Étude des interfaces logicielles/matérielles dans le cadre des systèmes multiprocesseurs monopuces et des modèles de programmation parallèle de haut niveau

W. Youssef

Université Joseph Fourier, Informatique et Mathématiques Appliquées

N° attribué par la bibliothèque
|__/__/__/__/__/__/__/__/__/__|

THESE

pour obtenir le grade de

DOCTEUR DE l'UNIVERSITÉ JOSEPH FOURIER

Spécialité : Informatique

préparée au laboratoire **TIMA** dans le cadre de L'école doctorale
« Mathématique, Sciences et Technologies de l'Information (informatique)»

présentée et soutenue publiquement par

Mohamed Wassim YOUSSEF

Le 10 Mars 2006

Titre

Étude des interfaces logicielles/matérielles dans le cadre des systèmes multiprocesseurs monopuces et des modèles de programmation parallèle de haut niveau

---

Jury

Mr.	Guy MAZARÉ,	Président,
Mr.	Emmanuel CASSEAU,	Rapporteur,
Mr.	Flavio Rech WAGNER,	Rapporteur,
Mme.	Anne-Marie FOUILLIART,	Examinatrice,
Mr.	Ahmed Amine JERRAYA,	Directeur de thèse.

Remerciements

Je tiens à exprimer mes remerciements à toutes les personnes qui ont plus au moins directement contribués à l'accomplissement de cette thèse. Je remercie :

Mr Ahmed Amine Jerraya, directeur de recherche au CNRS, chef du groupe SLS et mon directeur de thèse, de m'avoir accueilli dans son groupe, guidé et orienté dans mon travail. Je le remercie pour sa patience, ses encouragements et ses vaillants conseils.

Mr. Guy Mazaré, professeur à l'institut national polytechnique de Grenoble, de m'avoir fait l'honneur de présider mon jury de thèse.

Mr. Emmanuel Casseau, maître de conférence à l'Université Bretagne Sud et Mr. Flavio Rech Wagner, professeur à l'Universidade Federal do Rio Grande do Sul d'avoir accepté d'être les rapporteurs de mes travaux de thèse. Je les remercie aussi pour leurs conseils en vu de l'amélioration de ce manuscrit.

Mme Anne-Marie Fouilliart, ingénieur à THALES, d'avoir accepté de participer à mon jury de thèse.

Je voudrais aussi remercier Mr Sungjoo Yoo, chef de projet à Samusung Corea, pour son encadrement et ses conseils lors des deux premières années de thèse.

Je profite aussi de cet espace pour exprimer mes remerciements à toutes les personnes que j'ai côtoyées au quotidien pendant ces années de thèse :

Mes collègues de bureau Lobna Kriaa, Aimen Bouchhima et Iuliana Bacivarov pour l'ambiance qui règne dans le bureau, pour toutes les discussions que nous avons eu et pour leurs soutiens et leurs disponibilités. Je vous souhaite à tous tout le bonheur du monde pour vos nouvelles vies.

A Mr. Nacer Zergainoh et Mr. Frederic Rousseau pour leurs valeureux conseils et encouragements.

A Mme Sonja Amadou, pour sa joie de vie son aide et ses recommandations très précieuses.

Aux trois générations de thésards du groupe SLS que j'ai eu le plaisir de connaître et de travailler ou discuter avec. D'abord les anciens, Gabriela Nicolescu, Ferid Garsalli, Damien Lyonnard, Lovic Gauthier (ceux qui savaient la vraie réalité d'une thèse mais qui n'ont rien dit). Puis les contemporains Arif Sassongko, Yanick Paviot, Adriano Sarmento, Frederic Hunsinger, Arnaud Grasset, Ivan Petkov (ceux qui savent maintenant). Et puis les actuels ou futurs thésards du groupe : Katalin Popovici, Marius Bonaciu, Youssef Atat, Benaoumeur Senouci, Alexandre Chureau, Patrice Gerin (ceux qui l'apprenent).

Mes amis des autres groupes de TIMA, Faiza Kaddour, Amel Zenati, Abdelaziz Ammari, Kamel Slimani, Karim Hadjiat. Mes amis externes à TIMA, Badice Boualllègue, Abdelaziz ben Slimane, Toufik ouled Kachroum, Kamel Mnisri, Haithem Nagati, Taher Jarboui et Hatem Zidi pour leur éternel soutien et les agréables moments que j'ai partagé avec eux à Grenoble.

Finalement, je tiens à exprimer mes remerciements les plus sincères à Mlle Leila Temim pour ses encouragements et son aide précieuse lors de la relecture de ce manuscrit.

…Un grand merci à tous !

Dédicace

A ma mère, à mon père,

Aujourd'hui je suis très fière de pouvoir enfin vous offrir le fruit de plusieurs années de travail. J'ai la jouissance de vous dédier ce travail, veuillez y trouvez le témoignage de mon grand amour et de ma profonde reconnaissance.

Je vous souhaite la bonne santé, la joie de vivre et que dieu vous garde.

A Imen, Ines et Wajih ,

Avec tous mes vœux de bonheur et de santé, je vous souhaite un avenir radieux plein de succès et de gloire. Je vous dédie ce travail en témoignage de ma grande affection.

A ma grande famille, mon inépuisable source d'encouragements et de soutient.

Plan du mémoire

Liste des figures

Liste des Tableaux

Chapitre I

Introduction générale

I.1 Contexte : conception des systèmes multiprocesseurs monopuces

Les progrès constants réalisés dans la technologie de miniaturisation permettent l'intégration de systèmes complets dans une seule puce de silicium. Ces puces, on les retrouve de plus en plus dans la vie courante, dans des applications diverses et variées. Elles équipent les nouvelles générations de téléphones cellulaires, assurent des fonctions critiques dans nos voitures (gestion du freinage ABS, du déclenchement des airbags), forment le cœur des nouvelles consoles de jeux (PlayStation3 de Sony) et sont primordiales pour la majorité des appareils multimédias comme les lecteurs/encodeurs vidéo portables, les appareils photo numériques.

Avec le besoin croissant en performance et la possibilité offerte par le pouvoir de miniaturisation, les systèmes embarqués monopuces sont maintenant dotés d'une architecture matérielle multiprocesseur. Ils sont appelés systèmes MPSoC, pour *multiprocessor system on chip*. L'architecture matérielle des systèmes MPSoC peut être décomposée en quatre blocs de base : (1) processeur ou sous-système processeur pour exécuter le logiciel, (2) modules mémoire ou unités de stockage de données, (3) sous-système de calcul composé de matériel spécifique et (4) un réseau d'interconnexion. Il y a eu un développement et une sophistication continus de chacun de ces blocs de base, mais c'est surtout leur arrangement, qui différencie un système MPSoC d'un autre.

Les systèmes embarqués peuvent inclure plusieurs processeurs, qui exécutent des instructions spécifiques implémentées en logiciel pour des besoins de flexibilité. On estime alors que dans un futur proche, la complexité du code logiciel sera supérieure à celle de la partie matérielle et demandera par conséquent plusieurs hommes-années de durée de conception. Le logiciel ne pourra donc plus être développé en langage assembleur et une approche de conception à un niveau d'abstraction plus élevé est requise.

Les systèmes MPSoC sont donc constitués d'une partie matérielle et d'une partie logicielle. L'interface entre ces deux parties, notée interface logicielle/matérielle, affecte directement la performance du système final car c'est elle qui permet l'exécution de la partie logicielle sur la partie matérielle. Ce travail s'inscrit dans le cadre de la conception des interfaces logicielles/matérielles pour les systèmes MPSoC.

La conception de l'interface logicielle/matérielle est un travail complexe :

- Elle nécessite des concepteurs aux compétences hétérogènes, qui maîtrisent des techniques de programmation logicielle et de conception de circuits matériels.
- Elle nécessite des outils d'aide à la conception, qui couvrent aussi bien l'aspect logiciel que l'aspect matériel. Ces outils doivent pouvoir être utilisés pour des types d'applications différents (différents langages de programmation et support d'architectures matérielles diverses et hétérogènes). De plus, une certaine automatisation de la conception est requise pour réduire le temps total de conception d'un système MPSoC.

I.2 Objectif : un flot de conception des interfaces logicielles/matérielles pour les systèmes MPSoC basé sur l'utilisation et le raffinement des modèles de programmation parallèle

L'objectif général de cette thèse s'inscrit dans le cadre de la définition d'un flot pour la conception des systèmes MPSoC et la réalisation d'un outil CAO pour l'automatisation de cette conception.

On appellera « modèle de programmation parallèle » toute abstraction des interfaces logicielles/matérielles associée à un modèle d'exécution. Le modèle de programmation sera vu par le logiciel applicatif comme une interface (API + sémantique d'exécution) permettant d'interagir avec le matériel. Le modèle de programmation peut aussi être vu comme la spécification de la plateforme d'exécution matérielle et des couches basses du logiciel (figure I-1).

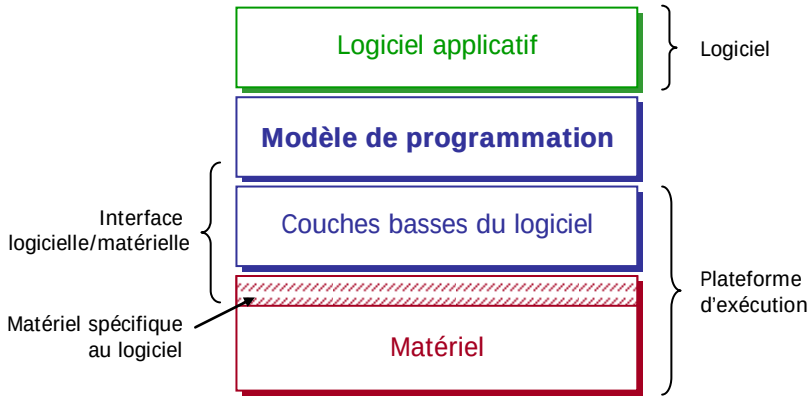


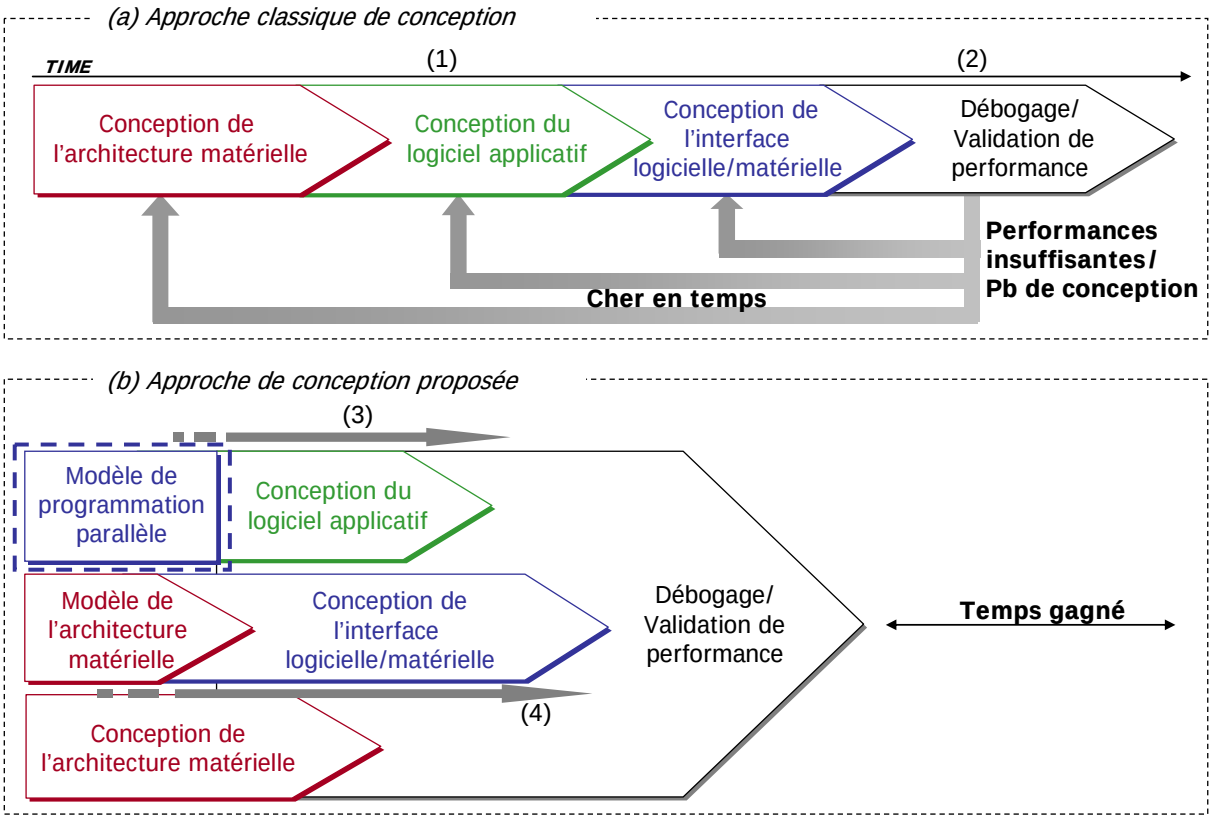
Figure I-1 : Le modèle de programmation en tant qu'abstraction du matériel associée à une plateforme d'exécution

L'interface logicielle/matérielle est constituée par les couches basses du logiciel et parfois une partie du matériel conçue spécifiquement pour le logiciel. Dans cette thèse on se limitera aux parties logicielles de l'interface et on considérera que le matériel est figé.

La figure I-2 – (a) présente l'approche de conception classique des systèmes MPSoC. Cette approche est basée sur une chronologie séquentielle dans le temps : conception de la partie matérielle, suivie de la conception du logiciel applicatif, puis de la conception des interfaces logicielles/matérielles et finalement une étape de débogage et de validation des performances. Dans cette approche classique, la conception du logiciel applicatif commence tardivement, après celle de la partie matérielle. Ceci est dû au fait que le logiciel applicatif est fortement optimisé vis-à-vis du matériel, et par conséquent il est dépendant du matériel. Aussi, l'approche classique repousse au dernier moment le débogage et l'évaluation de la performance du système entier. Il en résulte une phase de débogage plus complexe et des boucles vers des étapes de conception précédentes très coûteuses en temps.

La figure I-2 – (b) présente l'approche de conception proposée pour la conception des systèmes MPSoC. Cette approche est basée sur l'utilisation d'un modèle de programmation parallèle de haut niveau d'abstraction et d'un modèle de l'architecture matérielle :

- Le modèle de programmation parallèle permet de commencer le développement du logiciel applicatif sans attendre que la partie matérielle ne soit prête. Il permet aussi d'avoir des estimations de la performance du système final. De plus, à cause d'un niveau d'abstraction plus élevé, l'utilisation d'un modèle de programmation parallèle permet de concevoir un logiciel applicatif portable qui peut être réutilisé sur d'autres architectures.
- Le modèle de l'architecture matérielle permet de commencer la conception de l'interface logicielle/matérielle avant même la fin de la conception de la partie matérielle. La conception de l'interface logicielle/matérielle est réalisée en implémentant le modèle de programmation parallèle sur le modèle de l'architecture matérielle.



(1) Début tardif de la conception du logiciel ; (2) Prise en compte très tardive de la performance du système ; (3) L'utilisation d'un modèle de programmation parallèle permet de déboguer l'application et renseigne sur sa performance sans attendre la disponibilité de la partie matérielle ; (4) Débogage et évaluation des performances de l'interface logicielle/matérielle avancés dans le temps

Figure I-2 : Approches de conception d'un système MPSoC

Ce travail est axé sur l'étude des modèles de programmation parallèle et de leurs utilisations pour la conception des interfaces logicielles/matérielles. Au cours du raffinement des interfaces logicielles/matérielles, il faut décider de la communication entre les différents processus et de son implémentation. Les processus communiquent de différentes manières, toutefois limitées par les possibilités de l'architecture matérielle. Il existe trois principales manières :

- L'échange de messages : Quand un processus veut communiquer avec un autre, il lui envoie un message. La communication peut être bloquante (l'émetteur doit attendre que son message soit reçu avant de continuer) ou non bloquante. Dans ce dernier cas, des méthodes de synchronisation doivent être prévues afin d'éviter la perte de messages.
- Transfert par mémoire partagée : dans les architectures à mémoire partagée, la communication est établie par le fait que le processus expéditeur place les données à une adresse bien déterminée à partir de laquelle le processus récepteur peut les lire. La difficulté consiste à détecter quand les données sont valides et correctes pour être lues ou modifiées. Les techniques standard sont basées sur l'utilisation de sémaphores ou de verrous. Cependant elles sont coûteuses et compliquées à programmer. Quelques architectures fournissent un support matériel de ces techniques permettant de réduire le coût de leur utilisation.
- Accès direct à une mémoire distante : les premières architectures à base de mémoire distribuée nécessitaient que le processeur soit interrompu pour chaque requête d'accès à la mémoire. Ceci entraîne une utilisation très faible du processeur. Depuis, les nouvelles architectures contiennent un second processeur, ou coprocesseur, qui est responsable de la gestion des accès à la mémoire. Ce processeur gère le trafic et l'accès au réseau de communication. À la limite, ce mécanisme permet un accès distant direct à la mémoire distante.

Ces mécanismes de communication ne doivent pas forcément correspondre de façon directe à ce que fournit l'architecture matérielle. Il est courant de simuler une communication par échange de messages sur une architecture à base de mémoire partagée et il est aussi possible de simuler une communication à mémoire partagée en utilisant un échange de messages.

I.3 Contributions

Ce travail présente trois contributions : (1) étude de l'architecture des interfaces logicielles/matérielles ; (2) étude des modèles de programmation parallèle en vue de leur utilisation pour la conception des systèmes MPSoC et proposition d'un flot de conception de systèmes MPSoC ; (3) réalisation de deux expériences de conception (un encodeur vidéo et une radio définie par logiciel vers des architectures matérielles multiprocesseurs).

Chaque contribution sera présentée plus en détail par la suite.

I.3.1 Étude de l'architecture et de l'implémentation des interfaces logicielles/matérielles

Cette étude détaille l'architecture des interfaces logicielles/matérielles. Cette interface est décomposée en deux parties, l'adaptation logicielle et l'adaptation matérielle. L'adaptation logicielle est structurée en couches pour des besoins de flexibilité et de portabilité. Les couches constituant l'adaptation logicielle sont le middleware de programmation parallèle, le système d'exploitation et la couche d'accès au matériel. La partie matérielle est constituée de l'adaptateur du sous-système

processeur et de l'interface de communication avec le réseau. Cette étude sera détaillée dans le chapitre II.

La partie la plus importante de cette étude concerne l'adaptation logicielle. Nous discutons notamment de l'implémentation de l'adaptation logicielle où les différentes couches sont fusionnées pour constituer le logiciel dépendant du matériel. Cette fusion est requise pour optimiser l'adaptation logicielle vis-à-vis de l'architecture matérielle. Le chapitre III détaillera les approches pour l'implémentation de l'adaptation logicielle.

I.3.2 Étude des modèles de programmation parallèle et proposition d'un flot de conception pour les systèmes MPSoC

Nous avons réalisé une étude des modèles de programmation parallèle en vue de leurs utilisations pour la conception des systèmes MPSoC. Dans cette étude, nous avons classé les modèles suivant leurs degrés d'abstraction puis nous avons détaillé des modèles de programmation pour MPSoC utilisés dans l'industrie.

Nous avons proposé un flot basé sur l'utilisation des modèles de programmation parallèle pour la conception des interfaces logicielles/matérielles des systèmes MPSoC (voir figure I-3). Ce flot se distingue par une conception automatique de l'interface logicielle/matérielle, sans pour autant être limité à une classe d'architectures matérielles unique.

Dans la figure I-3, les ovales représentent des étapes de conception et les rectangles représentent une entrée et/ou une sortie des étapes de conception.

Le flot proposé est composé de sept étapes permettant la conception des systèmes MPSoC. Pour concevoir la partie logicielle, la première étape est la parallèlisation et la transposition de la spécification de l'application en tenant en considération quelques paramètres de l'architecture matérielle comme le nombre de sous-système processeur. Le modèle de l'application obtenu est débogué en haut niveau pour valider le logiciel applicatif. L'étape suivante est la conception du logiciel dépendant du matériel. Une fois conçu, il est débogué avec le logiciel applicatif en utilisant un prototype de l'architecture matérielle. La dernière étape de conception du logiciel est son intégration avec le matériel qui produit un système MPSoC. En parallèle à ces étapes de conception de la partie logicielle, se déroulent les étapes de conception d'un prototype matériel et du matériel final.

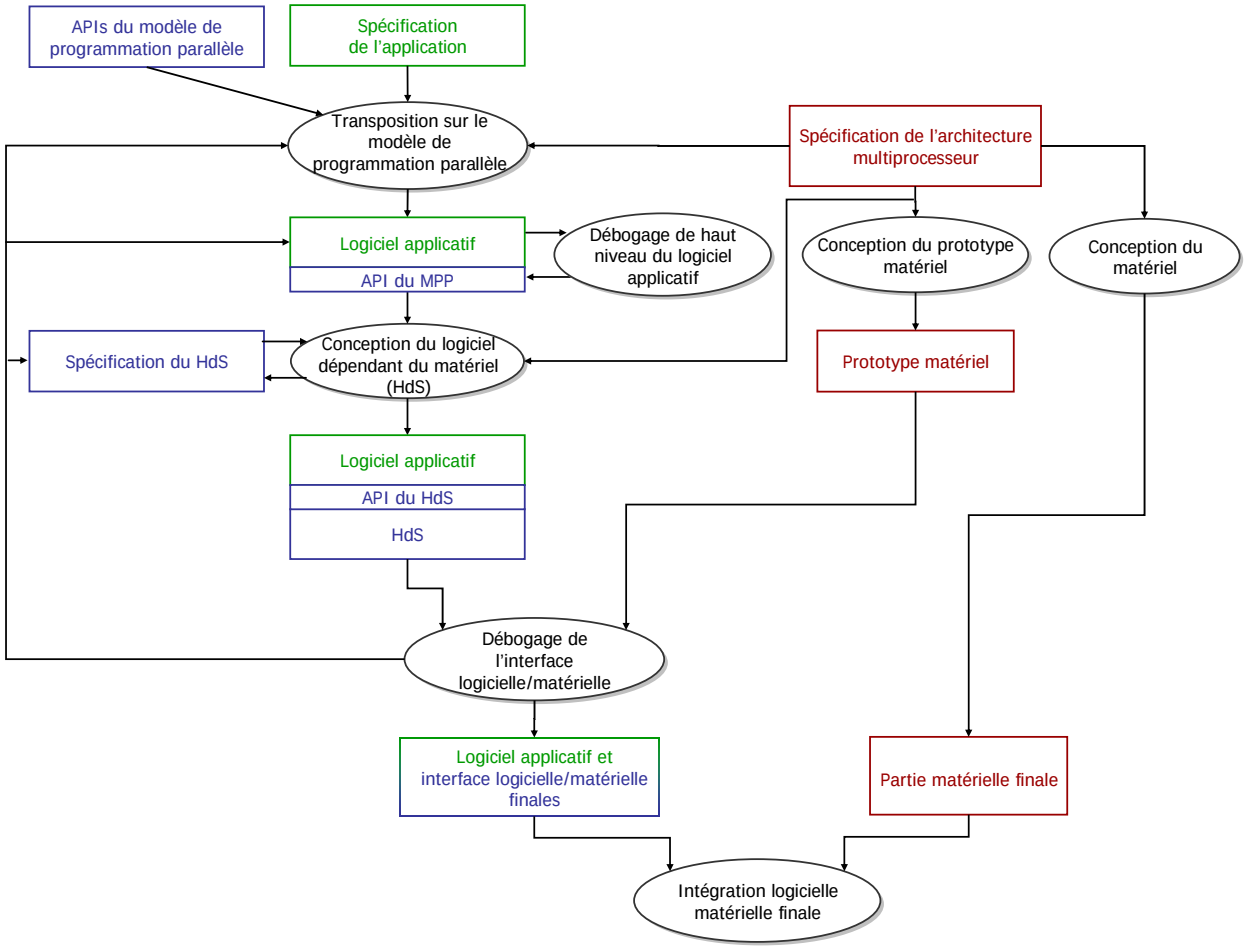


Figure I-3 : Flot de conception proposé

Ce flot, ainsi que l'étude des modèles de programmation parallèles seront détaillés dans le chapitre IV.

