur MWMR, mais il est possible d'accéder à chacun indépendamment. De même, le controleur MWMR fournit jusqu'à 8 adresses pour configurer le coprocesseur associé au contrôleur MWMR (registres de CONFIG, accesibles uniquement en écriture), ainsi que 8 adresses pour que ce coprocesseur fournissent des informations sur son état (registres de STATUT, accessible en lecture seulement). Le décodage des adresses employées alors est effectué comme suit.

9	8	7	6	5	4	3	2	1	0	Type de registre
0	0	0	0	0	0	0	0	*	*	Soft Reset
0	0	Z	Z	Z	Z	Z	Z	*	*	Reservé quand ZZZZZZ $\neq$ OOOOOO
										Registres de configuration des canaux
0	1	X	X	X	0	0	0	*	*	MWMR_STATE_AD
0	1	X	X	X	0	0	1	*	*	MWMR_OFFSET_AD
0	1	X	X	X	0	1	0	*	*	MWMR_LOCK_AD
0	1	X	X	X	0	1	1	*	*	MWMR_DEPTH
0	1	X	X	X	1	0	0	*	*	MWMR_BASE_AD
0	1	X	X	X	1	0	1	*	*	MWMR_RUNNING
										X X X définit l'index de FIFO :
										- 0 à 3 pour un canal de lecture
										- 4 à 7 pour un canal d'écriture
1	0	X	X	X	*	*	*	*	*	Registres de configuration
										du coprocesseur
										X X X définit l'index du registre
1	1	X	X	X	*	*	*	*	*	registres de statut du coprocesseur
										X X X définit l'index du registre

Chaque canal d'un contrôleur MWMR doit être configuré avant de pouvoir être utilisé par le coprocesseur associé. Lorsque cette fonction est utilisée dans le cadre d'une exécution sur station de travail Linux, elle n'a aucun effet.

ANNEXE A. MANUEL DE L'UTILISATEUR DE LA BIBLIOTHÈQUE MWMR

Annexe B

Acronymes

API *Application Program Interface*
ARP *Address Resolution Protocol*
CABA *Cycle Accurate, Bit Accurate*
CAO *Conception Assistée par Ordinateur*
CASS *Cycle Accurate System Simulator*
CRC *Contrôle de Redondance Cyclique*
DMA *Direct Memory Access*
FIFO *First In, First Out*
FSM *Finite State Machine*
IDCT *Inverse Discrete Cosine Transform*
IP *Internet Protocol*
IP *Intellectual Property*
IQ *Inverse Quantification*
LIP6 *Laboratoire d'Informatique de Paris 6*
LSB *Less Significant bits*
MJPEG *Motion JPEG*
MAC *Media Access Control*
MPSOC *Multi-Processor System On Chip*
MPLS *Multi-Protocol Label Switching*
MSB *Most Significant bits*
MWMR *Multi-Writer Multi-Reader*
NOC *Network On Chip*

NUMA *Non Uniform Memory Access*

OSCI *Open SystemC Initiative*

OSI *Open Systems Interconnection*

PAPR *Programmation et Architecture des Processeurs Réseaux*

RIP *Routing Information Protocol*

RISC *Reduced Instruction Set Computer*

RTL *Register Transfer Level*

SOC *System On Chip*

SoCLIB *System on Chip LIBrary*

TTL *time To Live*

TTY *TeleTYpewriter*

VASY *VHDL Analyzer for SYnthesis*

VCI *Virtual Component Interface* [VSI00]

VHDL *Very High Speed Integrated Circuits Hardware Description Language*

VLD *Variable Length Decoder*

ZZ *ZigZag*

Bibliographie

[ACG+03] Adrijan Andriahantenaina, Hervé Charléry, Alain Greiner, Laurent Mortiez, and C Zeferino. SPIN : a scalable, packet switched, on-chip micro-network. In *Design Automation and Test in Europe Conference (DATE'2003) Embedded Software Forum*, pages pp. 70–73, Muenchen, Germany, march 2003.

[APDG05] Ivan Augé, Frédéric Pétrot, François Donnet, and Pascal Gomez. Platform-based design from parallel c specifications. *CAD of Integrated Circuits and Systems*, 24(12) :1811–1826, December 2005.

[Bak95] F. Baker. Requirements for ip version 4 routers. Technical report, RFC Editor, United States, 1995.

[BKK+00] J.-Y. Brunel, W. M. Kruijtzer, Kenter Kenter, F. Pétrot, L. Pasquier, Kock Kock, and W. J. M. Smits. COSY communication IP´s. In *Proceedings of the 37th Conference on Design Automation (DAC-00)*, pages 406–409, NY, June 5–9 2000. ACM/IEEE.