I.3.3 Application du flot pour la conception d'interfaces logicielles/matérielles pour architectures MPSoC en partant d'une application décrite avec un modèle de programmation parallèle de haut niveau

Conception d'un encodeur vidéo en utilisant le modèle de programmation parallèle MPI pour l'architecture multiprocesseur ARM Integrator

Une expérience de conception d'un encodeur vidéo [OpenDivX] vers la plateforme multiprocesseur [ARM Integrator] a été réalisée. Nous avons utilisé [MPI] comme modèle de programmation parallèle pour la spécification fonctionnelle de l'encodeur, puis nous avons raffiné automatiquement les primitives MPI utilisées sur la plateforme ciblée. Cette expérience nous a permis d'identifier que le débogage de l'interface logicielle/matérielle est un goulet d'étranglement pour la conception des systèmes MPSoC. Nous avons aussi proposé des approches pour remédier à ce goulet. Cette expérience sera présentée dans le chapitre V.

Conception d'une radio logicielle en utilisant le modèle de programmation parallèle CORBA pour une architecture multiprocesseur spécifique

Une seconde expérimentation a été entamé. Elle concerne l'application du flot proposé pour la conception d'une radio logicielle en utilisant [CORBA] comme modèle de programmation parallèle pour une architecture multiprocesseur spécifique. Nous avons réalisé un modèle de simulation de haut niveau de l'application et nous avons présenté des perspectives pour le raffinement automatique du modèle de programmation parallèle sur l'architecture ciblée. Cette expérience sera présentée dans le chapitre VI.

I.4 Plan du document

Outre l'introduction générale, ce document est composé de six autres chapitres. Le chapitre II présente l'étude de l'architecture des interfaces logicielles/matérielles pour les systèmes MPSoC. Le chapitre III discute des approches pour l'implémentation de l'adaptation logicielle. Le chapitre IV présente une analyse des modèles de programmation parallèle et un flot de conception de MPSoC basé sur l'utilisation des modèles de programmation parallèle. Le chapitre V détaille l'expérience de conception de l'encodeur vidéo [OpenDivX] sur la plateforme [ARM Integrator] en utilisant le modèle de programmation parallèle [MPI]. Le chapitre VI détaille la seconde expérience de conception. Elle consiste en la conception d'une radio logicielle basée sur le modèle [CORBA] pour une architecture matérielle spécifique. Le chapitre VII conclut ce document.

Chapitre II

Études des architectures des interfaces logicielles/matérielles pour les systèmes monopuces multiprocesseurs

II.1 Introduction

Un système MPSoC est composé de trois parties : une partie logicielle, une partie matérielle et une troisième partie qui réalise l'interface entre les deux premières. La figure II-1 présente une vue abstraite d'un système MPSoC. La partie logicielle est composée par un ensemble de tâches (T1…Tn) tandis que la partie matérielle est composée de plusieurs unités d'exécutions (UE) (processeurs, processeurs de traitement du signal, mémoires, IP matériel spécifique…) et d'un réseau de communication. L'interface logicielle/matérielle est le moyen qui permet l'interaction entre les différentes tâches (partie logicielle) et leurs exécutions sur les différentes UE (partie matérielle). Sa conception est une étape très importante pour le succès d'un système MPSoC.

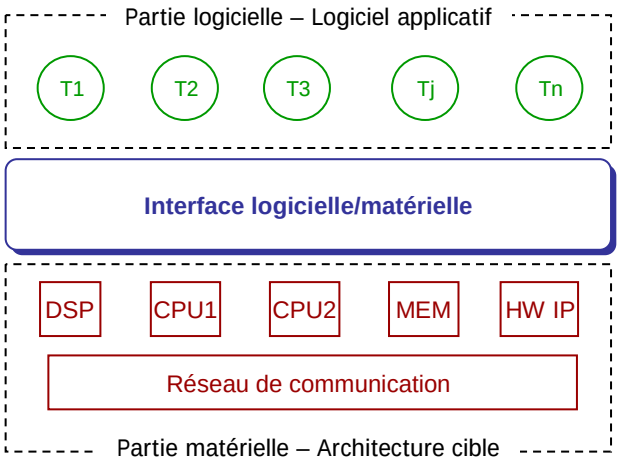


Figure II-1 : Vue abstraite d'un MPSOC

La conception des systèmes MPSoC doit répondre à divers critères, qui sont parfois contradictoires. Le concepteur d'un MPSoC doit alors satisfaire plusieurs compromis :

- Maximiser la performance de calcul, minimiser la consommation d'énergie.
- Maximiser la flexibilité (accroître la taille de la partie logicielle), minimiser la surface (réduire la taille des mémoires).
- Optimiser le système MPSoC et réduire son coût, produire plus vite.

Dans ce chapitre nous étudions l'architecture des interfaces logicielles/matérielles pour les systèmes monopuces multiprocesseurs. Nous présentons tout d'abord les composantes d'un système MPSoC spécifique à l'application. Puis, nous détaillons l'architecture conceptuelle des interfaces logicielles/matérielles. Nous constatons alors qu'elle est formée d'une adaptation logicielle et d'une adaptation matérielle.

II.2 Composantes d'un système MPSoC spécifique à l'application

Dans ce paragraphe, nous présentons la partie logicielle et la partie matérielle d'un système MPSoC. Puis, nous détaillons l'interface logicielle/matérielle entre la partie logicielle et la partie matérielle.

II.2.1 Partie logicielle

La partie logicielle d'un système MPSoC correspond au code logiciel de l'application (ou logiciel applicatif). Ce code logiciel est écrit en langage de haut niveau pour maîtriser la complexité de la conception. L'application peut être décrite comme un ensemble de tâches qui communiquent entre-elles.

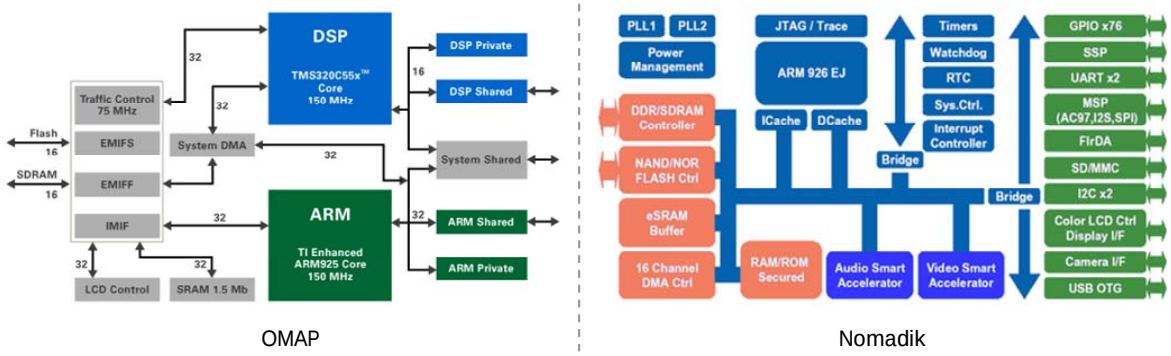
L'[ITRS] prévoit que les systèmes MPSoC seront dominés par leurs parties logicielles. Cette place accrue du logiciel dans un MPSoC est due primordialement à la complexité des applications. Des applications comme la téléphonie de troisième génération (3G), le multimédia (encodage vidéo), ne peuvent plus être conçus sur des ASIC : il en résulterait un circuit très complexe qui ne peut être prêt dans les délais de conception imposés. D'autres applications, comme la radio logicielle, tirent profit de la flexibilité du logiciel (par rapport à la rigidité du matériel) pour établir des connexions radio ayant des normes différentes avec ou sans cryptage de données, le cryptage lui-même peut être paramétrable, en utilisant un seul et même circuit MPSoC.

II.2.2 Partie matérielle

La partie matérielle d'un système MPSoC correspond à l'architecture matérielle cible de l'application. Elle est composée par un ensemble d'unités d'exécutions (processeurs, DSP, sous-systèmes processeur) interconnectés via un réseau de communication. Un sous-système processeur est composé par des unités d'exécution ou de traitement de données, de la mémoire et une interface réseau. La figure II-2 – (b) présente un exemple de l'architecture locale d'un sous-système processeur. Elle contient un processeur, une mémoire, un contrôleur d'interruptions programmable, un contrôleur DMA, un *timer* et une interface réseau interconnectés via un bus local. D'autres composants de l'architecture locale peuvent être l'unité de gestion de la mémoire (MMU, pour *memory management unit*), la mémoire cache, des dispositifs d'entré/sortie, etc…

Les puces [Nexperia] de Philips, [OMAP] de Texas Instruments, [Nomadik] de ST Microelectronics ou bien les puces plus récentes comme [Cell] du trio Sony-Toshiba-IBM, [Niagara] de Sun ou [BCM1480] de Broardcom sont des exemples de partie matérielle d'un MPSoC.

La figure II-3 présente une vue simplifiée de l'architecture des puces OMAP et Nomadik. La version OMAP 5910, contient un processeur ARM925 et un DSP TMS320C55x, chacun possède un bus de périphérique qui lui est propre. La puce Nomadik est architecturée autour d'un processeur ARM926EJ et d'un accélérateur vidéo et un autre audio.



OMAP Nomadik

Figure II-3 : Vue simplifiée de l'architecture des puces OMAP et Nomadik

Dans le cas des systèmes MPSoC spécifiques à l'application, les composants de l'architecture matérielle sont choisis en fonction des besoins de l'application. Les choix peuvent porter par exemple sur le type des processeurs, le nombre et l'architecture de chaque sous-système processeur ou la topologie et le protocole du réseau de communication...

II.2.3 Interfaces logicielles/matérielles

Les interfaces logicielles/matérielles jouent le rôle d'une « colle » entre la partie logicielle et la partie matérielle, elles permettent l'exécution du logiciel applicatif sur l'architecture matérielle cible. Les interfaces logicielles/matérielles sont elles aussi composées de deux parties (figure II-2 – (a)) : une partie logicielle, qu'on appellera adaptation logicielle, et une partie matérielle, qu'on appellera adaptation matérielle. L'interface ultime entre le « monde logiciel » (i.e. logiciel applicatif + adaptation logicielle) et le « monde matériel » (i.e. architecture matérielle cible + adaptation matérielle) est le processeur (ou le sous-système processeur).

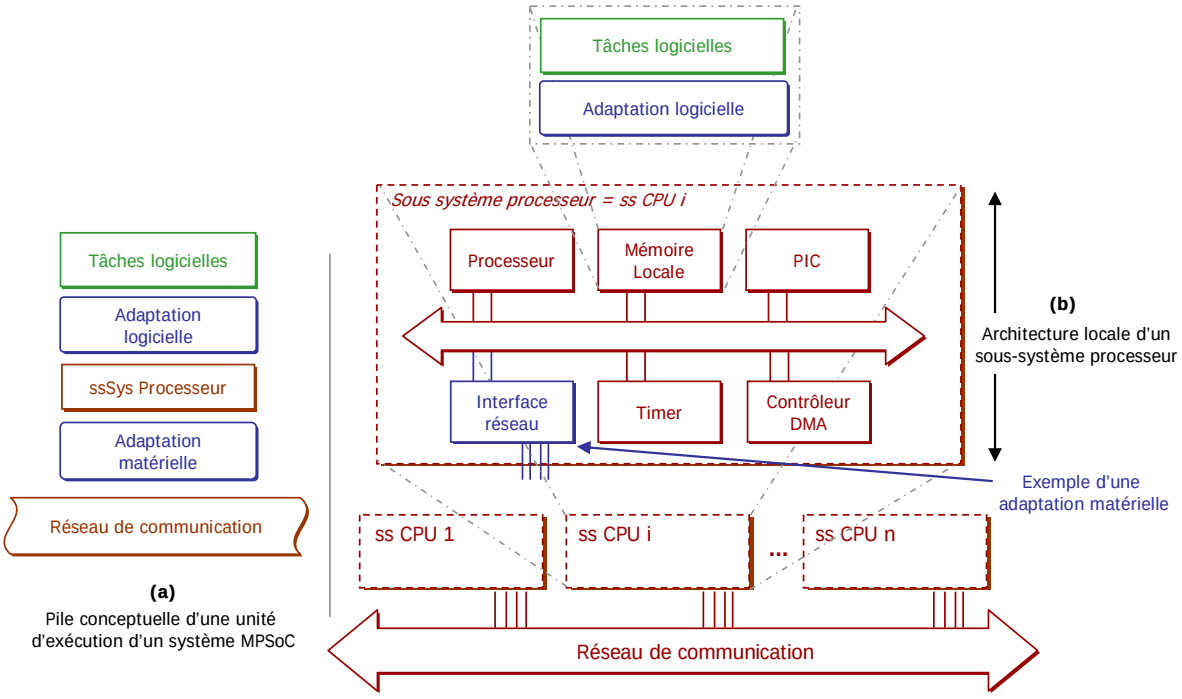


Figure II-2 : Architecture d'un système MPSoC

Évolution des interfaces logicielles/matérielles

On peut identifier trois périodes temporelles de l'évolution des systèmes monopuces (figure II-4, adaptée de [Schirrmeister 02]) :

- Dans la première période, des années 70 jusqu'aux années 80, la conception des systèmes embarqués était basée sur des modèles de transistors et de portes logiques. Puis, dans les années 80, elle était basée sur des modèles du niveau transfert de registres (RTL). Pendant cette première période la conception était 100% matérielle, les interfaces logicielles/matérielles n'existaient pas.
- La seconde période s'étend sur les années 90. Au cours de cette période, un premier processeur a été intégré sur la puce. La présence du processeur a requis le développement d'une couche d'adaptation matérielle pour permettre la communication avec les autres composants matériels, ainsi qu'une couche d'adaptation logicielle pour permettre l'exécution des tâches logicielles sur le processeur. Dans cette seconde période, la conception du logiciel et de l'interface logicielle/matérielle (adaptations logicielles et matérielles) se sont ajoutées à la conception du matériel.

Au début de la seconde période, le logiciel était « simple » et écrit à un bas niveau d'abstraction en utilisant le langage d'assembleur du processeur. L'utilisation du langage d'assemblage du processeur requiert uniquement la compilation du code logiciel avant son chargement et son exécution sur le système embarqué. Le code est utilisable de suite sans développement d'une adaptation logicielle, néanmoins certains paramètres doivent être passés au compilateur comme le plan mémoire pour générer le code binaire.

A la fin de la seconde période, l'utilisation des systèmes embarqués dans des d'applications plus complexes a requis une élévation du niveau d'abstraction du logiciel, d'ou l'utilisation de langages de programmation de haut niveau. Le code logiciel de haut niveau interagit avec le matériel en utilisant des appels des pilotes de matériel (procédures de bas niveau, écrites en assembleur). De plus, pour simplifier la programmation des applications les plus complexes, des systèmes d'exploitation (voir paragraphe II.3.2) ont fait leurs apparitions dans les systèmes embarqués. Ils permettent par exemple une programmation multitâche. Le système d'exploitation et les pilotes réalisent l'adaptation logicielle et permettent au code logiciel de haut niveau de l'application de s'exécuter correctement sur le matériel.

Déjà à cette période, la conception de l'adaptation logicielle est un défi : c'est un travail complexe qui requiert des compétences mixtes issues du domaine de la programmation logicielle et matérielle ainsi qu'une bonne maîtrise de l'architecture matérielle utilisée. De plus, c'est un travail qui est source d'erreurs et qui pour des raisons commerciales (réduction du temps de mise sur le marché) doit être réalisé le plus rapidement possible.

- La dernière période est en cours actuellement. Elle est caractérisée par un nombre élevé de sous-systèmes processeur hétérogènes intégrés sur la même puce. Un réseau de communication complexe est généralement utilisé pour interconnecter les différents sous-systèmes processeur. Cette période se différentie des deux précédentes par une complexité accrue de la conception des interfaces logicielles/matérielles.

Les systèmes MPSoC conçus dans cette période sont qualifiés de « hautement parallèles » car leurs architectures leur permettent d'exécuter plusieurs tâches en parallèle : plusieurs tâches peuvent être exécutées au niveau d'un sous-système processeur (parallèlisme d'exécution interne) et plusieurs tâches peuvent être exécutées, au même moment, au niveau de sous-système processeur différents (parallèlisme d'exécution externe ou globale).

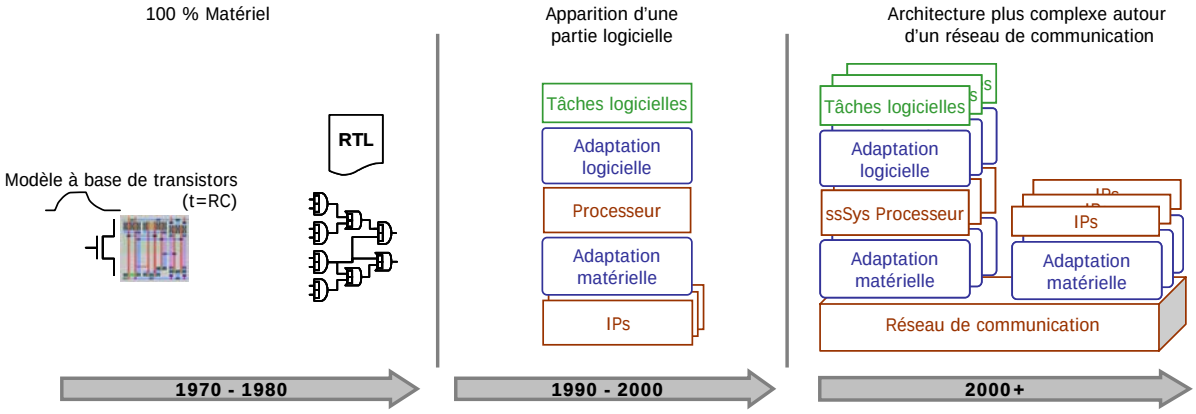


Figure II-4 : Évolution des systèmes monopuces

Remarque concernant la conception des systèmes embarqués

Actuellement, on maîtrise la conception d'un logiciel complexe pour des architectures multiprocesseurs classiques. On arrive à gérer un grand nombre d'utilisateurs, à exécuter plusieurs tâches en concurrence sur un seul processeur. On arrive notamment, grâce aux middlewares, à faire exécuter un programme de façon distribuée sur des milliers de processeurs. Du coté du matériel, on maîtrise aussi la technologie d'intégration de plusieurs processeurs sur une seul puce (le processeur [Cell] de la future PlayStation3 contenant plusieurs unités d'exécutions, sera produit à très grande échelle).

Par contre, on ne maîtrise pas encore le processus de conception de l'interface entre les parties logicielles et matérielles d'un MPSoC. La difficulté de cette tâche est due à l'écart important entre la vision du système qu'a un concepteur matériel (s'intéresse à tous les détails de son architecture) et celle du concepteur du logiciel (vision plus abstraite du système). Mais la difficulté est surtout due à la multiplicité des contraintes spécifiques au développement des MPSoC (plus de performance, plus rapidement et à moindre coût). L'interface entre le logiciel et le matériel est donc la clé de voûte pour le succès de la conception d'un MPSoC.

Restriction imposée à ce travail de recherche

Nous nous restreindrons dans le cadre de cette thèse à l'étude, la conception et la réalisation de l'adaptation logicielle des interfaces logicielles/matérielles. L'étude, la conception et la réalisation des adaptations matérielles font l'objet des travaux de thèse de [Grasset 06]. Toutefois, nous avons réalisé une expérience de conception d'une adaptation matérielle présentée dans l'annexe A.

Nous présenterons brièvement dans la suite l'adaptation matérielle. Une étude plus détaillée de l'adaptation matérielle se trouve dans [Grasset 06]. Puis nous détaillerons dans le paragraphe suivant l'architecture conceptuelle de l'adaptation logicielle.

Le rôle de l'adaptation matérielle est d'assurer la communication entre le processeur sur lequel s'exécute le logiciel (code de l'application + adaptation logicielle) et les éléments du sous-système processeur ainsi que les différents sous-systèmes processeur constituants le système via le réseau de communication.

La partie matérielle d'un système MPSoC peut être une puce commerciale [Nomadik], [OMAP], [Cell] ; ou bien développée sur mesure pour l'application. Dans le premier cas, l'adaptation matérielle de l'interface logicielle/matérielle est déjà conçue par le fournisseur du circuit. La conception de l'adaptation matérielle revient à configurer (si possible) l'adaptation matérielle fournie (la taille des buffers de communication par exemple). Dans le second cas, il s'agit de développer l'adaptation matérielle spécifique à l'application : ajout d'accélérateurs de communication (contrôleurs DMA, coprocesseur de communication, adaptateur de protocole matériel), de mémorisation intermédiaire, d'un arbitre d'accès au réseau entre les différents composants du sous système…

II.3 Architecture conceptuelle de l'adaptation logicielle dans les interfaces logicielles/matérielles

L'interface logicielle/matérielle se divise en deux parties, l'adaptation logicielle et l'adaptation matérielle. La première réalise l'interface entre le code logiciel de l'application et le sous-système processeur. La seconde réalise l'interface entre le sous-système processeur et le réseau de communication (figure II-3 – (a)).

Nous proposons une architecture conceptuelle en couches pour les interfaces logicielles/matérielles. L'adaptation logicielle se compose de trois couches (figure II-5), séparées par des APIs. Ces couches sont : la couche middleware de programmation parallèle (notée PPM pour *parallel programming middleware*), la couche système d'exploitation (notée OS pour *operating system*) et la couche d'accès au matériel (notée HAL pour *hardware access layer*). L'adaptation matérielle (figure II-5) se décompose quant à elle en deux couches : l'interface du processeur et l'interface du réseau de communication.

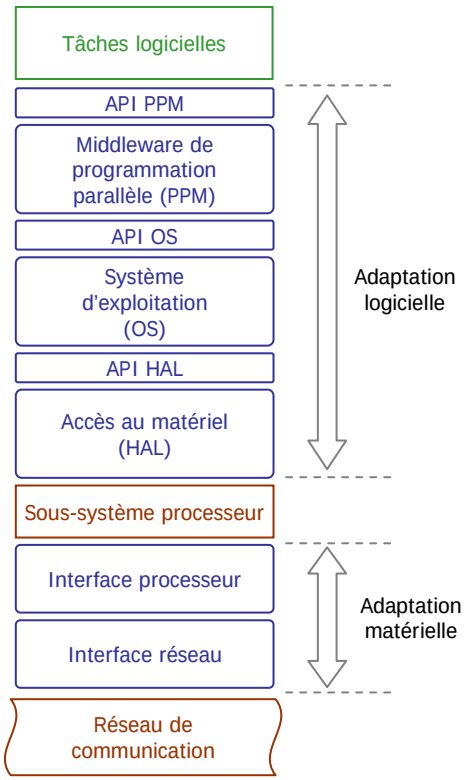


Figure II-5 : Architecture de l'interface logicielle/matérielle

Nous détaillerons, dans la suite de ce paragraphe, chacune des couches de l'adaptation logicielle.

II.3.1 Couche d'accès au matériel (HAL)

II.3.1.1 Présentation et rôle de la couche HAL

Dans cette étude, nous considérons le HAL comme la couche logicielle qui est directement dépendante du matériel sur lequel elle s'exécute [Yoo 04]. Des exemples de HAL incluent le code de l'amorçage (*boot code*), le code de changement de contexte, le code de configuration et d'accès à des ressources matérielles, comme l'unité de gestion de la mémoire (MMU), le pont entre bus (*bus bridge*), le bus sur puce, le *timer*, etc...

Comme le montre la figure II-6, le HAL apporte une abstraction du matériel du dessous (i.e. sous-système processeur) aux couches logicielles supérieures (i.e. OS/ PPM/ application). Le HAL abstrait aussi le processeur, il fourni une abstraction :

- Des types de données, sous forme de structures de données (la taille en bit d'un entier, booléen, flottant, etc...).
- Du code de l'amorçage (boot).
- Du contexte, sous forme de structures de données : format d'enregistrement des registres (par exemple R0-R14 pour le processeur ARM7).
- Des fonctions de changement de contexte (par exemple context_switch, setjmp/longjmp).
- Du changement du mode du processeur (utilisateur/noyau).
- Du masquage/démasquage, activation/désactivation des interruptions.

La figure II-6 – (a) présente un exemple d'une primitive de l'API HAL pour le changement de contexte, `__cxt_switch (cxt_type oldcxt, cxt_type newcxt)`. Cette primitive peut être utilisée pour le changement de contexte sur n'importe quel processeur. Pour un processeur donné, on a besoin d'implémenter le corps de la fonction. La figure II-6 – (b) montre aussi le code réel pour cette primitive pour un processeur ARM7.

(a)

```
1 :   // Fonction HAL API pour le changement de contexte
2 :   typedef int cxt_type [15] ;
3 :   void __cxt_switch (cxt_type oldcxt, cxt_type newcxt);
```

(b)

```
1 :   // Implémentatation de __cxt_switch spécifique à ARM7
2 :   __cxt_switch       ; r0, ancien pointeur de pile, r1, nouveau pointeur de pile
3 :   STMIA r0!,{r0-r14} ; sauvegarde les registres de la tache courante
4 :   LDMIA r1!,{r0-r14} ; restaure les registres de la nouvelle tache
5 :   SUB pc,lr,#0       ; retourne
6 :   END
```

Figure II-6 : Exemple d'une primitive de l'API HAL

Il existe dans la littérature d'autres concepts similaires au HAL : les nano-noyaux et les pilotes de périphériques. Le nano-noyau est généralement défini comme l'ensemble des routines d'interruption et des piles de tâches [Prebert 91]. Il sert de fondation pour le micronoyau du système d'exploitation. Sous cette définition le micronoyau peut être considéré comme une partie du HAL puisqu'il ne gère pas les entrées/sorties (E/S). Cependant, le nano-noyau est parfois utilisé pour représenter exactement le HAL. Par exemple dans le cas du système d'exploitation µchoice OS, le nano-noyau est équivalent au HAL.

Un pilote de périphérique fournit un accès aux E/S. Comparé au HAL, il est limité à la gestion des E/S, il ne couvre pas le changement de contexte, la gestion des interruptions, etc... Pour être plus exact, le pilote de périphérique ne fait pas entièrement partie du HAL. Dans les cas d'un pilote de périphérique, on doit le séparer en deux parties : une partie dépendante et une partie indépendante du matérielle [Wang 03]. La partie dépendante du matériel fait alors partie du HAL.

II.3.1.2 Différence avec le HAL des architectures multiprocesseurs classiques

Le HAL des architectures multiprocesseurs classiques fournit une abstraction complète de l'architecture matérielle. En fait, le HAL dépend plus du système d'exploitation que de l'architecture matérielle : la même HAL API spécifique à un système d'exploitation donné est implémentée quelle que soit l'architecture matérielle. Il s'agit d'une API HAL standard (fixe) par système d'exploitation.

Le HAL pour MPSoC dépend de l'architecture matérielle sur laquelle il s'exécute. Les primitives de l'API HAL fournissent, aux couches supérieures, uniquement les services existants offerts par le matériel. De ce fait, il n'existe pas une API HAL standard pour les systèmes MPSoC (chaque architecture matérielle est spécifique à l'application, il existera toujours une nouvelle architecture qui offre des nouveaux services non supportés par un éventuel standard HAL).

II.3.1.3 Avantages et limitations de l'utilisation d'une couche HAL pour la conception de l'adaptation logicielle

Avantages :

Le HAL fournit une abstraction de l'architecture matérielle, quand un système d'exploitation est conçu en utilisant l'API HAL, son code est portable vers une nouvelle architecture matérielle dés que l'API HAL peut être implémentée sur cette architecture. Le travail du concepteur du système d'exploitation est donc soulagé de la programmation des accès au matériel, travail difficile, fastidieux et source de nombreuses erreurs.

Le HAL permet de faire une première séparation entre la conception de la partie logicielle et la conception de la partie matérielle. Il établi une sorte de contrat entre le concepteur du logiciel et celui du matériel. Il permet de commencer au même temps la conception du logiciel et du matériel (sans attendre la phase initiale de conception du matériel nécessaire lors d'une conception sans HAL).

Limitations :

Il n'existe pas un HAL unique ou standard pour les systèmes MPSoC à cause de la spécificité de chaque architecture matérielle à une application donnée. La portabilité des couches logicielles supérieures au HAL est donc réduite.

II.3.1.4 Exemples de HAL pour les systèmes MPSoC

La notion de HAL pour les systèmes d'exploitation industriels et commerciaux se limite le plus souvent aux composants matériels du sous-système processeur, autres que le processeur. En fait, ces systèmes d'exploitation sont livrés pour une version bien déterminée du processeur cible. Ils incluent par exemple le code de l'amorçage et de changement de contexte spécifiques au processeur cible, qui font partie normalement du HAL. Cependant, le HAL au sens présenté dans II.3.1.1 est utilisé par les éditeurs du système d'exploitation lors de la phase de développement.

Le HAL pour les systèmes embarqués commerciaux se limite donc le plus souvent au BSP, pour *board support package*. En fait, le BSP représente la partie du HAL qui traite de tous les composants matériels, hormis le processeur. De plus, chaque système d'exploitation possède son propre HAL. [Windows Ce] propose plusieurs BSP pour différentes cartes de développement standard. Les BSP consistent en chargeur de boot (boot loader), une couche d'abstraction de OEM, des pilotes de périphériques (*device drivers*) et des fichiers de configuration. La couche d'abstraction d'OEM peut être configurée pour s'adapter à une architecture donnée. [eCos] propose un ensemble de primitives de l'API HAL bien défini, par contre la différence entre le HAL et les pilotes de périphériques n'est pas claire. [RT Linux] défini une API HAL temps réel appelée RTHAL pour abstraire les mécanismes d'interruption de [Linux] : activer/désactiver les interruptions et retourner du traitement d'une interruption.

[a386] est un exemple de HAL qui ne dépend pas d'un système d'exploitation. Il dépendait de l'architecture du processeur i386, puis il a été porté aux processeurs ARM.

II.3.2 La couche système d'exploitation

II.3.2.1 Présentation de la couche système d'exploitation

Dans ce paragraphe nous nous limiterons à la présentation d'un système d'exploitation en tant qu'une abstraction du matériel, qui gère des ressources et qui offre des services. Une description plus détaillée peut être trouvée dans [Tanenbaum 94].

a. Un système d'exploitation en tant qu'abstraction du matériel

Le matériel idéal et le matériel réel

Pour un programme, le matériel idéal aurait des ressources infinies (mémoire, calcul, etc...) et immédiatement disponibles. Tous ces composants disposeraient aussi de la même interface simple. Le matériel réel n'a hélas pas ces propriétés : aucun composant matériel ne peut réagir instantanément. Les ressources matérielles étant onéreuses, il est souvent nécessaire de les partager entre plusieurs tâches logicielles (c'est notamment le cas pour le processeur et la mémoire). Enfin les différents composants matériels peuvent proposer une très grande variété d'interfaces (pour des raisons de performance, ou tout simplement du fait de leurs fonctionnalités).

Solution apportée par le système d'exploitation

Un système d'exploitation peut être vu comme un matériel abstrait idéal. Il s'agit en fait d'une couche logicielle qui encapsule le matériel et dont le but est de simplifier la conception des applications logicielles. Il peut alors se placer comme unique interlocuteur avec les programmes car il présente les caractéristiques suivantes :

- Il peut cacher aux tâches logicielles les indisponibilités du matériel.
- Il peut fournir une interface simple identique quel que soit le composant matériel (par exemple [UNIX] encapsule les accès matériels dans des accès fichiers).

b. Un système d'exploitation en tant que gestionnaire de ressources

Dans le cas où le logiciel est composé de tâches concurrentes, il est nécessaire de pouvoir gérer les ressources partagées entre ces tâches. Le système d'exploitation peut être considéré comme ce gestionnaire de ressources matérielles pour l'application. Il peut gérer la ressource processeur grâce aux algorithmes d'ordonnancement de tâches, la ressource mémoire grâce aux fonctions d'allocation et de libération de la mémoire, ainsi que toutes les autres ressources grâce aux gestionnaires de périphériques.

Le système d'exploitation, associé au matériel est alors considéré comme un serveur de ressources. Tandis que les tâches logicielles sont considérées comme des clients.

c. Services rendus par un système d'exploitation

Un système d'exploitation propose des services aux tâches dont il permet l'exécution. Ces services peuvent être par exemple :

- La réalisation de communications entre tâches sur un même processeur ou sur plusieurs processeurs.
- La gestion de périphériques.
- La possibilité d'endormir ou de réveiller les tâches suivant certaines conditions.

Le mécanisme utilisé pour permettre aux tâches d'accéder à ces services est celui des appels système, réalisés en utilisant les primitives de l'API du système d'exploitation. Lorsqu'une tâche utilise un de ces appels, une interruption logicielle est générée. Une telle interruption fait passer le processeur en mode système pour l'exécution de la fonction réalisant ce service. Une fois le service est rendu, la tâche peut reprendre la main et en utiliser le résultat : par exemple traiter la donnée qu'elle avait demandée de récupérer.

II.3.2.2 Différences avec les systèmes d'exploitation pour architectures multiprocesseurs classiques

Les systèmes d'exploitation embarqués possèdent de nombreuses fonctionnalités communes avec les systèmes d'exploitation généraux. Ils doivent par exemple pouvoir gérer une ou plusieurs tâches et des ressources matérielles.

Ces fonctionnalités sont cependant à moduler suivant les besoins spécifiques d'un système embarqué : par exemple une gestion multitâche n'est pas nécessaire si une seule tâche est exécutée par processeur. De plus, elles doivent respecter des contraintes particulières aux systèmes embarqués, tel que des délais ou périodes d'exécution fixes, qui peuvent notablement changer leurs implémentations.

a. Fonctionnalités de communication spécifiques aux systèmes d'exploitation embarqués

Dans un système embarqué spécifique, et notamment dans un système monopuce, l'architecture est dédiée à l'application pour optimiser les performances et le coût. Cela implique que les architectures des MPSoC sont très variées. Cette variété se répercute directement sur les communications : tout d'abord parce qu'elles aussi sont optimisées pour l'application, mais aussi parce que les divers composants n'utilisent que rarement les mêmes types de communications.

Ainsi les communications peuvent être point à point ou multipoints comme le montre la figure II-7. Cette figure présente les deux types de communications, le premier requiert plus de connexions et donc plus de surface, tandis que le deuxième peut être un goulet d'étranglement et donc un facteur ralentissant. Elles peuvent être implémentées avec ou sans mémorisation intermédiaire. La mémorisation intermédiaire permet de désynchroniser deux sous-systèmes, sans pour autant les forcer à s'attendre mutuellement pour échanger des données. Cette mémorisation intermédiaire peut elle-même être gérée de plusieurs manières différentes : par exemple cela peut

être un système de mémoire partagée, ou cela peut être un système FIFO comme illustré dans la figure II-8.

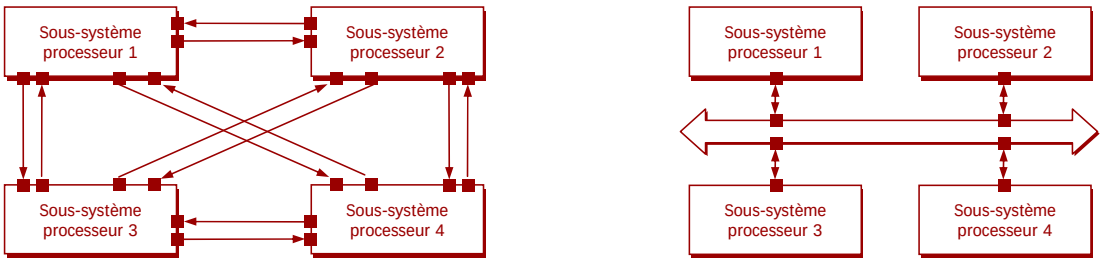


Figure II-7 : Communication point à point et communication multipoints

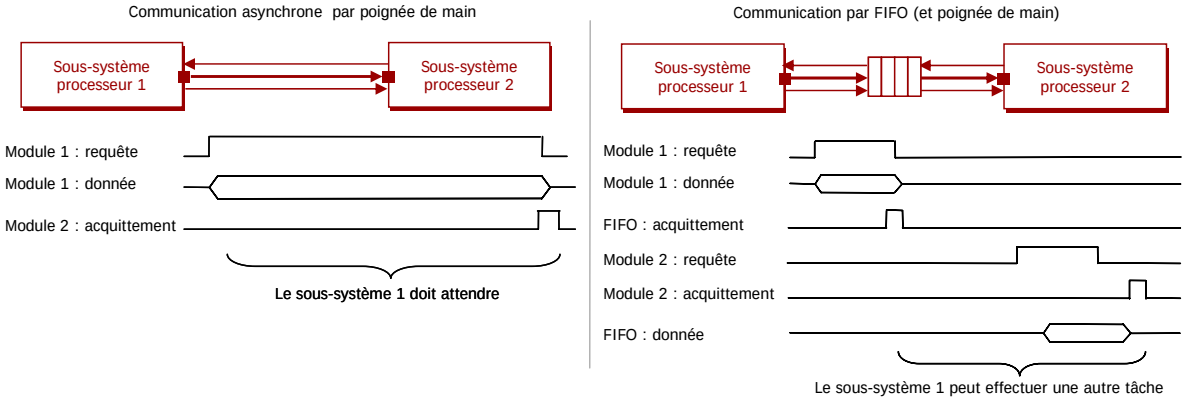


Figure II-8 : File d'attente FIFO pour désynchroniser deux blocs

Une caractéristique importante des communications est la définition des protocoles : ils sont très nombreux suivant les architectures, les données à transiter et les contraintes associées (par exemple : [CAN], [AMBA]). Ces communications peuvent être réalisées en faisant plus ou moins intervenir le logiciel ou le matériel, suivant les compromis choisis entre la performance et la souplesse. De plus, au cours de la conception, ou même après la réalisation, la frontière entre le logiciel et le matériel n'est pas fixe.

Ces divers cas se retrouvent souvent combinés dans la même architecture.

b. Fonctionnalités temporelles

Tout comme les systèmes d'exploitation multi utilisateurs tel que [UNIX], les systèmes embarqués ont des contraintes temporelles fortes. Cependant, ces contraintes n'ont pas la même nature [Dorseuil 91] :

- Pour les systèmes multi utilisateurs il est important de ne pas bloquer longtemps une tâche. Le modèle temporel employé est celui du temps partagé, le but étant d'assurer une certaine équité entre les tâches.

- Pour les systèmes d'exploitation embarqués, il est important de respecter des délais, même s'il est nécessaire de bloquer des tâches pendant une longue durée. Le modèle temporel employé est le modèle temps réel, (mou ou dur suivant les cas).

C. Gestion des utilisateurs

En général, les systèmes embarqués n'ont pas à gérer plusieurs utilisateurs en même temps. Il n'est donc plus nécessaire de gérer la sécurité ni le temps partagé entre les utilisateurs. Il en résulte une gestion plus simple des utilisateurs au niveau des systèmes d'exploitation pour MPSoC.

II.3.2.3 Désidérata pour les systèmes d'exploitation des MPSoC dédiés

A fur et à mesure de la progression de notre étude, nous avons pu établir trois critères qui sont à la base d'un système d'exploitation adéquat à une architecture MPSoC. Ces critères sont relatifs aux systèmes d'exploitation ainsi qu'à leurs méthodologies de conception. Nous présenterons dans la suite ces critères.

Adaptation du système d'exploitation au matériel et à l'application, systématisation

Pour être utilisé dans un contexte dédié, le système d'exploitation doit impérativement être adapté au couple matériel – application. Il doit gérer de la façon la plus optimale le matériel disponible (utiliser des instructions spécifiques, respecter les contraintes physiques…) et offrir les services nécessaires et suffisants au fonctionnement de l'application (un service manquant causera un disfonctionnement de l'application, un service en trop représente un gaspillage en terme de surface sur la puce).

Modularité du système d'exploitation et de sa méthodologie de conception

La modularité présente plusieurs avantages :

- Simplifie la conception : en décomposant le système en un ensemble de modules, on décompose aussi sa complexité et on simplifie les problèmes posés lors de sa conception en totalité (principe de diviser pour régner).
- Elle permet la réutilisation. C'est un facteur réducteur du temps de mise sur le marché. Quand on réutilise un processeur, on peut envisager de réutiliser le même système d'exploitation qui lui a déjà été associé moyennant le changement de quelques paramètres.
- Outre ces avantages, la modularité d'un système d'exploitation permet une meilleure maintenance et une adaptation plus aisée. Ainsi, pour ajouter une nouvelle fonctionnalité, il faudra seulement changer un module ou ajouter un autre sans avoir à revoir le système d'exploitation en intégralité.

[Gauthier 01] et [Lyonnard 03] proposent une méthodologie basée sur l'utilisation d'une bibliothèque de composants matériels (processeurs, DSP, IP, architectures de communication…) et d'une autre contenant des composants de système d'exploitation (services élémentaires,

protocoles, ordonnanceur). L'idéal serait alors de disposer d'un outil ou d'une méthodologie de conception qui permet, en premier temps, de créer un ou plusieurs systèmes d'exploitation pour exploiter une architecture multiprocesseur monopuce. Puis, dans un deuxième temps, d'adapter automatiquement ces systèmes d'exploitation aux besoins spécifiques de l'application à laquelle toute l'architecture est dédiée.

Optimisation du système d'exploitation au matériel et à l'application

Après avoir étudié l'adaptation du système d'exploitation à la fois au matériel et au logiciel, on passe par une phase d'optimisation qui consiste à choisir la meilleure adaptation. L'optimisation a pour but de limiter les besoins du système d'exploitation en énergie (ex : utiliser une politique d'ordonnancement qui diminue le nombre de changements de contexte) et en surface (ex : diminuer sa taille) tout en gardant des performances optimales.

Cette optimisation peut conduire à effectuer des modifications des protocoles de communications et à une amélioration des algorithmes... Elle donne lieu à un compromis entre l'adaptation, l'optimisation et les performances qui pourraient amener à revoir le partitionnement logiciel matériel.

II.3.2.4 Avantages et inconvénients de l'utilisation des systèmes d'exploitation pour les MPSoC

Dans cette section nous allons discuter des avantages et inconvénients de l'utilisation d'un système d'exploitation pour les systèmes MPSoC embarqués.

Avantages de l'utilisation des systèmes d'exploitation pour les MPSoC

- Programmation simplifiée des applications

Le système d'exploitation gère lui-même le matériel et propose aux applications des fonctions d'ordonnancement des tâches, de gestion du multitâche, de protection mémoire... Cet avantage serait encore plus important si tous les systèmes d'exploitation offraient une même interface. Ce n'est malheureusement pas le cas : les impératifs de performance empêchent souvent l'utilisation d'interfaces génériques abstraites et la multitude des systèmes d'exploitation et des architectures sont des freins à l'uniformité des interfaces.

- Utilisation des spécificités des processeurs :

Les systèmes d'exploitation, spécialement programmés pour le processeur sur lequel ils vont s'exécuter, peuvent tirer partie de ses spécificités :

Le mécanisme d'interruption permet d'interrompre le fonctionnement séquentiel du programme suite à un évènement extérieur. Ces interruptions ne sont en général pas prises en compte dans les modèles logiciels, de plus elles sont très variables d'un processeur à un autre. Un système d'exploitation est capable de les gérer.

Des instructions de réduction de consommation sont proposées par de nombreux processeurs pour systèmes embarqués. Il y a par exemple des fonctions de mise en veille du processeur jusqu'au prochain événement (manifesté par une interruption).

Des instructions de synchronisation ou d'exclusion mutuelle (par exemple l'instruction Test And Set) servent pour l'utilisation de mémoires partagées entre plusieurs processeurs.

Des instructions permettent de contrôler le fonctionnement du processeur, comme les instructions de gestion de cache (et qui permettent des optimisations de performances ou de consommation).

Inconvénients de l'utilisation des systèmes d'exploitation pour les MPSoC

- Les systèmes d'exploitation consomment de la mémoire :

Au niveau de certains systèmes d'exploitation commerciaux, la modularité est absente. En effet, étant prévus pour supporter plusieurs applications, ils proposent des services suffisamment généraux pour être utilisables par toutes les applications. La généralité du système d'exploitation vis à vis de l'application fait qu'il est souvent plus volumineux que nécessaire. C'est un défaut important dans le monde des systèmes embarqués où la mémoire est limitée.

Les systèmes d'exploitation modulaires tentent de résoudre ce problème en mettant leurs fonctionnalités sous la forme de modules optionnels, qui ne seront effectivement inclus dans le système d'exploitation que s'ils sont utilisés [QNX], [VxWorks].

Cependant, ces modules restent eux-mêmes généraux, à moins d'avoir une bibliothèque de modules contenant tous les types de modules spécifiques possibles, ce qui n'est guère réaliste.

- Les systèmes d'exploitation consomment des ressources processeur :

La majorité des systèmes d'exploitation existant sont très généraux pour les applications qu'ils doivent exécuter. Cette généralité se paye en terme de mémoire consommée, mais elle peut se payer aussi en terme de vitesse d'exécution : par exemple les synchronisations utiliseront toujours des mécanismes de sémaphores complets, alors que dans de nombreux cas un simple verrou suffit.

La vitesse du système d'exploitation est aussi limitée par l'ordonnancement dynamique des tâches qui demande du temps aussi bien pour la décision que pour le passage d'une tâche à l'autre.

- Les systèmes d'exploitation peuvent être non déterministes :

Dans les systèmes embarqués, des contraintes temps réel imposent que le fonctionnement soit déterministe. Ce déterminisme n'est pas toujours aisé à obtenir avec les systèmes d'exploitation qui sont des programmes à exécution complexe. En fait, il est souvent impossible de savoir avant utilisation si une application basée sur un système d'exploitation va respecter les délais imposés ou non.

II.3.2.5 Exemples de systèmes d'exploitation pour MPSoC

Il existe de nombreux systèmes d'exploitation pour architectures MPSoC dans le commerce. Le marché est très ouvert et il n'y a pas de meneur à cause de la diversité des applications utilisant les MPSoC [TRON 00] [DSP Consulting 99]. Les systèmes d'exploitation sont le plus souvent du type temps réel (RTOS). Les plus répandus sont [VxWorks], [QNX], [OSE], [WinCe], [pSOS], [eCos], [Chorus OS], [NucleusPlus]. Une étude des solutions systèmes d'exploitation commerciales sera présentée dans le paragraphe III.4.

II.3.3 **Couche Middleware de programmation parallèle**

II.3.3.1 Présentation et rôle de la couche middleware de programmation parallèle

Le terme « middleware », aussi appelé intergiciel ou logiciel médian, est utilisé pour désigner une classe de logiciels qui assurent le rôle d'un intermédiaire entre le code de l'application et le transport de données par le réseau de communication.

L'utilisation d'un middleware est d'une grande aide pour la migration de l'application d'un système à architecture monolithique vers un système à architecture de type client/serveur, en fournissant une communication à travers des plateformes hétérogènes. Dans ce sens, le middleware s'étend aux systèmes de gestion de bases de données, serveurs web, serveurs d'applications…

Dans ce travail, nous considérons la couche middleware de programmation parallèle comme un bus logiciel qui transporte en logiciel les données (par analogie au bus matériel). Elle permet à plusieurs tâches de s'exécuter sur un ou plusieurs sous-systèmes processeurs et d'interagir via un réseau de communication. Elle fournit aussi des abstractions et des services de haut niveau pour faciliter la programmation d'applications à fort parallèlisme et des tâches de gestion du système.

Dans les systèmes MPSoC, la couche middleware de programmation parallèle est utilisée, au dessus du système d'exploitation pour supporter des applications à fort parallèlisme. Cette couche se base sur les services du système d'exploitation. Elle offre à l'application, d'une façon simplifiée, des services de communication et de synchronisation distribués plus complexes que ceux proposés par le système d'exploitation.

II.3.3.2 Différences avec les middlewares de programmation parallèle des architectures multiprocesseurs classiques

Un exemple classique d'utilisation d'un middleware dans une architecture multiprocesseur classique est sa médiation (i.e. faire office d'intermédiaire) entre un client et un serveur d'une base de données transactionnelle. L'utilisation du middleware pour ce type d'applications permet de mieux servir les requêtes des clients - tout en réduisant le nombre de connexions à la base de données - et de retourner plus efficacement les données requises.

Les middlewares utilisés pour les systèmes multiprocesseurs classiques ne sont pas adéquats pour les systèmes MPSoC, car ils ne supportent pas leurs contraintes. Dans un contexte multiprocesseur classique, un middleware est conçu pour supporter un maximum d'applications. Il implémente toujours tous les services qu'il propose même si ils ne sont pas requis par l'application (par exemple [MPICH] implémente les 123 primitives constituant l'API de la spécification [MPI]). Cette approche ne peut être adoptée dans un contexte embarqué car les ressources sont limitées et onéreuses.

La couche middleware de programmation parallèle de l'adaptation logicielle d'un système MPSoC doit être optimisée par rapport à la partie matérielle à cause des contraintes spécifiques aux MPSoC (consommation d'énergie, coût en surface,...). Plus précisément, un middleware de programmation parallèle idéal pour MPSoC implémente uniquement les services requis par l'application.

II.3.3.3 Exemples de middleware de programmation parallèle

Dans les systèmes embarqués, les middlewares sont fournis comme :

- Un composant optionnel en complément du système d'exploitation, comme [OSE gateway] de [OSE] ou [Qnet] de [QnX] qui permettent une communication inter sous-systèmes. Dans ce contexte ils ne peuvent être utilisés qu'au dessus de ce système d'exploitation.
- Un composant indépendant du système d'exploitation, comme [eORB] de [PrismTech] ou [VisualityNQ] de [Visuality Systems]. [eORB], qui est une implémentation optimisée pour un contexte embarqué du middleware [CORBA], peut être utilisée au dessus de différents systèmes d'exploitation comme par exemple [VxWorks] ou [LynxOS].