[CF78] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE transactions on computers*, 1978.

[CM01] Benjie Chen and Robert Morris. Flexible control of parallelism in a multiprocessor pc router. In *Proceedings of the General Track : 2002 USENIX Annual Technical Conference*, pages 333–346, Berkeley, CA, USA, 2001. USENIX Association.

[DBCP97] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. In *SIGCOMM '97 : Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 3–14, New York, NY, USA, 1997. ACM Press.

[DHRA06] Matthew Drake, Henry Hoffman, Rodric Rabbah, and Saman Amarasinghe. Mpeg-2 decoding in a stream programming lan-

guage. In *International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, April 2006.

[dKSvdW+00] E. A. de Kock, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. A. Vissers, and G. Essink. Yapi : application modeling for signal processing systems. In *DAC '00 : Proceedings of the 37th conference on Design automation*, pages 402–405, New York, NY, USA, 2000. ACM Press.

[FGG06] Etienne Faure, Alain Greiner, and Daniela Genius. A generic hardware/software communication mechanism for Multi-Processor System on Chip, targeting telecommunication applications. In *ReCoSoC'06 : Proceedings of the 2006 conference on Reconfigurable Communication-centric SoCs*, 2006.

[HHG99] J. Hennessy, M. Heinrich, and A. Gupta. Cache-coherent distributed shared memory : Perspectives on its development and future challenges. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3) :418–429, 1999.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74 : Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[KMC+00] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3) :263–297, 2000.

[LIP] LIP6, ASIM. Alliance : a complete set of free CAD tools and portable libraries for VLSI design. Technical report, LIP6, Université Pierre et Marie Curie. URL=http ://www-asim.lip6.fr/recherche/alliance/.

[Mal98] G. Malkin. Routing information protocol, version 2. Technical report, RFC Editor, United States, 1998.

[MPGS06] Ivan Miro Panades, Alain Greiner, and Abbas Sheibanyrad. A low cost network-on-chip with guaranteed service well suited to the gals approach. In *International Conference on Nano-Networks (NanoNet'06)*, Lausanne, Switzerland, Sept 2006.

[Ope96] Open Microprocessor systems initiative. Draft Standart OMI 324 : Pi-Bus. Technical report, OMI, April 1996. URL=http ://www.omimo.be/public/data/_indstan.htm#OMI324.

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

[PGH03] F. Pétrot, P. Gomez, and D. Hommais. Lightweight implementation of the POSIX threads API for an on-chip mips multiprocessor with VCI interconnect. In A. A. Jerraya, S. Yoo, D. Verkest, and N. Wehn, editors, *Embedded Software for SoC*, part 1, chapter 3, pages 25–38. Kluwer Academic Publisher, November 2003.

[PPB02] Pierre Paulin, Chuck Pilkington, and Essaid Bensoudane. Stepnp : A system-level exploration platform for network processors. *IEEE Des. Test*, 19(6) :17–26, 2002.

[PPB03] Pierre G. Paulin, Chuck Pilkington, and Essaid Bensoudane. Network processing challenges and an experimental npu platform. In *DATE '03 : Proceedings of the conference on Design, Automation and Test in Europe*, page 20064, Washington, DC, USA, 2003. IEEE Computer Society.

[RVC01] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching architecture. Technical report, RFC Editor, United States, 2001.

[SOC03] SOCLIB Consortium. Projet SOCLIB : Plate-forme de modélisation et de simulation de systèmes integrés sur puce (the SOCLIB project : An integrated system-on-chip modelling and simulation platform). Technical report, CNRS, 2003. http ://soclib.lip6.fr.

[SRH+03] Naresh Soni, Nick Richardson, Lun-Bin Huang, Suresh Rajgopal, and George Vlantis. Npse : A high performance network packet search engine. In *DATE '03 : Proceedings of the conference on Design, Automation and Test in Europe*, page 20074, Washington, DC, USA, 2003. IEEE Computer Society.

[Tan95] A. Tanenbaum. *Distributed Operating Systems*, pages 169–185. Prentice Hall, 1995.

[TKA02] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt : A language for streaming applications. In *Computational Complexity*, pages 179–196, 2002.

[vdWLG+99] Pieter van der Wolf, Paul Lieverse, Mudit Goel, David La Hei, and Kees Vissers. An mpeg-2 decoder case study as a driver for a system level design methodology. In *CODES '99 : Proceedings of the seventh international workshop on Hardware/software codesign*, pages 33–37, New York, NY, USA, 1999. ACM Press.

[VSI00] VSI Alliance. Virtual Component Interface Standard (OCB 2 2.0). Technical report, VSI Alliance, August 2000. URL=http ://www.vsi.org/library/specs/ summary.htm#ocb.