II.4 Conclusion

Les systèmes MPSoC sont composés par une partie logicielle, une partie matérielle et une interface logicielle/matérielle. Cette interface logicielle/matérielle est elle-même composée d'une adaptation logicielle et d'une adaptation matérielle. Maîtriser le processus de conception de l'interface est la clé de voûte pour faire face à la complexité accrue de la conception des systèmes MPSoC.

Ce chapitre a présenté l'architecture des interfaces logicielles/matérielles. Les différentes couches conceptuelles de l'adaptation logicielle ont été détaillées. Nous verrons dans le chapitre suivant que pour des raisons de performance ces couches serons fusionnées pour former ce que nous appellerons le logiciel dépendent du matériel, dis HDS (pour *hardware dependent software*).

Chapitre III

Approches pour l'implémentation de l'adaptation logicielle de l'interface logicielle/matérielle

III.1 Introduction

Nous avons étudié dans le chapitre précédent, l'architecture des interfaces logicielles/matérielles. Nous avons présenté une approche conceptuelle de l'adaptation logicielle basée sur une décomposition en couche.

Nous discutons dans ce chapitre des approches pour l'implémentation de l'adaptation logicielle. Dans le premier paragraphe, nous présentons les approches classiques pour l'implémentation de l'adaptation logicielle dans les systèmes MPSoC. Puis, dans le second paragraphe, nous analysons la couche système d'exploitation (principale couche de l'adaptation logicielle où les tâches de l'application sont gérées (ordonnancement, affectation des priorités, …)). Nous étudions alors les différentes architectures des systèmes d'exploitation pour les systèmes embarqués puis les solutions système d'exploitation proposées dans le commerce ou issues du domaine de la recherche. Ensuite, nous étudions les approches pour l'implémentation de l'adaptation logicielle. Nous présentons dans le dernier paragraphe l'approche adoptée au sein du groupe SLS pour la conception des interfaces logicielles/matérielles.

III.2 Approches pour l'implémentation de l'adaptation logicielle pour les systèmes MPSOC

Nous avons identifié deux approches pour l'implémentation de l'adaptation logicielle pour les systèmes MPSoC. Il s'agit de l'approche matérielle et de l'approche logicielle. Nous présentons dans ce paragraphe chacune de ces approches. Puis nous discutons de leurs différences avec les approches pour systèmes multiprocesseurs classiques.

III.2.1 **Approche matérielle**

L'approche matérielle, utilisée par les manufacturiers des circuits matériels, est basée sur l'adoption des plateformes matérielles. Une plateforme matérielle est une architecture qui peut être à la base de plusieurs produits dans un même domaine d'application, par exemple les plateformes à base de [OMAP] et [Nomadik] pour les terminaux mobiles ou à base de [Nexperia] pour la télévision numérique. Dans cette approche, la conception des interfaces logicielles/matérielles se base sur des kits de développement de logiciel (appelés SDK pour *software development kit*) fournis par le concepteur de la plateforme matérielle.

Par exemple, les kits SDK proposés par Philips pour la plateforme [Nexperia] détaillent la structure logicielle à la base de cette plateforme, dont un ensemble d'interfaces de programmation d'applications pour les systèmes d'exploitation et la connectivité, et un autre ensemble pour la diffusion audio et vidéo en continu. Les kits SDK contiennent, entre autres, des primitives de l'API documentées et un système d'exploitation [Nexperia 8550].

Le logiciel applicatif développé avec cette approche est spécifique à la plateforme matérielle et doit être revu pour être réutilisé sur une nouvelle plateforme (réécrit suivant le SDK de la nouvelle plateforme).

III.2.2 Approche logicielle

L'approche logicielle est proposée par les concepteurs de logiciels embarqués. Elle consiste à utiliser des solutions logicielles industrielles pour réaliser l'interface logicielle/matérielle. Il s'agit de développer le logiciel applicatif par rapport à un système d'exploitation commercial, par exemple [WinCe] ou [SymbianOS] dans le domaine des terminaux mobiles. Le critère de sélection du système d'exploitation est son support de l'architecture matérielle ciblée.

Le logiciel applicatif développé par cette approche est dépendant du système d'exploitation choisi, il ne peut être porté que sur des architectures matérielles supportées par le système d'exploitation choisi. L'adoption d'une nouvelle architecture matérielle peut conduire au changement du système d'exploitation et à une nouvelle conception du logiciel applicatif (pas de réutilisation). De plus, les systèmes d'exploitation, étant conçus pour être utilisés avec un grand nombre d'applications, ils offrent des fonctionnalités qui ne seront pas utilisées pour une application donnée : c'est un surcoût en terme de ressources nécessaires au système.

III.2.3 Différence avec les systèmes multiprocesseurs classiques

La vue conceptuelle de l'adaptation logicielle utilisée pour les systèmes multiprocesseurs classiques est semblable à celle proposée en II.3 reprise dans la figure III-1 – (a) (une approche en couche, comme par exemple les sept couches OSI du protocole [TCP/IP]). Cependant, l'implémentation de cette vue conceptuelle de l'adaptation logicielle est différente entre les deux systèmes :

- Pour les systèmes multiprocesseurs classiques, les couches sont aussi présentes à l'implémentation. Chaque couche fournit des services exclusivement à celle de dessus et utilise exclusivement les services de celle de dessous. Ceci est dû au fait que la portabilité et la réutilisation des couches sont les critères les plus importants lors de la conception du logiciel des systèmes multiprocesseurs classiques. De plus, toutes les primitives offertes par l'API d'une couche y sont implémentées, même si elles ne sont pas utilisées par les autres couches dans le cadre d'une application donnée.
- Dans le cas des systèmes MPSoC dédiées à des applications spécifiques, les ressources matérielles sont réduites et onéreuses. Les différentes couches de l'adaptation logicielle doivent être taillées suivant les besoins de l'application et les caractéristiques de l'architecture matérielle. Seules les primitives de l'API utilisées y sont implémentées. Aussi, l'adaptation logicielle pour les systèmes MPSoC se distingue de celles des architectures multiprocesseurs classiques par une optimisation inter couches. Une couche peut utiliser les services d'une couche qui n'est lui est pas directement adjacente.

Les deux approches (matérielle et logicielle) adoptent une conception basée principalement sur l'utilisation des systèmes d'exploitation. Ainsi, la possibilité d'optimiser l'adaptation logicielle aux besoins de l'application et aux caractéristiques de l'architecture est fortement dépendante des possibilités offertes par le système d'exploitation utilisé.

L'adaptation logicielle est donc dépendante du matériel dans le cas de la conception des systèmes MPSoC, elle est alors appelée HdS pour *hardware dependant software* (figure III-1 – (b)). L'API du HdS est formée par l'ensemble des primitives des API PPM, système d'exploitation et HAL utilisées par l'application.

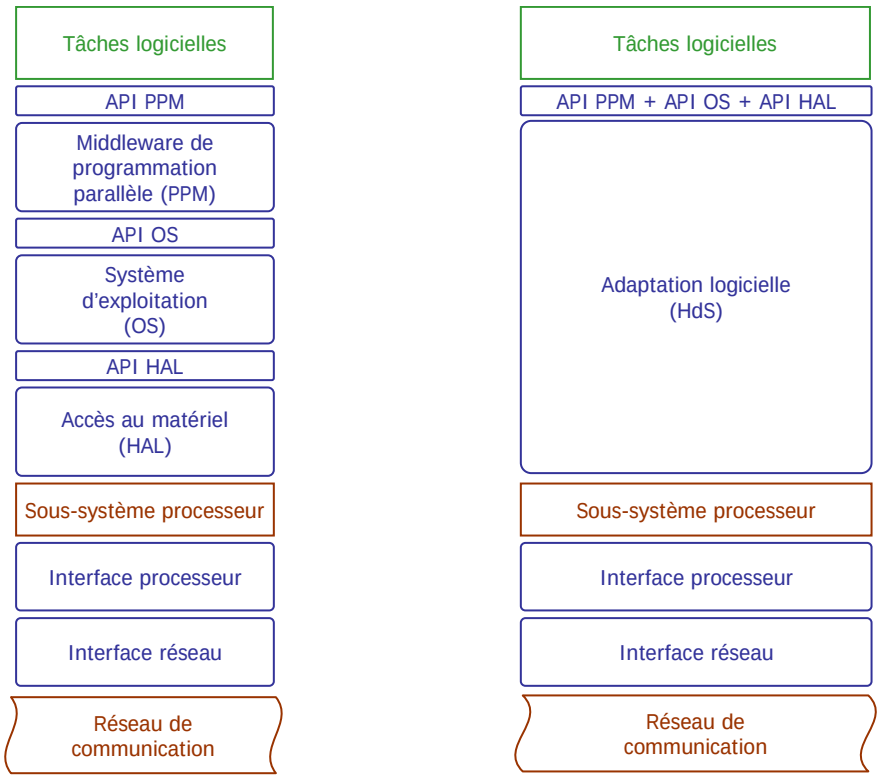


Figure III-1 : Vue conceptuelle et vue de l'implémentation de l'adaptation logicielle

III.3 Architectures des systèmes d'exploitation pour les systèmes embarqués multiprocesseurs

III.3.1 **Architecture monolithique**

Le terme monolithique est utilisé dans la littérature pour qualifier le noyau d'un système d'exploitation utilisé dans un environnement distribué. Un tel noyau fournit la plupart des services (gestion des fichiers, des processus...) et peux gérer ainsi une architecture multiprocesseur.

Dans ce mémoire, on utilisera le terme monolithique pour qualifier le système d'exploitation en totalité. Un système d'exploitation monolithique est un système d'exploitation qui gère à lui seul tous les processeurs de l'architecture. Le système d'exploitation peut résider sur un seul processeur ou être distribué sur plusieurs unités de traitement (figure III-2). Dans ce dernier cas, chaque unité de traitement assure alors une partie des fonctionnalités. Le fonctionnement distribué de tous les processeurs en même temps réalise la fonctionnalité globale de ce système d'exploitation.

La principale caractéristique des systèmes d'exploitation monolithiques est la gestion centralisée des tâches. Un ordonnanceur central, affecte les tâches aux processeurs. Au niveau de ces unités de traitement, il y a des files d'attente permettant de désynchroniser les processeurs entre eux.

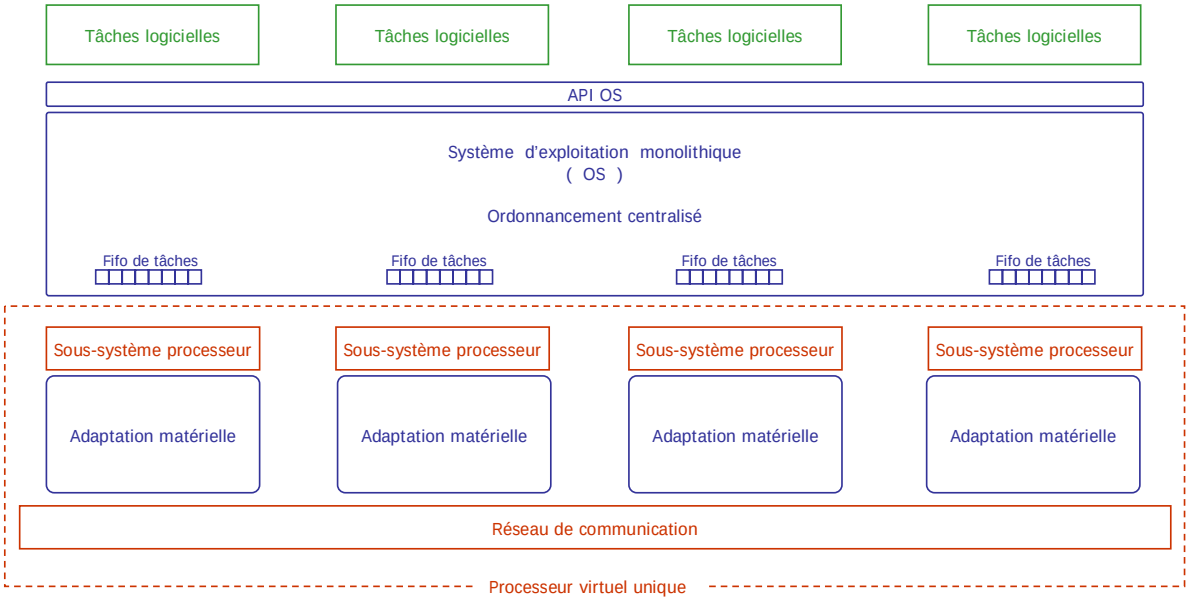


Figure III-2 : Système d'exploitation monolithique

Évaluation du type système d'exploitation monolithique

- Ce type de système d'exploitation encapsule les différentes unités de traitements et les présente comme un seul et unique processeur (notion de processeur virtuel unique). La programmation de l'application est ainsi simplifiée.
- Un système d'exploitation monolithique est caractérisé par un ordonnancement centralisé des tâches. Cet ordonnancement offre la possibilité de la répartition dynamique des tâches sur les processeurs pour en assurer une exploitation optimale en charge. Cependant, l'exploitation de cet avantage (i.e. l'ordonnancement dynamique centralisé) dans un système MPSoC est très discutable :
  - Si on considère qu'un MPSoC contient des processeurs hétérogènes destinés à des calculs spécifiques, l'apport d'un ordonnancement dynamique centralisé n'est pas significatif : les tâches sont destinées – dés le début de la conception – à des processeurs spécifiques. Il est alors contradictoire d'envisager leurs migrations à un autre processeur libre non spécifique. Un ordonnanceur statique est plus simple et mieux adapté à ce type d'architectures.

- Mais, si on considère que les processeurs embarqués sont homogènes (approche existante mais moins utilisée, par exemple l'*Emotion Engine* de la PlayStation2 ou le Cell de la PlayStation3 renferment plusieurs processeurs homogènes) l'avantage de l'utilisation d'un ordonnancement centralisé dynamique est évident : il permet une meilleure gestion de la charge des processeurs et donc une exploitation optimale des ressources.

- Un système d'exploitation monolithique est adapté à une architecture particulière et ne peut être utilisé ou porté facilement à une nouvelle architecture. On ne peut alors bénéficier facilement d'une réutilisation éventuelle d'une partie du système.

- Pour implémenter un système d'exploitation monolithique, on utilise le plus souvent une machine virtuelle (une sorte de super HAL) au dessus des processeurs pour (1) pouvoir exécuter le logiciel dans un seul langage d'assemblage et (2) donner l'impression de l'existence d'un seul et unique processeur. C'est une simplification très importante de la complexité du codage de l'application (programmer un seul processeur avec un seul langage). Le concepteur de l'application verra donc sa tâche diminuée en complexité. Cependant, le développeur du système d'exploitation aura à gérer une complexité accrue, d'autant plus s'il doit gérer des processeurs hétérogènes.

La solution monolithique offre la possibilité de programmer les applications d'une façon simplifiée. Elle est à envisager si les processeurs sont homogènes. Cependant, dans le cas le plus fréquent, les processeurs sont hétérogènes car dédiés à des besoins spécifiques. Le système d'exploitation monolithique s'avère alors très compliqué à développer. De plus il ne tirera certainement pas tout le profit de la spécificité des processeurs : optimiser le système d'exploitation à la fois pour tous les processeurs implique une grande augmentation en taille, qui n'est pas tolérée dans un contexte embarqué. Il en résulte une sous exploitation des ressources et une dégradation des performances.

Exemples de systèmes d'exploitation monolithiques

[OS/360], [VMS], [UNIX] : sont prévus pour des architectures multiprocesseurs homogènes. Ils permettent une utilisation efficace du parallèlisme en distribuant les tâches complètes sur les différents processeurs.

[VSPWorks] et [Virtuoso] utilisent le principe du processeur virtuel unique (VSP, pour virtual single processor). Ils permettent à l'application de voir une architecture multiprocesseur, composée de processeurs et de processeur de traitement de signal (DSP), comme un processeur virtuel unique.

III.3.2 Architecture en réseau de système d'exploitation

La deuxième façon d'exploiter les ressources matérielles d'une architecture multiprocesseur consiste à exécuter sur chaque processeur un système d'exploitation qui lui est propre. Le fonctionnement distribué et parallèle des systèmes d'exploitation sur ces processeurs formera alors

un réseau de systèmes d'exploitation. Les différents systèmes d'exploitation collaborent pour assurer le fonctionnement global du système (figure III-3).

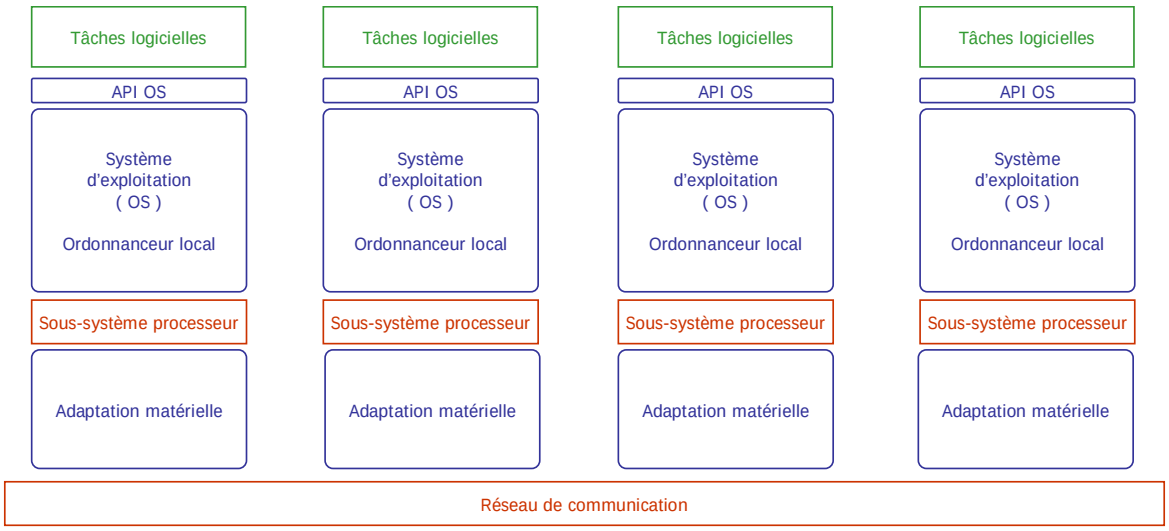


Figure III-3 : Réseau de systèmes d'exploitation

Évaluation du type réseau de système d'exploitation

La deuxième solution consiste à utiliser un système d'exploitation par processeur, il est évident que cette solution est mieux adaptée pour être utilisée avec des processeurs hétérogènes qu'avec des processeurs homogènes. Cette organisation permet de remédier aux défauts de l'approche monolithique (manque de modularité, sous exploitation des ressources hétérogènes ...) mais elle ne garde pas ses avantages (programmation simplifiée...).

De plus, cette seconde architecture apporte des contraintes supplémentaires. Tout d'abord les contraintes en surface deviennent plus fortes, car plusieurs systèmes d'exploitation sont susceptibles de consommer plus de mémoire qu'un seul. Ensuite il est assez difficile de maintenir une cohérence globale avec plusieurs systèmes sans engendrer une surcharge de messages de contrôle. Enfin, avec une telle architecture, les tâches ne peuvent pas passer facilement d'un processeur à un autre : il faut décider avant la compilation de la conception à quel processeur chacune est allouée (pas de migration dynamique des tâches).

Toutefois, la solution réseau de système d'exploitation présente deux avantages importants : une meilleure modularité permettant la réutilisation des systèmes d'exploitation (le système d'exploitation étant relatif à un sous-système processeur et non plus à l'architecture entière) et une meilleure optimisation des performances puisque chaque système d'exploitation tire le maximum des ressources de son processeur.

Exemples de système d'exploitation à architecture en réseau système d'exploitation

[QNX] : Utilise l'approche d'implantant des micros noyaux de 10 KB entourés par un ensemble de processus optionnels qui fournissent des services d'un niveau plus haut.

[Mach] : Originaire de l'université de Carnegie Mellon, devient à la base de plusieurs systèmes de recherches.

III.4 Analyse des systèmes d'exploitation existants

Pour être utilisé dans un système MPSoC, un système d'exploitation commercial ou industriel doit avoir une capacité à être adapté à la fois à l'architecture (support des multiples processeurs cibles, taille réduite) et à l'application (sélection et configuration des services). Aussi, des outils d'aide à la conception et d'automatisation sont requis pour réduire le cycle de conception.

Dans la suite, nous présentons une analyse des systèmes d'exploitation industriels et commerciaux et des solutions issues du domaine de la recherche (notamment des systèmes d'exploitation matériels) par rapport aux critères définis dans II.3.2.3.

III.4.1 Adaptation à l'architecture matérielle

Support du processeur cible

Deux solutions prévalent pour supporter les processeurs cibles :

- Soit le système est fourni sans les sources, auquel cas il est proposé une version du système par processeur cible. C'est le cas de [VxWorks], [Virtuoso], [C EXECUTIVE], [Chorus OS], [QNX], [EYRX]. Cette situation ne favorise pas une exploration avec des processeurs différents, de plus il est impossible d'utiliser le système avec un processeur qui n'a pas été prévu par le fournisseur.
- Soit le système est fourni avec les sources, auquel cas le changement de processeur revient à changer quelques fichiers spécifiques et à les recompiler, il n'est donc pas nécessaire d'avoir plusieurs versions du système d'exploitation. C'est le cas pour [LynxOS], [eCos], [NucleusPLUS], [RTXC], [MicroC/OS-II]. En outre, dans le cas où le système n'a pas été prévu pour un processeur donné, il est toujours possible de l'adapter (avec plus ou moins de facilité suivant la portabilité du système, la qualité de ses sources et l'utilisation ou non d'une couche HAL).

La deuxième solution est plus flexible que la première, mais elle peut être plus difficile à supporter par l'éditeur du système d'exploitation car l'architecture cible est moins figée.

Pour les autres parties de l'architecture, il est fourni en général des règles pour écrire la couche intermédiaire permettant d'adapter le système d'exploitation. Cette couche est appelée BSP (Board Support Package pour [WinCe]). Il est aussi habituel de permettre l'ajout de pilotes de périphériques au système pour pouvoir gérer des types de périphériques non pris en compte.

Adaptation à la taille de la mémoire disponible

Pour s'adapter aux ressources de l'architecture en terme de taille mémoire disponible, les éditeurs de système d'exploitation proposent des versions de tailles différentes de leurs solutions. [WindRiver System] propose [VxWorks 5.0], [VxWorks AE] et [VxWorks 6.0] pour une taille allant de 80 ko à 500 ko. [Enea Data] propose [epsilon], [OSEck] et [OSE] pour une taille mémoire

respectivement de l'ordre de 4 ko, 8ko et inférieure à 100 ko. [Sun] propose le système d'exploitation [ChorusOS] décliné dans les versions [micro], [CalssiX] et [JaZZ].

La différence de la taille mémoire de ces versions est due à la différence en nombre de services supportés. La taille extrêmement réduite de la version [epsilon] de [OSE] est elle due a l'utilisation directe d'un langage assembleur pour l'écriture du système d'exploitation.

Les différentes versions du système d'exploitation sont destinées à des domaines d'applications différents. Elles représentent une première adaptation (à très gros grain) du système d'exploitation aux besoins de l'application.

Support des architectures multiprocesseurs

Certains systèmes d'exploitation supportent les architectures multiprocesseurs d'une façon native. Tel est le cas de [QNX] qui tire profit de son architecture client serveur basée sur une communication par échange de messages. Sa bibliothèque [Qnet] étend le mécanisme d'échange de messages au delà d'un nœud et fournit une transparence du réseau, le système peut alors être distribué sur plusieurs nœuds sans avoir besoin de changer le code de l'application.

[OSE] permet aussi, grâce au « Link Handler » et « Name Server » une connexion transparente des processeurs aux services, quelque soit leur localisation physique. [OSE] permet d'envoyer un message de la même façon quelque soit la localisation du processus destinataire (sur un même ou sur un processeur différent). A l'utilisation, OSE ne fait pas de distinction entre une architecture mono ou multiprocesseur.

De plus l'éditeur de [OSE], propose la solution [OSE gateway] qui permet à un réseau de systèmes d'exploitation hétérogènes de communiquer entre eux, à condition d'avoir une version OSE qui tourne sur l'un d'eux.

[Virtuoso] puis [VSPWorks] utilisent le principe du processeur virtuel unique, qui permet de voir un ensemble de processeurs (et/ou DSP) comme un processeur unique virtuel. Il s'agit de deux systèmes d'exploitation de type monolithique, où la communication entre des tâches sur des processeurs différents ainsi que leur migration sont gérées d'une façon transparente pour l'application.

D'autres systèmes d'exploitation ne sont pas adaptés à des architectures multiprocesseurs. Pour [VxWorks] par exemple, les files de messages (message queues) ne peuvent être utilisées pour communiquer entre deux processeurs que si une mémoire partagée entre les deux processeurs existe. Aussi, une bibliothèque additionnelle, [VxMP], doit être achetée pour la gestion de telles mémoires partagées.

[pSOS] utilise une autre approche pour supporter les architectures multiprocesseurs. Il propose une version différente du système d'exploitation [pSOS+m] plus adaptée. Sous [pSOS+m], un nœud doit être configuré comme maître et tous les autres nœuds comme esclaves. [pSOS+m] permet une communication au delà du nœud. Les nœuds peuvent être interconnectés par de la mémoire partagée, par un échange de messages ou par des liens personnalisés.

Finalement, certains systèmes d'exploitation, comme [WinCe] [eCos] ou [MicroC/OSII], ne supportent pas les architectures multiprocesseurs. Ils n'offrent aucune facilité pour la communication inter-processeurs. L'utilisateur doit développer sa propre couche de communication inter-processeurs.

III.4.2 Adaptation à l'application

Sélection de services

La majorité des systèmes d'exploitation ont une architecture modulaire. Elle est basée sur les micros noyaux (microkernel, ou nanokernel) associés à des processus coopérants, parfois aussi appelés composants ou modules optionnels.

Le micro noyaux implémente les principaux services du système d'exploitation tels que la gestion de processus, l'ordonnancement, la gestion des interruptions. Par exemple, [Neutrino], [Wind] et [epsilon] sont les micros noyaux respectifs de [QNX], [VxWorks] et [OSE].

Les processus coopérants implémentent généralement des fonctionnalités additionnelles ou optionnelles comme un gestionnaire de fichier ou un gestionnaire de protocole [TCP/IP].

Avec la plupart des systèmes modulaires, il est possible de choisir les services qui seront fournis. Cela permet d'éliminer les services superflus. Cette sélection de services est rendue possible principalement par deux méthodes :

- L'utilisation d'une bibliothèque de modules compilés (codes objets) qui peuvent être liés pour obtenir le système final. C'est le cas de [VxWorks], [QNX], [Chorus OS], [Virtuoso], [C EXECUTIVE] et [EYRX].
- La présence de sources que l'on peut choisir d'inclure ou non dans le flot de compilation. C'est le cas de [LynxOS], [NucleusPLUS], [RTXC], [eCos] et [MicroC/OS-II].

Un paramètre important est la finesse du grain des services : plus le grain est fin plus le système résultant sera petit, mais plus il sera complexe à obtenir (des outils peuvent faciliter le choix de services à grain fin). Cependant, comme constaté dans [Grandpierre 00], les services fournis par l'ensemble des systèmes d'exploitation modulaires restent très généraux (par exemple boîte aux lettres), ce qui a pour conséquence un grain de service assez grossier. Des services plus précis auraient permis un grain beaucoup plus fin, mais cela aurait fortement augmenté leur nombre dans les bibliothèques.

Une meilleure optimisation de la taille du système d'exploitation peut être obtenue en utilisant des outils fournis par l'éditeur. Par exemple, l'outil [dietician] de [QNX] permet d'enlever automatiquement toutes les fonctions non nécessaires du système d'exploitation avant la phase de l'édition des liens.

Finalement, pour adapter le système d'exploitation à l'application, le concepteur peut parfois aussi implémenter son propre service spécifique à l'application comme processus coopérant avec le micro noyau, ce service fera alors partie des services du système d'exploitation (possibilité offerte par [QNX], [eCos]). Pour [eCos], le concepteur peut définir ses propres modules en utilisant un

langage spécifique à ce système d'exploitation. Puis, il utilise un second langage qui exprime les dépendances entre les différents modules de [eCos] et les siens.

Configuration des services sélectionnés

Certains systèmes modulaires proposent en plus du choix de services à inclure la possibilité de les configurer. Cette configuration est effectuée par le biais de macros (basées sur le préprocesseur du langage C [Stallman 91]), ce qui permet de définir les paramètres de configuration. Parmi ce type de système, nous avons : [LynxOS], [VxWorks], [eCos], [RTXC], [C EXECUTIVE], [EYRX].

Ces macros apportent plus de finesse pour la conception du système final. Elles restent cependant limitées à la définition de paramètres et à l'inhibition de certaines parties du code.

Stratégie de la gestion de la mémoire

La présence d'un mécanisme de gestion mémoire (MMU) permet de concevoir des applications plus fiables, un processus défaillant ne perturbe pas des processus corrects, mais au coût d'un changement de contexte plus long. Ne pas utiliser un MMU permet donc d'améliorer les performances mais diminue aussi la fiabilité totale du système.

La stratégie de la gestion de la mémoire diffère d'un système d'exploitation à un autre :

- Certains systèmes d'exploitation, comme [QNX], [VxWorks6.0], [OnCore OS], incluent par défaut un mécanisme de gestion de la mémoire (i.e., ne peut pas être enlevé du système d'exploitation).
- D'autres, comme [pSOS], le fournissent comme un composant optionnel (i.e. lors de la configuration le concepteur décide de sa présence)
- Pour d'autres systèmes d'exploitation il est additionnel (i.e. payant ex : [VxWorks5.0] et son composant [VxVMI]).

Une variante du mécanisme de gestion de la mémoire est le domaine de protection (ou protection domain) utilisé par [VxWorks AE]. Cette variante consiste en un gestionnaire de mémoire plus léger qu'une MMU. Le principe est de grouper les processus en domaines et de protéger ces derniers. C'est une protection mémoire moins élevée pour les processus (un processus défaillant peut affecter un autre processus correct s'ils sont dans le même domaine) mais qui permet un changement de contexte plus rapide (moins de contexte à sauvegarder si on change d'un processus à un autre dans le même domaine).

III.4.3 Automatisation de la conception

La configuration des systèmes d'exploitation est facilitée par l'utilisation de constructeurs de système d'exploitation. Les constructeurs de système d'exploitation sont en fait des interfaces utilisateur donnant accès à un ensemble d'outils permettant la configuration, la compilation et le débogage d'applications avec utilisation d'un système d'exploitation modulaire. Ils sont appelés IDE

pour *integrated development environment*. Ces IDE permettent de générer automatiquement une version adaptée et configurée du système d'exploitation.

Ces environnements sont généralement liés à un système d'exploitation donné, comme [pRISM+] pour [pSOS], ou bien [System builder tool] pour [QNX]. Ils encapsulent la modularité de ces systèmes pour en faciliter l'accès à l'utilisateur.

Un bon exemple de constructeur de système d'exploitation est [Tornado II] pour le système [VxWorks] de [Windriver]. Il permet à l'utilisateur de réaliser facilement les opérations suivantes :

- Choix des services du système d'exploitation : un système de dépendances entre services aide au choix interactif des services souhaités pour l'application.
- Configuration du système d'exploitation : les macros de paramétrage peuvent être modifiées par l'intermédiaire de l'interface.
- Compilation, choix de la cible : si l'utilisateur dispose de plusieurs versions du système pour des cibles différentes, il peut effectuer les choix à partir de l'interface. Les fichiers de compilation (makefile) seront générés. La compilation est elle-même contrôlable à partir de l'interface.
- Interface avec la cible : il est possible de transférer l'exécutable sur l'architecture cible et de l'exécuter avec débogage.
- Extension : l'interface peut être étendue par l'ajout de plugiciels (plug-in).

Certains éditeurs, comme ceux de [QNX] et [OSE] proposent des outils de test automatiques pour valider le système d'exploitation conçu.

[Dietician] est un outil intéressant fourni par [QNX] qui permet d'optimiser automatiquement la taille du système d'exploitation généré à travers l'IDE, seules les éléments nécessaires et suffisants de la bibliothèque seront inclus dans la version finale.

III.4.4 Quelques solutions issues du domaine de la recherche

III.4.4.1 Systèmes d'exploitation matériels

Implémenter certaines fonctionnalités du système d'exploitation en matériel plutôt qu'en logiciel peut s'avérer comme une solution très intéressante. Par exemple, un ordonnanceur matériel est plus rapide et plus prédictible qu'un ordonnanceur logiciel. De plus, il libère le processeur pour qu'il exécute en même temps le code d'une tâche de l'application : l'ordonnancement et l'exécution sont alors fait en parallèle. Par contre la rigidité du matériel ne permet pas de développer des systèmes d'exploitation facilement adaptables à un nombre variable de processeurs.

On peut trouver dans la littérature des travaux portant sur le partitionnement logiciel matériel du système d'exploitation :

- [Lindh 98] présente un accélérateur matériel pour un système d'exploitation temps réel pour architectures monoprocesseur et multiprocesseur. Une unité temps réel (RTU pour *real time unit*) a été développée. Elle implémente une partie du noyau temps réel en matériel. L'ordonnancement est basé sur un déclenchement périodique des tâches.

L'utilisation de la RTU apporte les avantages suivants par rapport à un système d'exploitation logiciel : plus de déterminisme, isolation des services du noyau (gestionnaire d'interruptions, sémaphores, IPC) de l'application, libération de l'interruption d'horloge du processeur.

La RTU ne peut gérer que 3 processus à la fois. Son architecture interne est fixée et ne peut être adaptée à d'autres applications ou réutilisée si une nouvelle topologie d'interconnexion est requise. Elle ne donne pas la possibilité de choisir quelles fonctionnalités du système d'exploitation sont logicielles et les quelles sont matérielles. Cette RTU exécutera toujours la même liste de fonctions quelque soit l'application (n'est pas configurable).

- [Lee 02] présentent un flot pour la génération des différentes variantes de système d'exploitation basées sur une version logicielle ou matérielle d'un gestionnaire de cache (noté SoCLC pour *system-on-a-chip lock cache*) [Akgul 01], d'un détecteur d'inter blocage (noté SoCDDU pour *system-on-a-chip deadlock detection unit*) [Shiu 01] et d'une unité de gestion de la mémoire (notée SoCDMMU pour *system-on-a-chip dynamic memory management unit*) [Shalan 00].

Ce flot propose une approche pour automatiser le partitionnement en logiciel et en matériel du système d'exploitation en des parties préconçues. Ce travail peut être considéré comme un outil d'aide au partitionnement : en utilisant le flot proposé, l'utilisateur peut explorer plusieurs compromis entre des configurations à base de modules logiciels ou matériels en les simulant [Micheli 96] [Gupta 95].

Ce flot est basé sur une bibliothèque de composants matériels, limitée aux SoCLC, SoCDDU et SoCMMU. Ceci implique que les capacités matérielles du système d'exploitation sont limitées à résoudre les problèmes de cache, d'inter blocage et de gestion de la mémoire. Aussi ce travail ne couvre pas la configuration des systèmes à base de processeurs hétérogènes avec un partitionnement en logiciel et matériel propre à chaque système d'exploitation.

- [Lindsley 02] propose un accélérateur d'ordonnancement de tâches (noté TSA pour *task scheduling accelerator*) qui implémente en matériel l'ordonnancement des tâches sans imposer de nouvelles contraintes : pas de limitation du nombre des tâches, pas de limitation sur la profondeur des sémaphores, aussi, la détermination de la prochaine tâche prête est fonction uniquement du nombre de priorités et non pas du nombre de tâches.

Dans cette approche, les décisions d'ordonnancement sont prises au niveau d'un accélérateur externe pour permettre une décision plus rapide, en parallèle avec une activité sur le processeur. Cette approche est plus rapide qu'un ordonnancement logiciel et n'exécute pas la fonctionnalité d'ordonnancement en série avec celle des tâches de l'application. Cependant, cette approche se limite à porter seulement l'ordonnancement au matériel.

III.4.4.2 L'approche SynDEx

La solution SynDEx [Grandpierre 00] propose de générer automatiquement un exécutif distribué temps réel implémentant une application sur une architecture parallèle. La description de l'application donnée en entrée est un graphe flot de données qui a été automatiquement

ordonnancé. La partie système d'exploitation est donc réduite aux fonctions de communication et de synchronisation.

L'objectif de cette approche est de concevoir automatiquement une application flot de données sur une architecture multiprocesseur, avec une seule tâche par processeur pour s'abstraire des indéterminismes de l'ordonnancement dynamique.

SynDEx est un environnement basé sur une méthodologie globale (niveau système) prenant en compte toutes les phases du développement, de la spécification haut niveau des algorithmes et des architectures matérielles, à l'exécution du code. La spécification repose sur des modèles et l'exécution du code repose sur des techniques d'ordonnancement hors-ligne.

Dans cet environnement, on modélise une architecture hétérogène distribuée par un graphe orienté (S, A) où S est l'ensemble des sommets de ce graphe et A l'ensemble de ses arcs. Chaque sommet est une machine à états finis qui produit et consomme des données, chaque arc correspond à une connexion entre deux machines à états finis. Un ensemble de règles est utilisé pour représenter une architecture sous forme d'un graphe : un DSP par exemple est considéré comme capable de lire simultanément une instruction et d'accéder (en lecture ou écriture) à deux opérandes. Il sera alors modélisé par un sommet opérateur connecté à trois mémoires (figure III-4).

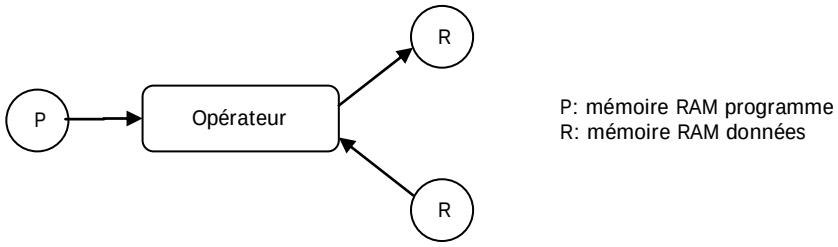


Figure III-4 : Modélisation d'un DSP dans l'approche SynDEx

L'algorithme est aussi spécifié par un hypergraphe flot de données. Un tel graphe peut être obtenu par la transformation d'un algorithme exprimé dans un langage synchrone (Esterel, Lustre…). Les sommets de l'hypergraphe orienté sont les opérations de calcul de l'algorithme et les arcs sont les dépendances de données entre opérations, également appelés dépendances de données inter-opérations. Chaque opération est une suite indivisible d'instructions, appelée région atomique dans [Zwiers 93], c'est-à-dire qu'on ne pourra pas la partager pour en exécuter une partie sur un processeur et une autre sur un autre processeur. Autrement dit, ce sont des opérations non préemptives. Les arcs induisent un ordre partiel d'exécution sur les opérations. Une opération peut s'exécuter lorsque ses données d'entrée sont présentes. Elle consomme ses données d'entrée et produit des données en sortie qui sont ensuite utilisées par ses successeurs.

L'environnement SynDEx permet, à partir d'un graphe d'algorithme et d'un graphe d'architecture, de construire l'ensemble des graphes d'implantation dits valides au moyen de la composition de trois relations (le routage, la distribution et l'ordonnancement) (figure III-5). Le choix des services de communication et de synchronisation est effectué en fonction des opérations

effectuées par l'application. La génération du code du système d'exploitation, c'est-à-dire des fonctions de communication et de synchronisation, est effectuée à l'aide du programme d'expansion de [macro m4]. Ce programme traite un langage de macro à fort pouvoir d'expression, mais basé uniquement sur le principe de substitutions récursives. Sa puissance d'expression permet d'obtenir une très grande finesse lors de la génération de code, permettant par exemple la modification des algorithmes du code généré en fonction des paramètres de l'architecture cible.

SynDEx permet ensuite de générer automatiquement les exécutifs distribués (système d'exploitation) taillés sur mesure pour chaque application.

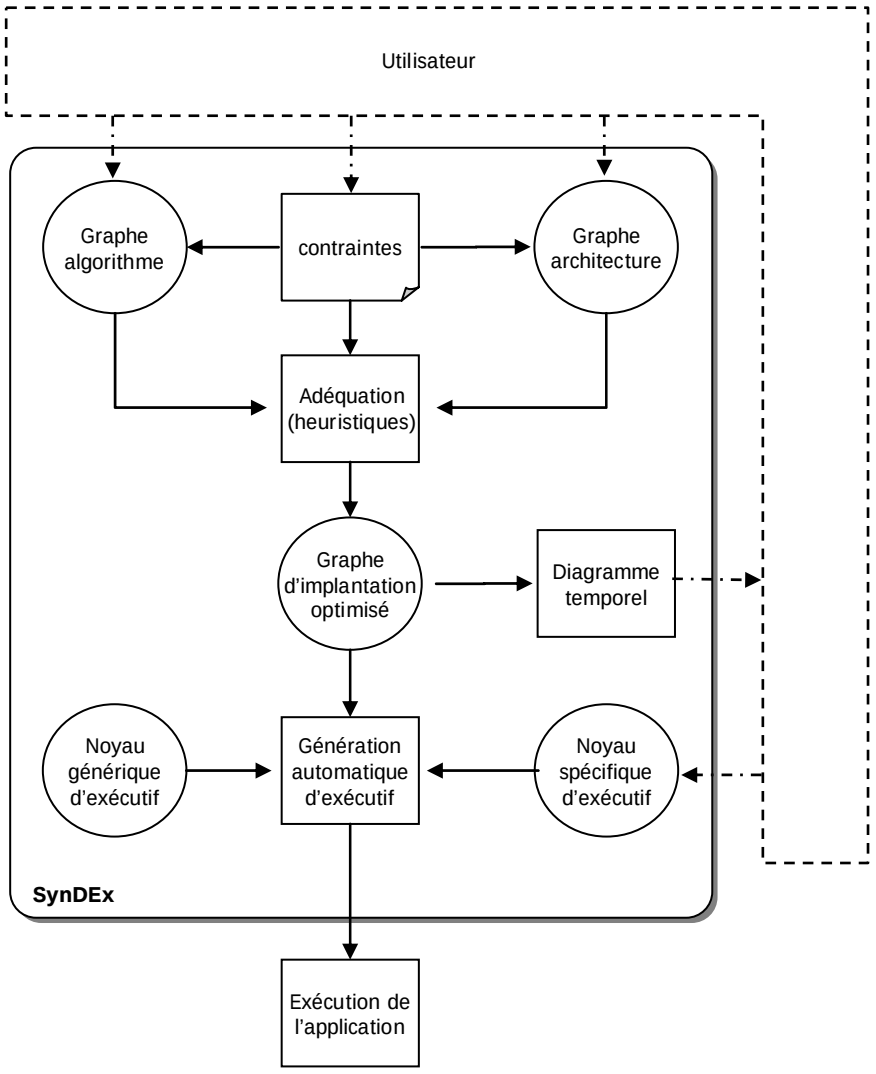


Figure III-5 : Environnement SynDEx

Évaluation de la solution SynDEx

- Solution déterministe donc adaptée aux applications type temps réel.
- Elle bénéficie d'une base théorique (représentation par des graphes orientés) des preuves peuvent être fournies plus facilement.

- L'environnement SynDEx propose une solution complètement déterministe de génération d'exécutif, mais se restreint au cadre des applications flot de données uniquement. Les exécutifs générés sont de faible taille et donc bien adaptés aux MPSoC. Par contre, le support unique d'applications flot de données limite son utilisation pour les MPSoC où les applications peuvent aussi contenir du contrôle. Cette limitation a été corrigée dans les dernières versions de SynDEx [Pernet 02].

III.5 Vers une conception automatisée des interfaces logicielles/matérielles : le flot ROSES

Dans cette partie nous présentons le flot ROSES, solution en cours de développement au sein du groupe SLS, pour la conception des interfaces logicielles/matérielles. Ce flot permet la conception des systèmes MPSoC à partir d'une description d'architecture virtuelle en utilisant une approche de raffinement de la communication.

La figure III-6 montre une représentation simplifiée du flot. À partir d'une description d'un système MPSoC sous forme d'une architecture virtuelle annotée, le flot ROSES permet la génération automatique de l'adaptation logicielle et de l'adaptation matérielle ainsi qu'une adaptation pour la cosimulation (pour permettre par exemple la communication entre deux modules écrits avec des langages différents).

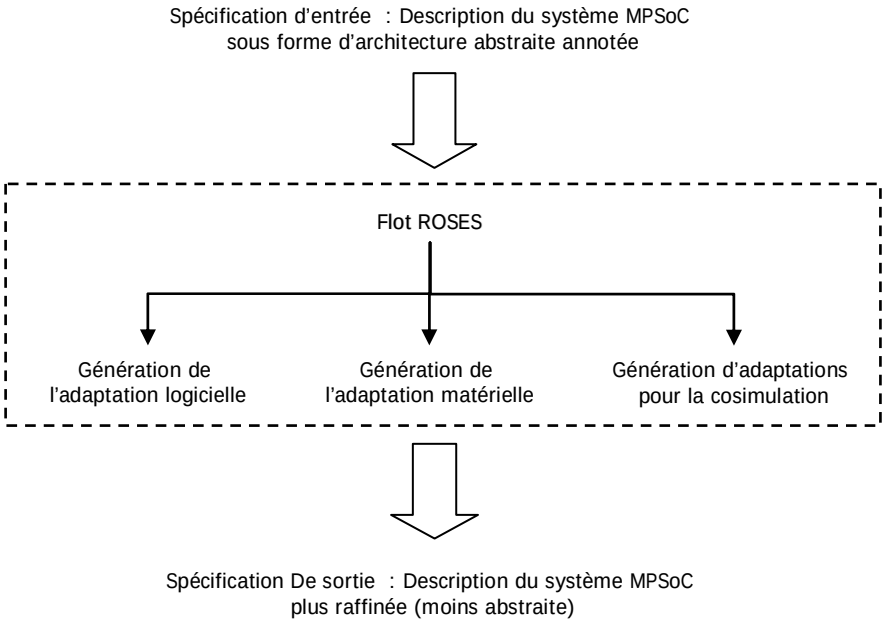


Figure III-6 : Vue simplifiée du flot ROSES

Dans la suite de ce paragraphe, nous détaillerons la description d'entrée du flot ROSES et nous présenterons les outils et le mécanisme de génération des adaptations logicielles et matérielles utilisés dans ROSES.

III.5.1 Spécification d'entrée du flot ROSES : La spécification VADeL

VADeL : Un langage de spécification d'architectures virtuelles

L'entrée du flot ROSES est décrite en un langage de spécification développé par le groupe SLS. Ce langage est une extension de SystemC [Systemc 00] nommé VADeL [Cesario 01], pour *virtual architecture description language*, par référence au concept de modules virtuels.

Le concept de virtualisation de l'architecture est un concept puissant. Il représente une abstraction à la fois du logiciel et du matériel : un module virtuel peut être transposé vers un composant logiciel (tâche) ou matériel (IP) du système réel [Casseau 02].

En VADeL, Un système est décrit comme une architecture virtuelle formée par un ensemble de composants virtuels interconnectés par des canaux virtuels de communication (figure III-7) :

- Un composant virtuel consiste en un module et une interface abstraite (aussi appelée adaptateur) :
  - Un module correspond à une (ou plusieurs) tâche(s) logicielle(s), ou une (ou plusieurs) fonction(s) matérielle(s). Il peut, être caché (boite noire), contenir des références à un comportement (ex : fichier C ou VHDL/matériel connu). Il peut aussi être un sous-système et décrire une hiérarchie.
  - L'interface abstraite (partie grisée sur la figure III-7) adapte l'accès du module aux canaux connectés au composant virtuel. Elle donne les ports par lesquels le module communique.
- Un canal virtuel (aussi appelé canal abstrait) est un canal de communication, décrit à un haut niveau d'abstraction, qui représente un ou plusieurs canaux de bas niveau d'abstraction. Il peut contenir des références à un comportement ou contenir d'autres canaux virtuels (une hiérarchie).

Un canal virtuel permet la communication entre des composants virtuels. Un composant virtuel encapsule un module. Le module accède au canal virtuel en traversant une interface abstraite (port virtuel). L'interface abstraite est composée de ports internes (connectés aux modules) et de ports externes (connectés aux canaux virtuels). Elle adapte les protocoles et les niveaux d'abstractions entre les ports externes et les ports internes. Ceci permet aux modules des composants virtuels de ne pas être décrits au même niveau d'abstraction et de pouvoir s'échanger des informations. En VADeL, le module et le(s) canal(ux) peut(vent) être différent en terme de : protocole de communication, niveau d'abstraction et langage de spécification.

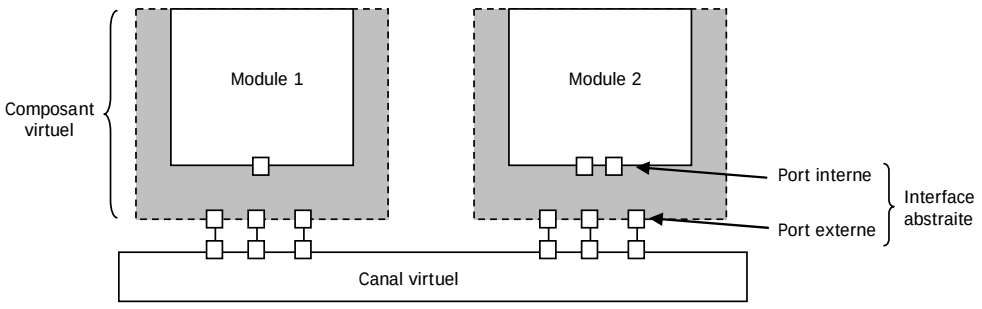


Figure III-7 : Description d'un système en VADeL

VADeL : La notion de description annotée

Une fois l'architecture abstraite du système décrite en VADeL, la description est annotée avec des paramètres. L'annotation permet de guider les outils internes de ROSES afin de générer les adaptations logicielles et matérielles lors du raffinement de l'interface abstraite. En VADeL, un système peut être décrit à différents niveaux d'abstractions : plus le niveau d'abstraction est élevé, plus le nombre de paramètres à spécifier est réduit.

Une description VADeL annotée contient les informations suivantes :

- Des informations topologiques : il s'agit du nombre de sous-systèmes processeurs, du nombre de tâches logicielles et de leur transposition, par quels ports les tâches communiquent-elles, etc... Ces informations proviennent de la hiérarchie des modules de la description.
- Les caractéristiques des sous-systèmes processeurs : les types de processeur (68000, ARM7, etc...), les ressources locales (mémoire locale et interruptions), etc... Ces informations font partie des paramètres des modules processeurs.
- les médias utilisés pour les interactions : ces informations sont situées au niveau des ports des processeurs et des canaux de communication.
- Des informations sur l'API : il s'agit de l'API utilisée par les tâches. Ces informations font parties des paramètres des ports des tâches.
- Les paramètres d'allocation pour les tâches et les divers éléments de l'adaptation logicielle. Dans ces paramètres se trouvent les adresses, les tailles, mais aussi les types de données ou les priorités des tâches. Ces paramètres peuvent être sous la forme de valeurs fixées, ou sous la forme de demandes d'allocations. Ces informations sont données comme annotations de la description initiale suite à l'étape d'affectation globale de la mémoire.
- Les caractéristiques du canal de communication : le protocole utilisé, la taille du bus, la taille de la FIFO. Néanmoins, les détails de réalisation n'apparaissent pas explicitement (le bus de données, les signaux de contrôle, d'adresses, ...).

III.5.2 Les outils et le mécanisme de génération des adaptations logicielles et matérielles utilisés dans le flot ROSES

Le flot ROSES (figure III-8) permet la génération automatique des interfaces logicielles/matérielles à partir d'une spécification du système en VADeL et d'annotations pour guider le raffinement (protocoles, adressage et autres paramètres). La génération est basée sur un assemblage d'éléments [Cesario 02]. Les éléments sont issus de bibliothèques d'éléments. Ils sont choisis sur la base d'un graphe de dépendance de services [Gauthier 01] [Sarmento 05] [Kriaa 05]. Des outils internes à ROSES automatisent la génération des interfaces. Il s'agit de l'outil ASOG [Gauthier 01] [Paviot 04] pour la génération de l'adaptation logicielle (basée sur une architecture de réseau de systèmes d'exploitation), de l'outil ASAG [Lyonnard 03] [Grasset 05] pour la génération de l'adaptation matérielle et de l'outil CosimX [Nicolescu 02-B] [Kriaa 05] pour la génération d'adaptateurs de (co)simulation.

La sortie du flot ROSES est une description plus raffinée du système. Le niveau d'abstraction de la sortie produite dépend de la description en entrée et du nombre de paramètres annotés. La description la plus raffinée est au niveau RTL.

Nous détaillerons dans la suite les différentes étapes, représentées par des rectangles dans la figure III-8, du flot ROSES. Nous présenterons alors le format CoLIF, puis la notion de graphe de dépendance de services et les outils de génération automatique.

III.5.2.1 Format de représentation interne : CoLIF

La première étape du flot ROSES consiste à traduire la spécification d'entrée décrite en VADeL annotée en un langage intermédiaire noté CoLIF [Cesario 01] (figure III-8 – (a)).

Dans la spécification d'entrée, un module logiciel (par exemple le module 2) est écrit en utilisant les primitives offertes par l'API associée aux ports internes de l'interface abstraite. Cette API n'est pas fixe et elle dépend des besoins du logiciel.

La spécification CoLIF, basée sur le langage XML, sera utilisée par les différents outils de génération automatique. Elle décrit un système comme un ensemble d'objets interconnectés de trois types : les modules, les ports et les liens. Un objet, quel que soit son type, est composé de deux parties. La première partie, nommée «entity», permet la connexion avec les autres objets. La deuxième partie, nommée «content», fournit une référence à un comportement ou des instances d'autres objets. Cette possibilité d'instancier des éléments permet le développement des modules, ports et liens hiérarchiques.

III.5.2.2 Bibliothèques internes de ROSES et notion de graphe de dépendance de services

Les bibliothèques internes de ROSES fournissent les éléments qui, paramétrés et assemblés, produiront le code des adaptations logicielles et matérielles. Une bibliothèque utilise une représentation relationnelle décrite dans le langage Lidel [Gauthier 01] (figure III-8 – (e)), donnant les informations permettant de retrouver et d'identifier les divers éléments. Une seconde bibliothèque utilise une représentation comportementale (qui peut être décrite dans n'importe quel langage, (figure III-8 – (f))) qui donne le code de ces éléments sous la forme de macros.

Chaque élément de la bibliothèque fournit et/ou requiert un ou plusieurs services. Par exemple, l'élément E3 de la figure III-9 – (b) fournit le service S3 et requiert les services S4 et S7. La mise en correspondance des services fournis et requis de tous les éléments de la bibliothèque permet la construction d'un graphe maximal de dépendance de services (SDG, pour *service dependency graph*) [Sarmento 05]. Un graphe de dépendance de services permet alors d'adapter deux interfaces. Dans l'exemple présenté dans la figure III-9, le SDG permet d'adapter l'interface du module logiciel 2 à l'interface du réseau de communication. La première interface est l'API des ports internes de la spécification VADeL, tandis que l'interface du réseau est l'ensemble de services qu'il offre.

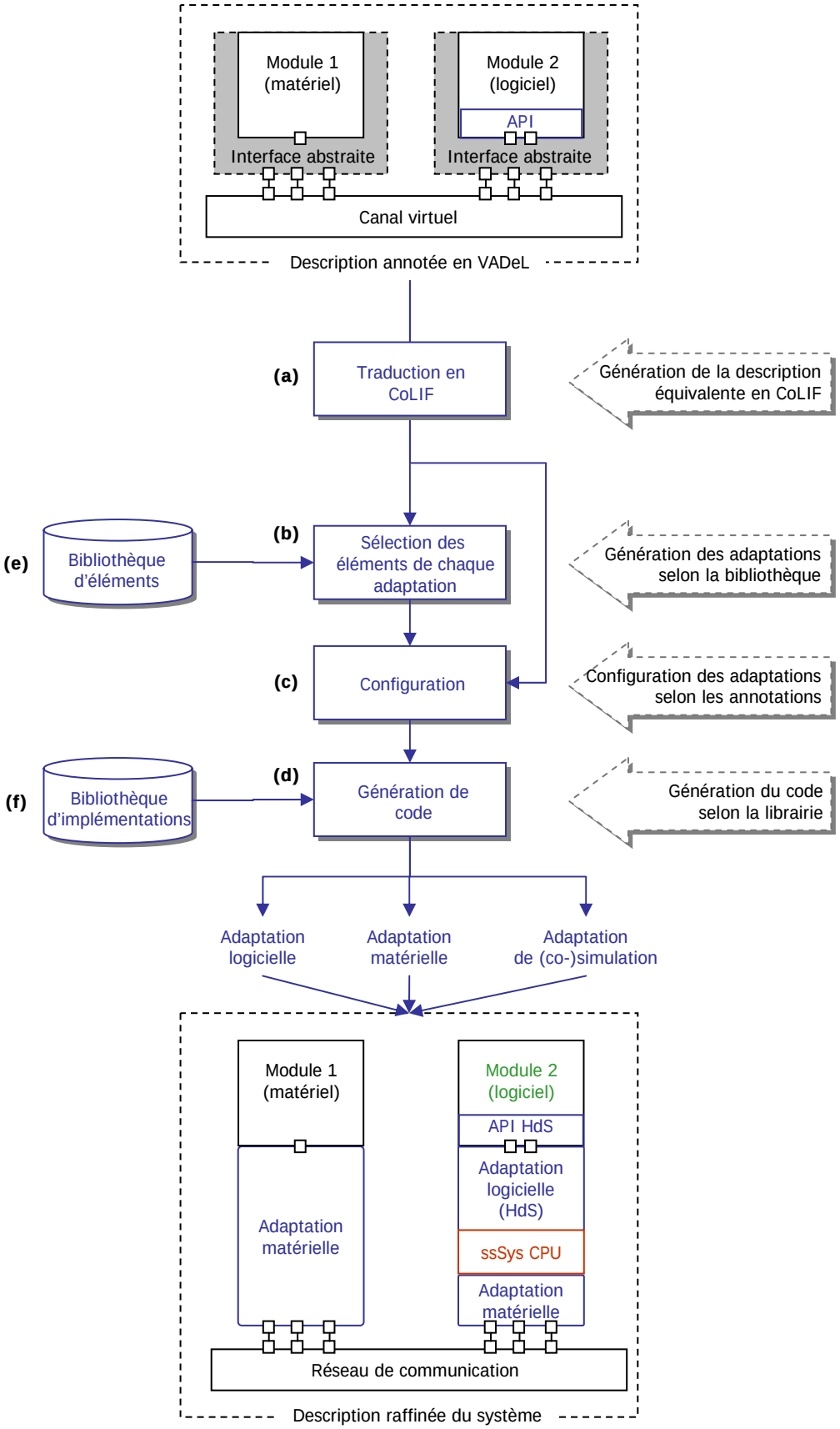


Figure III-8 : Le flot ROSES

III.5.2.3 Outils de génération des adaptations logicielles et matérielles

La génération des adaptations logicielles et matérielles se base sur un modèle générique d'adaptateur, lui-même basé sur l'assemblage de composants [Gauthier 01] [Sarmento 05] [Kriaa 05]. La figure III-8 représente sous forme de rectangles les différentes étapes pour la génération des adaptations logicielles et matérielles. Après une traduction en CoLIF (figure III-8 – (a)) de la description annotée en VADeL, l'étape suivante consiste à identifier un sous graphe, du graphe maximal de la bibliothèque, qui réalise l'adaptation requise. Par exemple, le sous graphe de dépendance de services représenté par la partie grisée de la figure III-9 – (b) permet l'adaptation entre les services requis/fournis par les ports du module 2 (exprimée à travers des primitives de l'API) avec ceux du réseau de communication. Les éléments du sous graphe de dépendance de service sont sélectionnés de façon à ce que seuls les services requis seront implémentés dans l'adaptateur généré.

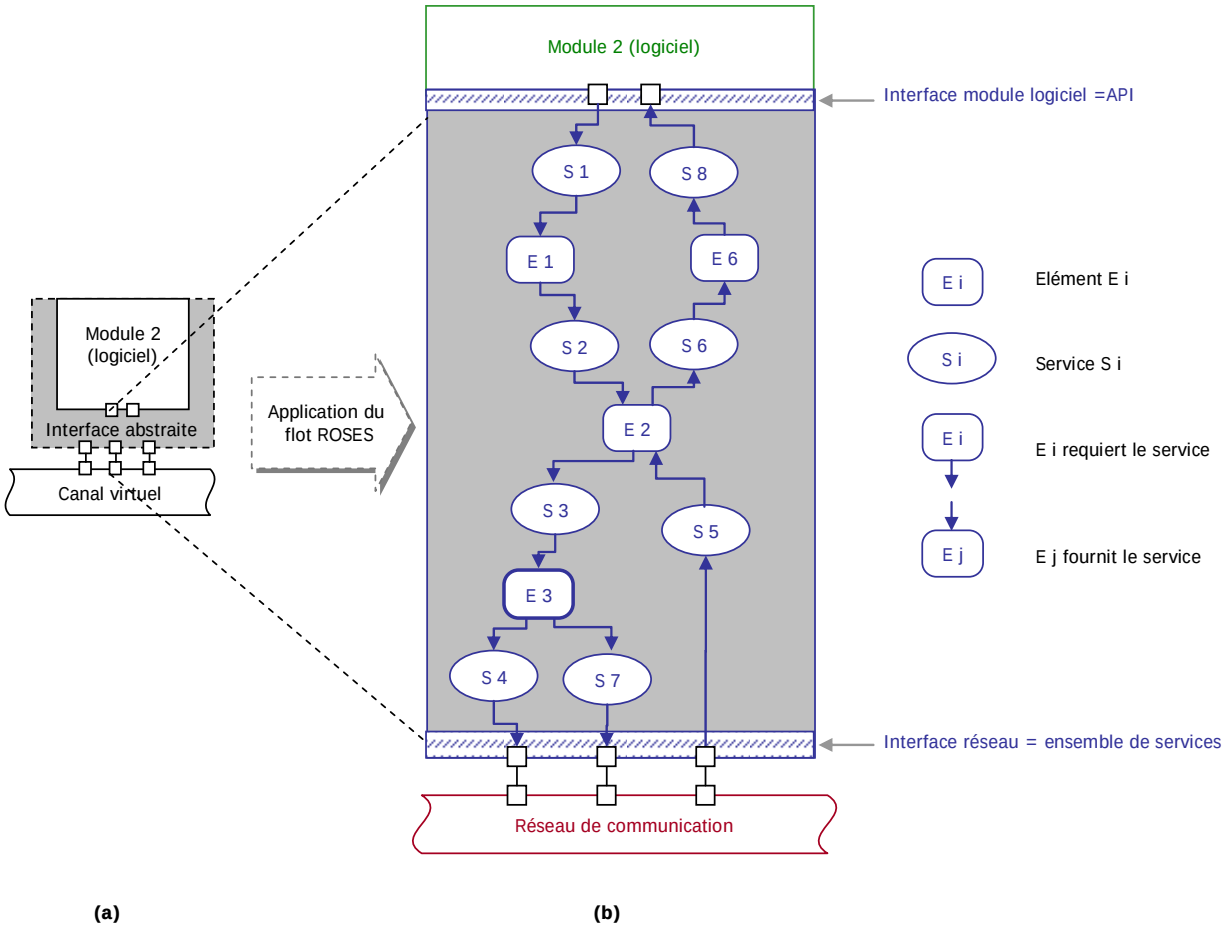


Figure III-9 : Exemple d'un sous graphe de dépendance de services = modèle d'adaptateur

L'étape suivante est la configuration des éléments (figure III-8 – (c)). Il s'agit dans cette étape de rassembler tous les paramètres concernant un élément donné à partir des annotations de la description VADeL traduite en CoLIF. La dernière étape est la génération du code (figure III-8 –

(d)) des différentes adaptations. Cette génération est basée sur une bibliothèque d'implémentations.

III.5.3 Utilisation du flot ROSES dans le cadre de la conception des systèmes multiprocesseurs monopuces et des modèles de programmation parallèle

Notion de modèle de programmation parallèle

Un modèle de programmation est une interface séparant les propriétés de bas niveau de celles de haut niveau. C'est une machine abstraite qui peut être considérée à des niveaux d'abstraction différents. Elle définie une API, à travers laquelle elle fournit des services au niveau de programmation supérieure et requiert une infrastructure qui implémente ces services. En ce sens tout langage de programmation est un modèle de programmation. Un modèle de programmation parallèle est alors une machine abstraite à fort parallèlisme. Elle fournit des services qui permettent d'exprimer du parallèlisme.

Utilisation des modèles de programmation parallèle pour la conception des MPSoC

Le logiciel applicatif des systèmes embarqués est caractérisé par sa complexité croissante. L'utilisation de l'API d'un modèle de programmation parallèle peut simplifier cette complexité car elle offre une abstraction de l'architecture matérielle.

D'une autre part, pour des raisons de coût et de ressources limitées (tout particulièrement, en terme d'énergie), les systèmes embarqués adoptent des architectures matérielles multiprocesseurs. Un modèle de programmation parallèle permet de mieux abstraire ce type d'architecture qu'un modèle d'abstraction séquentiel (qui n'offre pas une API qui permet d'exprimer du parallèlisme).

De plus, à un haut niveau d'abstraction, l'interface logicielle/matérielle des systèmes MPSoC hétérogènes et hautement parallèles peut être abstraite par l'API d'un modèle de programmation parallèle. Le raffinement de cette API sur l'architecture matérielle ciblée permet le développement de l'interface logicielle/matérielle.

L'API du modèle de programmation peut être utilisée comme un contrat entre l'équipe qui développe la partie matérielle et celle qui développe la partie logicielle (l'application) du système MPSoC. Elle est alors la vue dont disposeraient les concepteurs du logiciel applicatif de l'architecture matérielle cible [Paul 03]. La conception des parties logicielles et matérielles peut alors être faite en parallèle et permettre ainsi de réduire le temps de conception. La tâche de l'équipe qui développe le logiciel applicatif est d'optimiser l'application par rapport à l'API du modèle de programmation. Celle de l'équipe qui développe la partie matérielle est d'optimiser l'architecture pour l'implémentation des primitives de l'API du modèle de programmation (ajout d'accélérateurs matériels par exemple).

Pour toutes ces raisons, nous pensons que l'utilisation des modèles de programmation parallèle pour la conception des systèmes embarqués est une approche très prometteuse.

ROSES, conception de MPSoC et modèles de programmation parallèle

Le flot ROSES permet de générer une adaptation (logicielle ou matérielle) entre deux interfaces de deux modules. Pour la conception des systèmes MPSoC, nous proposons d'utiliser l'API des modèles de programmation parallèle comme interface pour les modules logiciels. Puis d'utiliser l'approche et les outils du flot ROSES pour raffiner automatiquement les primitives de l'API sur l'architecture matérielle ciblée.

Cette approche de conception présente les avantages suivants :

- Accélération de la conception par la génération automatique de l'implémentation de l'interface logicielle/matérielle. De plus, le flot permet aussi la génération des adaptations pour une cosimulation multi-niveaux. L'utilisation de la cosimulation permet de détecter les bugs à une phase avancée du cycle de conception.
- Permet la réutilisation puisque les adaptations sont générées à partir de bibliothèques d'éléments et d'implémentations.
- Les adaptations générées sont optimisées et taillées sur mesure : elles n'implémentent que les primitives utilisées par l'application.
- Permet l'exploration de l'espace de conception des interfaces logicielles/matérielles. La génération des interfaces logicielles/matérielles repose sur un mécanisme d'assemblage et de configuration d'éléments de bibliothèques. Différents assemblages et différentes configurations couplés à des techniques d'estimation de performance en haut niveau [Posadas 04] [Bouchhima 04] [Bouchhima 05] permettent alors d'explorer l'espace de conception des interfaces logicielles/matérielles.

Toutefois, le succès de cette approche dépend primordialement de deux facteurs :

- La qualité des bibliothèques utilisées pour le raffinement du modèle de programmation parallèle.
  - De cette qualité dépend la qualité des interfaces générées par les outils du flot ROSES. Il faut toujours veiller à l'enrichir et à améliorer la qualité des sources. D'autant plus qu'enrichir les bibliothèques est un processus complexe. Déjà, il faut trouver la granularité adéquate de l'élément de façon à ce qu'il corresponde à une fonctionnalité bien déterminée. Puis, il faut insérer le nouvel élément dans le graphe maximal de dépendance de services. Ceci requiert la connaissance des langages internes des bibliothèques.
  - Le résultat généré est difficile à déboguer car il est fortement optimisé. L'absence d'une démarche formelle ne permet pas une génération valide par construction (le résultat de la composition de divers éléments déjà validés n'est pas forcément valide).
- Le choix du modèle de programmation parallèle, de son niveau d'abstraction et aussi des primitives de son API. Ce choix est de première importance pour les performances du système final.

Dans le chapitre suivant, nous proposerons un flot qui décrit cette approche de conception des systèmes MPSoC. Nous présenterons aussi une étude et une classification des modèles de programmation parallèle à fin de trouver des candidats pour la conception des systèmes MPSoC. Puis dans les chapitres V et VI nous présenterons deux expériences de conception de systèmes MPSoC qui utilisent cette approche.

III.6 Conclusion

Dans ce troisième chapitre, les approches pour l'implémentation de l'adaptation logicielle ont été discutées. Cette discussion a abouti à la conclusion que l'optimisation de l'implémentation de l'adaptation logicielle est fortement dépendante des capacités de configuration et d'adaptation du système d'exploitation utilisé. Une étude sur cette capacité a été exposée. Il en résulte que le pouvoir de configuration et d'adaptation diffère d'un système d'exploitation à un autre et que (1) généralement la configuration se fait à gros grains et (2) que le support d'architectures multiprocesseurs n'est pas largement adopté actuellement.

Dans la dernière partie de ce chapitre, le flot du groupe SLS pour la conception des interfaces logicielles/matérielles a été présenté. Une proposition de l'utilisation des APIs de modèles de programmation parallèle pour la spécification de l'entrée du flot a été formulée.

Une étude de modèles de programmation parallèle en vue de leur utilisation pour l'abstraction et la conception des interfaces logicielles/matérielles sera présentée dans le chapitre suivant.

Chapitre IV

Étude des modèles de programmation parallèle et proposition d'un flot de conception pour systèmes MPSoC

IV.1 Introduction

Les systèmes embarqués sont utilisés dans le cadre d'applications de plus en plus complexes. Pour maîtriser cette complexité, l'élévation du niveau d'abstraction lors de la conception de ces systèmes est requise. Dans ce contexte, l'utilisation de modèles de programmation parallèle, pour l'abstraction de l'architecture matérielle ainsi que de l'interface logicielle/matérielle est une approche prometteuse.

Dans ce chapitre, nous étudions les modèles de programmation parallèle pour trouver le modèle (la classe de modèles) le (la) plus adéquat(e) pour la conception des MPSoC hautement parallèles. Dans le premier paragraphe, nous définissons le terme modèle de programmation parallèle. Dans le second paragraphe, nous proposons un flot pour la conception des interfaces logicielles/matérielles des systèmes MPSoC, basé sur l'utilisation des modèles de programmation parallèle. Dans le dernier paragraphe, nous présentons d'abord une classification des modèles de programmation parallèle selon leurs degrés d'abstraction, puis nous analysons plus en détail certains modèles de programmation parallèle.

IV.2 Définition d'un modèle de programmation parallèle

Le terme «modèle de programmation parallèle» est couramment utilisé dans la littérature scientifique avec plusieurs significations globalement équivalentes. Dans ce paragraphe, certaines de ces définitions seront présentées puis nous proposerons notre propre définition.

- Un modèle de programmation, tel que défini par [Shah 03], est une abstraction qui expose seulement les détails importants de l'architecture nécessaires pour une implémentation efficace de l'application. C'est la vue que le programmeur possède de l'architecture, elle oscille entre opacité et visibilité. L'opacité en tant qu'abstraction de l'architecture sous-jacente, réduit le besoin du programmeur du logiciel applicatif de connaître en détail l'architecture qu'il cible et lui donne la possibilité de commencer la programmation de suite. La visibilité permet l'exploration de l'espace des solutions d'implémentation, donc l'optimisation de l'implémentation via des compromis entre différents paramètres de conception et ainsi l'exploitation de toute la puissance de l'architecture ciblée. Le modèle de programmation doit donc satisfaire deux besoins contradictoires du programmeur : le besoin d'une programmation simplifiée (une abstraction plus élevée) et le besoin d'une implémentation optimisée (connaître en détail l'architecture matérielle).

- Pour [Beidas 03], le modèle de programmation est une abstraction qui sépare l'application de l'architecture et qui peut être définie à différents niveaux. Une infrastructure logicielle matérielle est souvent requise pour réaliser cette abstraction. Ainsi, un jeu d'instructions est considéré comme un modèle de programmation à un bas niveau d'abstraction. En fait, il cache des détails

architecturaux comme le « *pipeline* » ou le « *out-of-order* » qui sont réalisés par une architecture matérielle complexe. De même, un langage de programmation est aussi un modèle de programmation se situant à un niveau d'abstraction plus élevé. Il cache plusieurs jeux d'instructions différents. Un compilateur réalise cette abstraction en générant le programme écrit avec le jeu d'instruction adéquat à l'architecture. Finalement, un middleware comme [CORBA] ou [DCOM] est considéré comme un modèle de programmation parallèle. Il s'agit d'une infrastructure qui est utilisée pour abstraire les détails d'un environnement distribué et pour implémenter une application distribuée comme une application séquentielle.

Notre définition d'un modèle de programmation parallèle

Dans ce travail, nous considérons le modèle de programmation comme une abstraction de l'interface logicielle/matérielle et de l'architecture matérielle. Cette abstraction peut être définie à plusieurs niveaux et elle représente, à chaque niveau, la vue dont dispose le programmeur du logiciel applicatif de l'architecture matérielle et de l'interface logicielle/matérielle :

- À un haut niveau d'abstraction, c'est une vue globale à gros grains de l'architecture qui considère, par exemple, uniquement le nombre d'unités d'exécutions et la topologie de l'interconnexion (point-à-point ou multipoint).
- À un bas niveau d'abstraction, c'est une vue détaillée à grains fins de l'architecture qui considère, par exemple, le type et l'architecture des unités d'exécution ainsi que le protocole de communication. Elle permet d'adresser les spécificités de l'architecture matérielle.

Les modèles de programmation parallèles sont des modèles qui définissent une abstraction des architectures matérielles à fort parallèlisme. Ils permettent d'exprimer le parallèlisme lors de la conception de l'application.

En pratique, un modèle de programmation est un ensemble de primitives constituant une API dont la sémantique est bien déterminée. Il définit une plateforme logicielle qui abstrait le matériel. Le raffinement du modèle de programmation parallèle sur l'architecture matérielle ciblée doit être ensuite réalisé pour permettre l'exécution du logiciel applicatif sur l'architecture matérielle. Le raffinement consiste à implémenter les primitives de l'API du modèle de programmation utilisées lors de la conception de l'application, sur l'architecture matérielle. Cette implémentation nécessite le développement d'une adaptation logicielle et/ou d'une adaptation matérielle.

L'utilisation des modèles de programmation parallèle pour la conception des MPSoC permet (1) de découpler la conception de la partie logicielle de celle de la partie matérielle et (2) d'assurer la portabilité du logiciel applicatif afin de permettre la réutilisation de certaines fonctionnalités lors de la conception de nouveaux systèmes MPSoC.

Nous proposons dans la section suivante, un flot de conception des interfaces logicielles/matérielles basé sur l'utilisation des modèles de programmation parallèles.

IV.3 Flot de conception des interfaces logicielles/matérielles pour les systèmes MPSoC, basé sur les modèles de programmation parallèle

Dans ce paragraphe nous proposons un flot pour la conception des interfaces logicielles/matérielles des systèmes MPSoC. Ce flot est basé sur l'utilisation et le raffinement des modèles de programmation parallèle.

La figure IV-1 montre le flot de conception proposé. Dans cette figure, les ovales représentent les étapes de la conception et les rectangles représentent une entrée ou une sortie d'une étape de la conception.

Les étapes du flot de conception proposé sont :

- Parallèlisation de la spécification de l'application en utilisant les primitives de l'API du modèle de programmation parallèle et sa et transposition sur les ressources de l'architecture multiprocesseur ciblée ;
- Le débogage en haut niveau du logiciel applicatif ;
- La conception du logiciel dépendant du matériel (HdS) ;
- La conception du prototype matériel avant que l'architecture matérielle finale ne soit prête ;
- Le débogage des interfaces logicielles/matérielles ;
- La conception de l'architecture matérielle ;
- L'intégration finale, quand la partie matérielle et la partie logicielle sont finies.

Dans la suite, nous détaillerons les différentes étapes de conception.

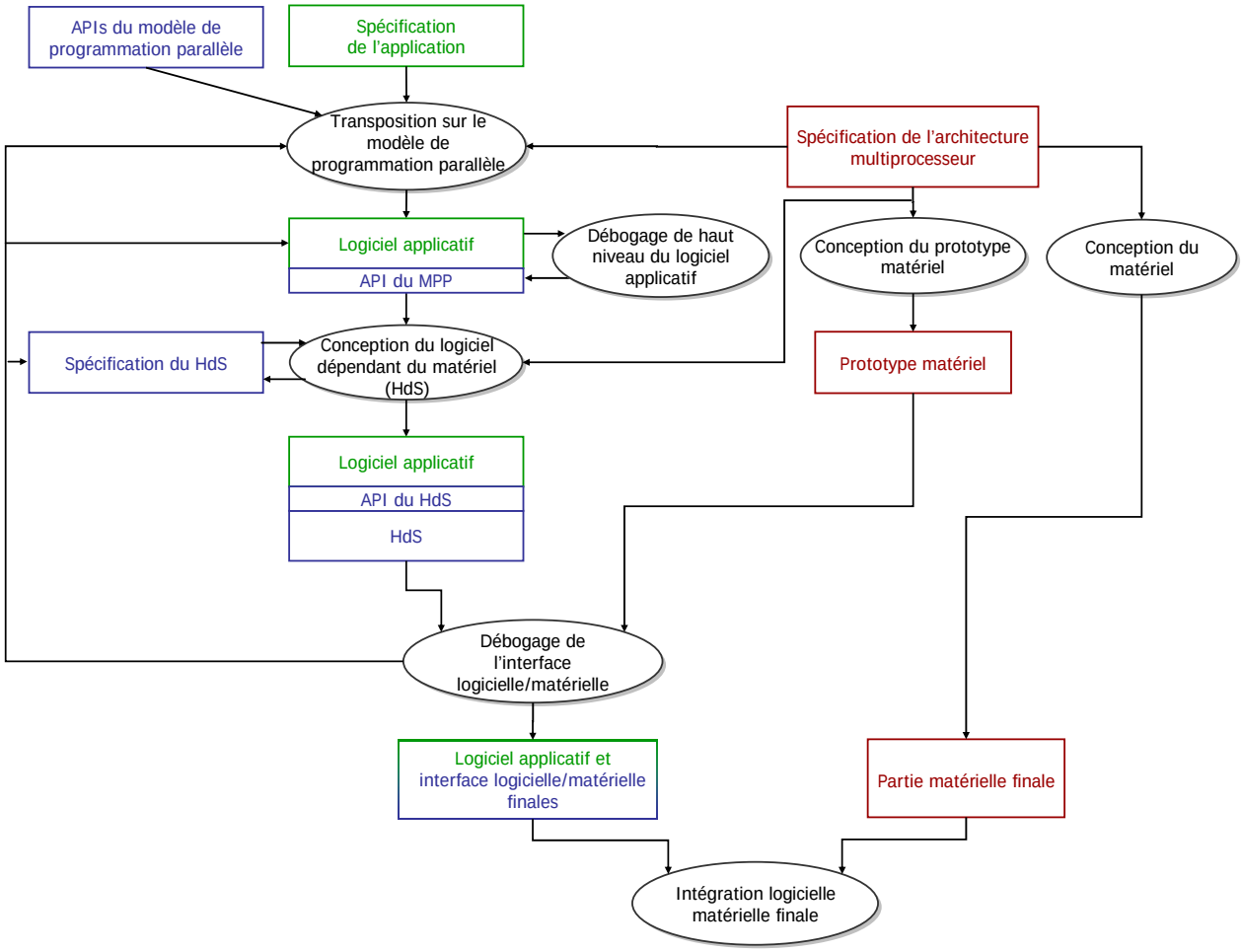


Figure IV-1 : Flot de conception proposé

IV.3.1 Parallèlisation de la spécification de l'application et sa transposition sur une architecture multiprocesseur

Nous présentons dans cette section, la première étape du flot qui consiste à parallèliser la spécification initiale de l'application et à la transposer sur l'architecture multiprocesseur ciblée.

IV.3.1.1 Parallèlisation de la spécification de l'application

La spécification initiale de l'application peut être décrite en utilisant des modèles de spécification de haut niveau [Brisolara 04] ou bien, en utilisant des modèles de plus bas niveau tel que les langages impératifs.

Pour porter l'application sur une architecture multiprocesseur, le code source du logiciel applicatif doit être distribué, c'est-à-dire parallèlisé en utilisant un modèle de programmation parallèle, sur l'architecture. La parallèlisation de ce code peut être faite au niveau des tâches ou au niveau des instructions. Il peut s'agir alors d'une parallèlisation :

- De type SIMD, pour *single instruction multiple data*, où une même partie du code de l'application est exécutée sur des données différentes, par exemple la parallèlisation de l'encodage audio ;

- De type MIMD, pour *multiple instruction multiple data*, où des parties différentes du code de l'application sont exécutées en parallèle sur des données différentes, par exemple la parallèlisation d'un traitement à la chaîne.

La parallèlisation d'une application, comme exposée dans [De Lange 04] lors de [MPSoC'04], ne peut être faite d'une façon automatique que pour une petite gamme d'applications. La parallèlisation d'une application séquentielle reste donc une opération réalisée majoritairement à la main.

La parallèlisation au niveau des tâches consiste à décomposer le code séquentiel de l'application en un ensemble de tâches communicantes entre elles via l'utilisation de primitives du modèle de programmation parallèle. Elle est réalisée en trois étapes :

- L'identification des sections ou de bouts de code (fonctions) qui doivent être transformés en tâches concurrentes. Pour identifier les fonctions à parallèliser, on peut utiliser par exemple la technique de profilage[1] : on mesure le coût de chaque fonction en terme de temps d'exécution ou de nombre de cycles (en utilisant un simulateur du processeur cible). La fonction la plus coûteuse en nombre de cycles est alors candidate à être parallèlisée. Elle sera, si son code le permet, remplacée par N tâches concurrentes. Ici, il faut aussi prendre en considération le volume de la communication engendrée pour confirmer le choix de cette fonction : la parallèlisation n'est avantageuse que si le temps de communication est inférieur au temps de calcul.
- La transformation des sections de code identifiées en tâches concurrentes. Lors de cette transformation, les structures de données sont généralement modifiées.
- L'établissement de la communication entre les tâches parallèlisées en utilisant les primitives du modèle de programmation parallèle (par exemple : send/receive).

Le code est parallèlisée en utilisant les primitives de l'API du modèle de programmation parallèle.

IV.3.1.2 Transposition du logiciel applicatif parallèlisé sur une architecture multiprocesseur

La transposition du logiciel applicatif sur une architecture multiprocesseur consiste à fixer pour chaque tâche, le sous-système processeur (ou l'unité d'exécution) sur lequel elle sera exécutée. La transposition détermine les performances et le coût du système final. Pour les mesurer avec précision, il est souvent nécessaire de finir la conception. Si les performances du système conçu ne sont pas suffisantes, un second cycle de conception et de mesure de performances est nécessaire. D'où une possibilité d'un nombre élevé de cycles de conception. Pour remédier à ce problème, l'évaluation des performances doit être faite au plus tôt lors du cycle de conception. Elle peut être réalisée en utilisant des techniques d'estimation de haut niveau, par exemple « *trace based*

[1] Profilage, du terme anglais « profiling »

simulation » [Lieverse 01] ou la simulation à haut niveau [Bouchhima 04] ; ou bien à bas niveau en utilisant la co-simulation logicielle matérielle.

L'opération de transposition inclut les choix d'affectation de chaque tâche à un processeur ainsi que l'affectation de la communication entre les tâches au réseau de communication sur puce. L'espace de solution des transpositions possibles est énorme puisque sa taille est exponentielle en fonction du nombre de tâches logicielles concurrentes de l'application et du nombre des unités d'exécutions de l'architecture.

IV.3.2 **Débogage en haut niveau du logiciel applicatif parallèlisé**

Le logiciel applicatif parallèlisé est validé en comparant les résultats de son exécution avec ceux du code séquentiel sur un même ensemble de test. Si une différence existe, il est alors débogué en utilisant un modèle de simulation implémentant les primitives utilisées de l'API du modèle de programmation parallèle. A cette étape, le débogage est dit de haut niveau, car il fait abstraction des interfaces logicielles/matérielles. Par exemple, [MPICH] qui est une implémentation des primitives de l'API du modèle de programmation parallèle [MPI], peut être utilisé comme modèle de simulation pour déboguer des applications utilisant les primitives de [MPI].

IV.3.3 **Conception du logiciel dépendant du matériel par le raffinement du modèle de programmation parallèle**

Porter le code parallèlisé à une architecture multiprocesseur requiert le raffinement du modèle de programmation parallèle pour cette architecture. Le raffinement consiste à développer l'adaptation logicielle (HdS) et l'adaptation matérielle. Le HdS implémentera alors les primitives utilisées de l'API du modèle de programmation parallèle. Il peut être conçu de façon manuelle ou automatique. Sa conception peut être faite en suivant une approche logicielle ou matérielle. Toutefois, quelles que soient la méthode et l'approche utilisées pour la conception du logiciel dépendant du matériel, le concepteur doit fixer tous les paramètres de l'implémentation du HdS, comme par exemple le plan mémoire, les priorités d'interruption, la taille des piles, etc... Ces paramètres représentent alors une spécification du logiciel dépendant du matériel. Les valeurs de ces paramètres influent directement sur les performances du système MPSoC final.

Le logiciel dépendant du matériel est généralement conçu d'une façon manuelle. Cette conception manuelle est complexe, et de plus, est source d'erreurs. Certains outils commerciaux de conception au niveau système comme [System Studio], [ConvergenSC], [Platform Express], etc... aident le concepteur lors de cette phase de conception manuelle.

Afin de contrer les problèmes de la conception manuelle du logiciel dépendant du matériel, des outils ont été développés [Brunel 00][Cesario 02][Sarmento 05]. Ils permettent une conception automatisée du logiciel dépendant du matériel.

IV.3.4 Conception d'un prototype matériel

Un prototype de la partie matérielle est généralement conçu avant que le circuit matériel final ne soit prêt. Le prototype possède la même architecture et parfois la même performance que le circuit final, mais il n'est pas réalisé au même niveau d'intégration (c'est plutôt un système sur carte qu'un système monopuce). Un prototype peut aussi être obtenu par la configuration d'une plateforme de prototypage comme [ARM AP Integrator] [Virtex-II Pro Dev system].

Le prototype matériel est utilisé pour déboguer la partie logicielle ainsi que l'interface logicielle/matérielle.

IV.3.5 Débogage de l'interface logicielle/matérielle

Le concepteur a besoin de déboguer l'interface logicielle/matérielle après la conception du logiciel dépendant du matériel. Ici, on définit le terme débogage de l'interface logicielle/matérielle par le débogage du HdS et de son interaction avec le matériel.

L'étape de débogage est une étape complexe de la conception. Elle nécessite une bonne connaissance du logiciel applicatif, de l'architecture matérielle et de l'interface logicielle/matérielle. De plus, la concurrence lors de l'exécution du logiciel applicatif complique encore plus cette étape : le débogage doit être fait d'une manière non-intrusive [Beynon 02] [Moore 03], c'est à dire sans influencer le comportement global du système MPSoC.

Pour réaliser le débogage, plusieurs techniques peuvent être utilisées :

- L'utilisation d'un modèle de simulation de l'architecture matérielle : plus le modèle est fidèle à l'architecture plus il permettra de trouver un nombre plus élevé de bugs [Benini 03]. Cependant, la simulation de l'exécution de l'application sur l'architecture est très lente.
- L'utilisation du prototype matériel développé précédemment. Le prototype matériel permet une exécution plus rapide du logiciel, au prix d'une perte d'observabilité des signaux.

IV.3.6 Conception du circuit matériel final et intégration logicielle matérielle finale

Lors de la phase du débogage de l'interface logicielle/matérielle, le circuit final est en phase de conception. Une fois le circuit matériel final conçu, le logiciel et l'interface logicielle/matérielle validés sur le prototype matériel, l'intégration finale logicielle/matérielle a lieu. Le résultat de cette ultime étape est le système embarqué final. Dans cette étape, on a également, recours à des phases de test pour une validation finale. Comme la visibilité du logiciel est très limitée dans le circuit final, on utilise des techniques de test *on-line* ; c'est-à-dire, on embarque des procédures supplémentaires de test du logiciel et du matériel dans le produit final lui-même, pour tester ses fonctionnalités [Moraes 05].

Le flot proposé est basé sur l'utilisation des modèles de programmation parallèle pour l'abstraction des interfaces logicielles/matérielles, puis sur son raffinement sur l'architecture cible.

Le niveau d'abstraction du modèle de programmation parallèle étant différent d'un modèle à un autre, son choix est très important pour l'optimisation et la réutilisation du système conçu :

- Les modèles de bas niveau d'abstraction permettent une programmation efficace, optimisée, mais moins flexible et moins portable vers de nouvelles architectures matérielles.
- Les modèles de haut niveau d'abstraction permettent une programmation moins efficace et moins optimisée mais plus flexible et plus portable

Il faut donc trouver un compromis entre la visibilité (plus explicite) et l'opacité (plus abstrait) qu'offre un modèle de programmation de l'architecture matérielle.

Nous dédions la section suivante à l'étude des modèles de programmation parallèle. Le but de cette étude est de trouver un niveau d'abstraction de modèles de programmation parallèle adéquat pour la conception des systèmes MPSoC.

IV.4 Étude des modèles de programmation parallèle en vue de l'abstraction des interfaces logicielles/matérielles

Dans ce paragraphe, nous étudions les modèles de programmation parallèle en vue de l'abstraction de l'interface logicielle/matérielle. Nous classons tout d'abord ces modèles selon leurs niveaux d'abstraction. Puis, nous analysons des modèles de programmation utilisés en industrie pour la conception des systèmes MPSoC.

IV.4.1 Classification des modèles de programmation parallèle en fonction de leurs niveaux d'abstraction

Les modèles de programmation font abstraction, à des niveaux différents, de l'architecture matérielle ciblée ainsi que de l'interface logicielle/matérielle. En fait, un modèle de programmation parallèle est plus abstrait qu'un autre s'il peut cacher plus de détails d'implémentation que ce dernier. Un détail d'implémentation qui a été abstrait par un modèle de programmation parallèle est un détail que le concepteur du logiciel applicatif ne verra lors de la phase de développement du logiciel applicatif. Il est alors considéré comme un détail implicite pour le concepteur du logiciel applicatif. De même, un détail non abstrait par le modèle de programmation parallèle est un détail explicite pour le concepteur du logiciel applicatif, il peut être pris en charge par ce dernier. Finalement, un détail caché par le modèle de programmation parallèle est un détail qui sera pris en charge par le concepteur de l'interface logicielle/matérielle, responsable de l'intégration logicielle/matérielle.

Nous utiliserons les détails de l'implémentation du logiciel applicatif, pour la classification d'un modèle de programmation parallèle, tels qu'établis dans [Sckillicorn 98] :

- La **concurrence** lors de l'exécution du logiciel applicatif ;
- La **décomposition** du logiciel applicatif en des tâches concurrentes. Cette décomposition peut impliquer une modification des structures de données.

- La **transposition,** c'est à dire la décision de fixer les tâches qui seront sur la même unité de traitement et ceux qui seront sur des unités de traitement différentes. Il ne s'agit pas de fixer le type de l'unité d'exécution ciblée, mais plutôt de faire la différence entre des tâches sur une même unité d'exécution et des tâches sur des unités d'exécution différentes.
- La **communication** entre les tâches. Il s'agit de connaître pour chaque tâche, les tâches avec lesquels elle communique et d'en assurer la consistance (ne pas attendre une donnée qui n'arriva jamais).
- La **synchronisation** entre les différentes tâches concurrentes.

Classification proposée

Nous utiliserons encore la classification des modèles de programmation parallèle suivant leurs niveaux ou degrés d'abstraction tels que donnée dans [Sckillicorn 98]. Cette classification comprend les six classes décrites ci-dessous dans un ordre décroissant d'abstraction.

- **Classe 0 -** Les modèles de programmation parallèle où tous les détails sont implicites :

Ces modèles décrivent seulement le but du programme et non pas comment atteindre ce but. Les concepteurs du logiciel applicatif n'ont même pas besoin de savoir si le programme développé sera exécuté en parallèle ou non.

Ces modèles sont les plus abstraits, relativement simples d'utilisation puisque les programmes ne sont pas plus complexes que des programmes séquentiels classiques.

- Exemples : [Haskell], [PPP], [OPERA], [PALM], [P3L], etc…

- **Classe 1 -** Les modèles de programmation parallèle où seulement la concurrence est explicite ; le reste des détails sont implicites :

En utilisant ce type de modèle, le concepteur du logiciel applicatif sait qu'un parallèlisme sera utilisé. Il doit donc exprimer un potentiel au niveau de l'application pour le supporter. Toutefois, le concepteur ne sait pas jusqu'à quel degré le parallèlisme sera utilisé lors de l'exécution de son logiciel. Les modèles de cette classe ont souvent besoin que le programmeur exprime au maximum le parallèlisme présent dans l'algorithme ; puis ils réduisent le degré de parallèlisme pour s'adapter à celui offert par l'architecture ciblée. Ces modèles gèrent alors les implications de cette adaptation sur la transposition, la communication et la synchronisation.

- Exemples : [Sisal], [Concurrent Prolog], [Modula3], [Patrallel sets], [Dataparallel C*], etc…

- **Classe 2 -** Les modèles de programmation parallèle où seulement la concurrence et la décomposition sont explicites ; et où la transposition, la communication et la synchronisation sont implicites :

En plus des informations nécessaires aux modèles de la classe 1, les modèles de la classe 2 ont besoin d'informations sur le découpage du programme initial (séquentiel) en sous-programmes

concurrents. Ils prennent, cependant, en charge les implications de la décomposition pour assurer une exécution correcte du programme.

- Exemples : [CORBA], DSOC/SMP, [BSP], [LogP], etc...

Classe 3 - Les modèles de programmation parallèle où la concurrence, la décomposition et la transposition sont explicites ; et où la communication et la synchronisation sont implicites :

En utilisant un modèle de programmation parallèle de la classe 3, le concepteur de l'application doit décider de la décomposition de son application en sous-programmes concurrents et doit aussi décider de la meilleure affectation de chaque sous-programme à une unité d'exécution (la meilleure transposition).

Cependant, trouver la meilleure transposition est une des principales difficultés de la programmation parallèle. Il faut maîtriser le compromis entre la distribution de la charge sur les différentes unités de traitement et les effets de cette distribution sur la performance globale du système. De plus, une maîtrise de la relation entre la vitesse de calcul des processeurs et celle des réseaux de communication est nécessaire pour obtenir une meilleure performance du système, aussi, il faut prendre en compte la performance de l'architecture mémoire.

Déjà à ce niveau, la portabilité de l'application développée avec cette classe de modèles vers de nouvelles architectures est fortement compromise : les décisions de transposition étant motivées par les caractéristiques de l'architecture matérielle ciblée.

- Exemples : [Compositional C++], [ISETL-Linda], [Opus], etc...

Classe 4 - Les modèles de programmation parallèle où la concurrence, la décomposition, la transposition et la communication sont explicites ; et où la synchronisation est implicite ;

Ici, le concepteur doit gérer la plupart des détails relatifs à l'implémentation du parallèlisme, exception faite de la synchronisation qui reste encore gérée implicitement. Cette gestion est généralement associée à une sémantique asynchrone : des messages sont émis mais l'émetteur ne sait pas quand ils sont reçu, de plus, la réception de plusieurs messages peut être dans un ordre différent de celui de l'envoi.

- Exemples : [TTL], [Actors], [POOL-T], [Argus], [Alpha], etc...

Classe 5 - Les modèles de programmation parallèle où tous les détails sont explicites

Ici, tous les détails de l'implémentation du parallèlisme sont gérés par le concepteur. La conception d'une application hautement parallèle est difficile en utilisant cette classe de modèles. Cependant, même si avec ces modèles tous les détails sont implicites, nous les considérons toujours comme des modèles de programmation de haut niveau. En fait, ils font toujours abstraction de l'interface logicielle/matérielle et de l'architecture matérielle.

- Exemples : [PVM], [MPI], [JAVA], [Concurent C], [Occam], [TLM Transfert], [FORK], etc...

Récapitulatif

La figure IV-2 représente la classification proposée. Les détails d'implémentation utilisés pour la classification y sont présentés sur un axe d'abstraction. Les classes de modèles de programmation parallèle sont représentées par les intervalles entre les différents détails (seuls les classes 2, 3 et 5 ont été représentées sur cette figure pour assurer la lisibilité). Plus une classe est à gauche, plus son niveau d'abstraction est élevé.

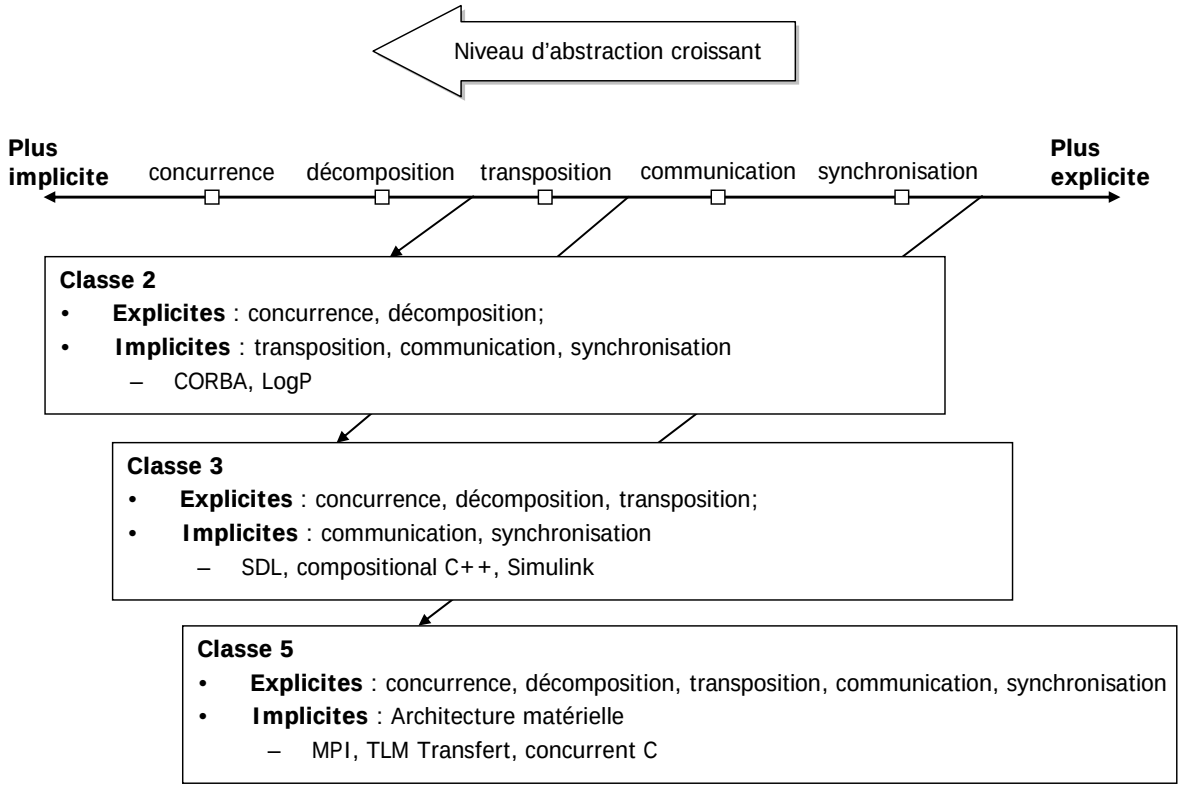


Figure IV-2 : Classement des modèles de programmation parallèle par niveau d'abstraction

Nous pensons que pour être efficace sur une architecture matérielle hétérogène et hautement parallèle, le logiciel applicatif doit être pensé et conçu comme un logiciel parallèle dès le début et non plus comme un logiciel séquentiel qui sera parallèlisé par des outils. Les modèles des classes 2, 3, 4 et 5 semblent alors être les plus adéquats pour la conception des systèmes MPSoC. Tous laissent à la charge du concepteur, l'expression de la concurrence et la décomposition du logiciel applicatif sous forme de tâches concurrentes.

IV.4.2 Exemples de modèles de programmation parallèle pour la conception des systèmes MPSoC

L'utilisation des modèles de programmation parallèle pour la conception des MPSoC a pour but l'abstraction des interfaces logicielles/matérielles. Le raffinement du modèle de programmation parallèle utilisé sur l'architecture ciblée conduit au développement de l'adaptation logicielle (HdS)

et/ou de l'adaptation matérielle. Pour le développement du HdS, les approches logicielle ou matérielle peuvent être utilisées (voir paragraphe III.2).

Dans ce paragraphe, nous présentons une étude de quelques modèles de programmation parallèle. Les deux premiers sont utilisés en industrie, pour la conception de systèmes MPSoC. Le premier est le modèle DSOC/SMP, proposé par STMicroelectronics. Ce modèle suit l'approche matérielle. Le second est le modèle TTL, proposé par Philips. Il aborde l'approche logicielle. Puis nous présenterons les modèles MPI et CORBA qui sont des modèles de programmation qui suivent l'approche logicielle. Ces modèles sont généralement utilisés pour la conception des systèmes multiprocesseurs classiques, nous étudierons alors leur utilisation dans le contexte de la conception des MPSoC.

IV.4.2.1 Le modèle DSOC/SMP pour la plateforme StepNP, par STMicroelectronics

Le modèle DSOC/SMP [Paulin 04-A][Paulin 04-B] suit l'approche matérielle. Nous rappelons que cette approche consiste à développer, tout d'abord, une plateforme matérielle et à l'abstraire, ensuite, en utilisant un modèle de programmation parallèle. Le modèle DSOC/SMP (constitué en réalité, de deux modèles qui sont DSOC et SMP) a été développé par STMicroelectronics, pour abstraire la plateforme multiprocesseur StepNP [Paulin 02].

Le modèle DSOC/SMP fait partie de la classe 2 car la concurrence et la décomposition y sont explicites (l'application étant décomposée dans ce modèle sous forme d'objets communicants).

Dans la suite, nous présenterons brièvement la plateforme StepNP, puis nous détaillerons les modèles DSOC et SMP.

a. La plateforme StepNP

La plateforme StepNP, représentée dans la figure IV-3, inclut des modèles de :

- processeurs hétérogènes standard ou reconfigurables ;
- réseau sur puce ;
- éléments hétérogènes configurables de traitement matériel ;
- entrées/sorties spécifiques au réseau.

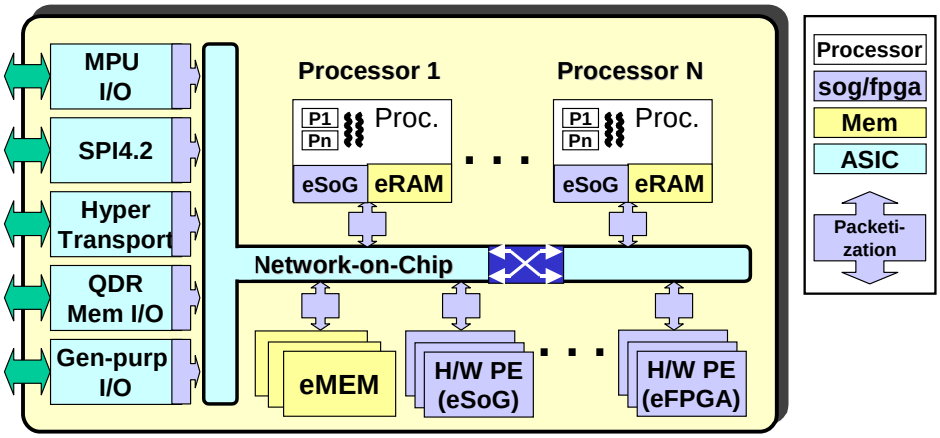


Figure IV-3 : Plateforme StepNP
(figure extraite de [Paulin 04 - A])

b. Présentation du modèle DSOC

Le modèle DSOC, pour *distributed system object component*, s'inspire des modèles [CORBA] et [DCOM]. Il supporte le calcul distribué hétérogène où des objets distribués communiquent par échange de messages. Chaque objet possède une interface décrite en un langage de définition d'interfaces appelé SIDL [Paulin 02] (SIDL, pour *systemc interface description language*). A travers son interface un objet peut requérir ou offrir des services. Les objets peuvent être ciblés sur les diverses unités d'exécution : sur un processeur générique équipé d'un système d'exploitation standard, sur un processeur matériellement multiprocessus, ou sur une unité de traitement matérielle.

Lors de l'exécution de l'application sur la plateforme StepNP, les différents objets communiquent par échange de messages. Une étape de conversion des données échangées vers un format intermédiaire est requise pour permettre la communication entre les différentes unités d'exécution hétérogènes de la plateforme.

L'exécution en parallèle d'une application dans le modèle DSOC peut être réalisée en utilisant un ou plusieurs mécanisme(s) parmi les suivants :

- Plusieurs objets clients peuvent s'exécuter en parallèle.
- La charge de traitement d'un service peut être distribuée sur plusieurs ressources.
- Une requête de service d'un client peut retourner avant que le serveur ne termine son traitement, afin de permettre l'exécution parallèle du client et du serveur.

La figure IV-4 ci-dessous, illustre le modèle de programmation de DSOC. Les quatre objets représentent des fonctions de l'application. Ces objets peuvent être assignés à un processeur standard exploité par un système d'exploitation standard (par exemple pour l'objet2), à un processeur matériellement multi processus (objets 1 et 3), ou à une unité de traitement matérielle (objet4). La communication entre les objets peut être faite en logiciel (entre objet 1 et objet 2) ou d'une façon mixte, logicielle/matérielle (par exemple la communication entre l'objet 2 et l'objet 4). On remarque la présence d'adaptateurs pour l'échange de messages entre chaque composant

matériel de la plateforme et le réseau de communication. Il s'agit d'accélérateurs matériels d'échanges de messages que nous détaillerons par la suite.

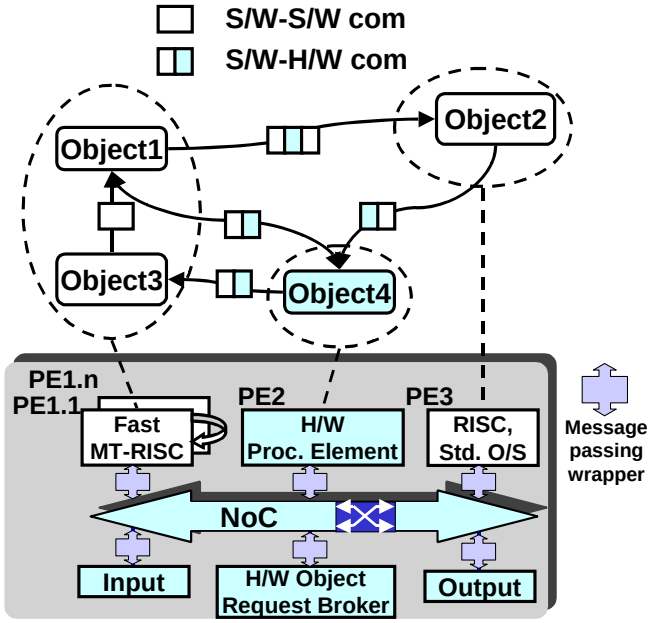


Figure IV-4 : Du modèle DSOC à la transposition sur la plateforme (figure extraite de [Paulin 04 - A])

C. Implémentation du modèle DSOC sur la plateforme StepNP

L'implémentation de DSOC sur la plateforme StepNP se base sur trois services principaux. Pour des besoins en performance, un choix principal de conception est l'exécution de ces services en matériel. Ces services sont :

- (1) L'accélérateur matériel d'échange de messages, noté MPE pour *message passing engine*. Il est utilisé pour optimiser les communications inter processus. Il convertit un message sortant en une représentation portable formatée pour une transmission sur le réseau. Il fournit la fonctionnalité inverse du coté récepteur.

Le compilateur du langage SIDL gère cette conversion en générant, pour un objet implémenté en logiciel, le code logiciel de bas niveau qui pilote le MPE. Le compilateur génère aussi, pour un objet implémenté sur un composant matériel, le matériel qui assure cette conversion et l'interface avec le réseau de communication.

L'échange de messages étant supporté matériellement, le surcoût logiciel d'une invocation de service distant est de l'ordre d'une douzaine d'instructions seulement. Il n'y a pas de changement de contexte ; si l'objet serveur doit retourner un résultat, le processus client est figé jusqu'à ce que le résultat soit prêt.

- (2) Le courtier matériel des requêtes sur les objets, noté ORB pour *object request borker*. Il est utilisé pour coordonner la communication entre les objets.

La figure IV-5 illustre un exemple d'une plateforme d'exécution du modèle de programmation parallèle DSOC. La partie supérieure de la figure décrit un ensemble d'objets DSOC ciblés sur un ou plusieurs processeurs matériellement multi processus (comme représenté du côté supérieur gauche). La figure inclut également un certain nombre d'objets DSOC transposés sur des composants matériels (montrés dans le côté supérieur droit). L'ORB est représenté en bas de la figure.

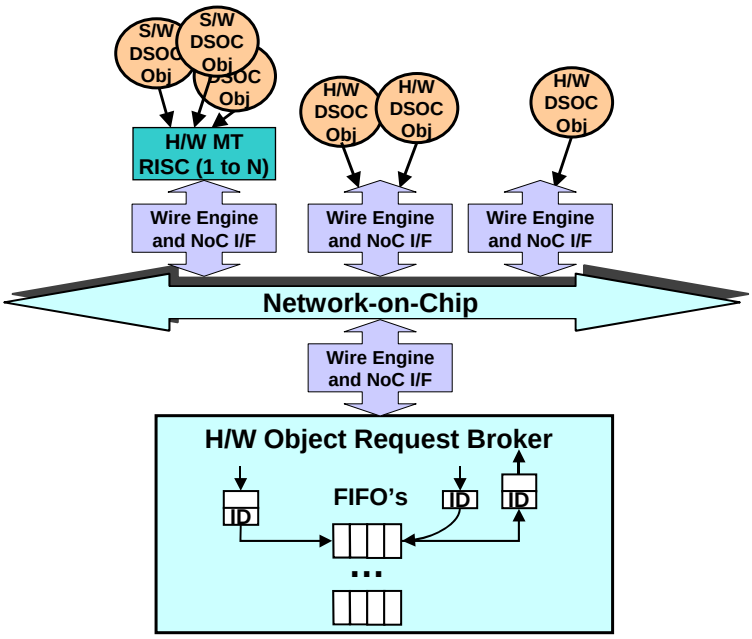


Figure IV-5 : Exemple d'une plateforme d'exécution pour le modèle de programmation parallèle DSOC (figure extraite de [Paulin 05])

L'approche utilisée pour l'implémentation du modèle DSOC implique la duplication des objets serveurs sur les différentes unités d'exécution de l'architecture. Le courtier de demande d'objet (ORB) est responsable de la mise en correspondance des demandes de services des clients avec les serveurs adéquats, suivant certains critères. Dans sa version actuelle, c'est le serveur le moins chargé qui est choisi.

- (3) La gestion en matériel des processus : La gestion et la coordination de l'exécution des processus sont effectuées en matériel. Tous les processus logiques de l'application sont directement ciblés sur des processus matériels des processeurs matériellement multi processus. Il n'y a pas un multiplexage entre les processus logiciels et les processus matériels. La gestion globale des processus est assurée par l'ORB et chaque processeur matériellement multi processus gère lui-même ses propres processus actifs.

Le raffinement du modèle DSOC sur la plateforme StepNP utilise les composants matériels MPE et ORB. Par rapport à la terminologie détaillée dans le second chapitre, nous considérons ces composants matériels comme l'adaptation matérielle issue du raffinement du modèle DSOC. De

même, nous considérons comme adaptation logicielle (HdS), le code logiciel de bas niveau qui commande le MPE.

d. Présentation du modèle SMP

Le modèle SMP, pour *symetric multi-processing*, suit aussi l'approche matérielle. Ce modèle supporte des processus concurrents accédant à une mémoire partagée. Le concept de programmation SMP est semblable à celui de [Java] ou de [Microsoft C#]. L'implémentation de ce modèle de programmation prend en charge l'ordonnancement basé sur les priorités, avec le support des processus et des mécanismes de programmation comme les conditions et les sémaphores.

e. Implémentation du modèle SMP sur la plateforme StepNP

La fonctionnalité SMP est implémentée par la combinaison d'une fine couche logicielle et d'un moteur matériel de gestion de concurrence, noté CE pour *concurency engine*. Le CE est responsable de la gestion de la pile des processus en exécution, la balance de charge, etc… Sous le modèle SMP, il n'y a pas de changement de contexte puisque les processeurs sont matériellement multi processus. Un processus bloqué est simplement suspendu jusqu'à son déblocage (libération d'un sémaphore, activation d'une condition, etc…).

Le CE apparaît aux processeurs comme un composant transposé en mémoire qui contrôle matériellement un nombre d'objets concurrents (figure IV-6). Par exemple, un segment d'adresses inclus dans la plage d'adresses gérées par le CE peut correspondre à un moniteur. Les applications sur le moniteur sont réalisées par des opérations d'écriture et de lecture sur ce segment d'adresses. Chaque lecture ou écriture dans une des adresses associées au CE est équivalente à une commande bien déterminée et préconfigurée du CE.

Par rapport à la terminologie détaillée dans le second chapitre, nous considérons le composant matériel CE comme l'adaptation matérielle issue du raffinement du modèle SMP sur l'architecture StepNP. L'adaptation logicielle (HdS) est alors la fine couche logicielle qui pilote le CE.

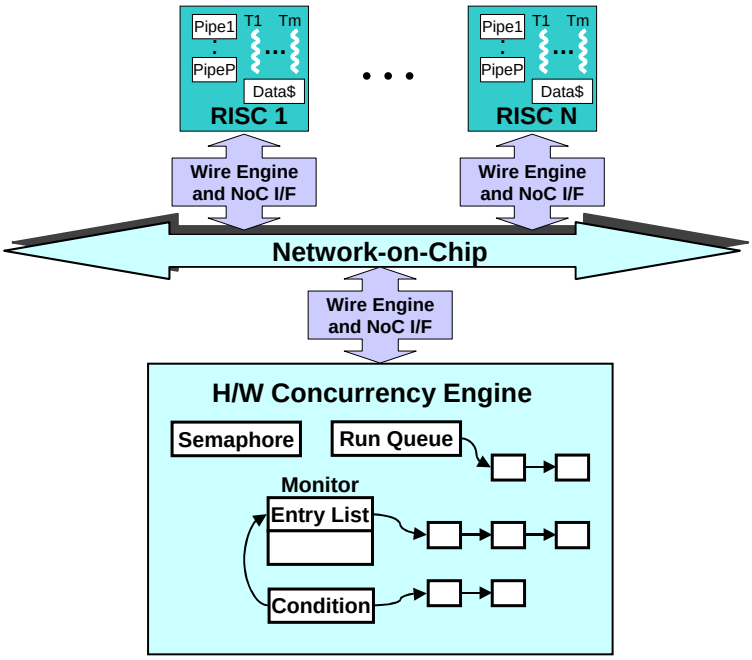


Figure IV-6 : Plateforme d'exécution pour le modèle SMP (figure extraite de [Paulin 05])

f. Évaluation du modèle DSOC/SMP

DSOC/SMP est un modèle de programmation qui abstrait la plateforme StepNP. Il offre l'avantage d'un passage systématique d'une programmation en haut niveau, basée sur le langage SIDL, vers une implémentation optimisée sur la plateforme matérielle propriétaire StepNP.

Des expériences de conception à base du modèle DSOC/SMP et de la plateforme StepNP relatent des performances suffisantes pour des applications comme la gestion du trafic sur le réseau [Paulin 04]. Les performances de ce modèle sont dues à son adéquation avec l'architecture matérielle sur laquelle il s'exécute. En effet, le choix d'implémenter majoritairement l'interface logicielle/matérielle en matériel sur la plateforme influe directement et positivement sur les performances : les différents moteurs matériels de l'architecture StepNP permettent de limiter le coût (1) de la communication par échange de messages grâce au MPE, (2) de la gestion de la programmation à base de mémoire partagée grâce au CE et (3) du nombre de changements de contexte grâce, notamment, à l'utilisation de processeurs matériellement multi processus.

Le revers de la médaille du choix d'une implémentation majoritairement matérielle de l'interface logicielle/matérielle est la perte de portabilité des applications développées : le modèle DSOC/SMP ne cible – d'une façon automatisée – que la plateforme StepNP. Le changement de plateforme implique la conception manuelle de l'adaptation logicielle et de l'adaptation matérielle.

IV.4.2.2 Le modèle TTL, par Philips

Le modèle de programmation parallèle TTL, pour *task transaction level*, ou transaction au niveau tâches suit l'approche logicielle. Nous rappelons que cette approche consiste à développer tout d'abord un modèle de programmation indépendant de toute architecture matérielle et à le raffiner ensuite vers l'architecture ciblée. Le modèle TTL a été développé chez Philips [Van der Wolf 04] [Van Der Wolf 05]. Il est adapté aux applications de type flux de données, telle que la diffusion de vidéo en continu (*streaming*). Le but de ce modèle est de permettre la conception et la programmation des systèmes MPSoC en utilisant une approche basée sur des transactions au niveau des tâches. Puis d'optimiser le raffinement des primitives TTL utilisées par l'application sur l'architecture matérielle ciblée, en utilisant des techniques de transformation de code.

Le modèle de programmation parallèle TTL permet une gestion explicite de la concurrence et de la communication. Il appartient donc à la classe 4.

La sémantique de ce modèle de programmation est la suivante : les tâches communiquent en utilisant un canal de données. Une tâche accède au canal à travers un port. Un port est associé à un seul canal. La tâche accède alors au canal en appelant des primitives de l'API TTL sur le port qui lui est associé. Un canal contient un certain nombre fixe et fini de jetons. Un jeton est soit libre (vide) soit rempli de données. Le canal est unidirectionnel. Il assure une transmission ordonnée ou non ordonnée des données, via les jetons. Le type de données transmises via le canal est arbitraire mais unique pour chaque canal. Finalement, la seule communication multipoint possible est avec un unique émetteur pour *n* récepteurs.

Pour communiquer, un émetteur doit acquérir des jetons libres du canal et y écrire les données qu'il veut envoyer. Le récepteur de son coté, doit récupérer des jetons remplis, les lire et enfin les libérer. La figure IV-7 représente le modèle logique de communication en TTL.

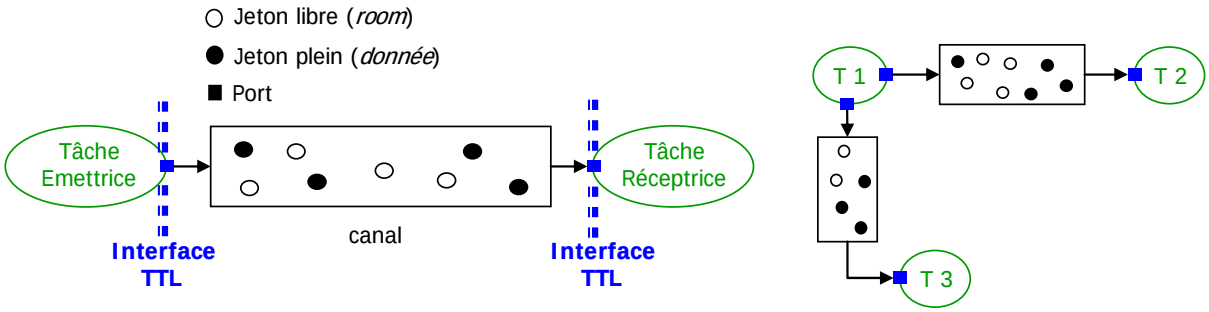


Figure IV-7 : Modèle logique de communication en TTL

a. Interfaces et APIs du modèle TTL

TTL définit sept différents types d'interfaces de communication pour supporter des styles différents de communication. Les sept types sont basés sur le même modèle logique de communication, ce qui permet l'utilisation de différents types lors d'une même conception.

Chaque interface TTL offre des primitives différentes avec des sémantiques différentes. Dans la suite nous présenterons les sept interfaces TTL. Pour chaque interface, nous détaillerons les primitives offertes et discutons de ses avantages et ses limitations.

- Interface CB, pour *combined bloking*
  - Présentation de l'interface CB :

L'interface CB est l'unique interface qui combine le transfert de données (sous forme de vecteurs) et la synchronisation. C'est une interface bloquante qui ne retourne que quand l'opération de lecture ou d'écriture est achevée. C'est aussi l'interface la plus abstraite de TTL.

- Primitives offertes par l'interface CB :

Côté émetteur :

```
- Write (port, vector, size) ;
```

Permet d'écrire le vecteur `vector` de taille `size` dans le canal connecté au port `port`.

Côté récepteur :

```
- Read (port, vector, size) ;
```

Permet de lire le vecteur `vector` de taille `size` à partir du canal connecté au port `port`.

- Avantages de l'interface CB :

Cette interface est facile d'utilisation.

- Limitations de l'interface CB :

L'interface CB présente un surcoût de copie si les variables à transférer ne se trouvent pas dans les buffers locaux de transfert (une copie de l'emplacement physique de la variable vers l'emplacement physique du buffer de transfert). De plus, l'utilisation de buffers locaux peut engendrer un besoin important en mémoire lorsqu'un grand nombre de jetons est utilisé ou bien un surcoût de synchronisation lorsque des buffers de tailles réduites sont utilisés.

Les interfaces suivantes séparent, en offrant des primitives différentes, le transfert des données de la synchronisation.

- Interfaces RB et RN, pour respectivement *relative bloking* et *relative non-bloking*
  - Présentation des interfaces RB/RN :

Les primitives RB et RN séparent la synchronisation du transfert des données. Elles agissent sur des vecteurs. Les adresses des jetons sont cachées pour l'application.

- Primitives offertes par les interfaces RB/RN :

Côté émetteur :

```
- reAcquireRoom (port, count); //interface RB
```

Acquisition bloquante de `count` jetons vides du canal connecté au port `port`. Attend que les jetons soient disponibles en nombre suffisant avant de retourner.

```
- tryReAcquireRoom (port, count); //interface RN
```

Acquisition non bloquante de `count` jetons vides du canal connecté au port `port`. Retourne immédiatement un succès ou un échec d'acquisition.

- `store (port, offset, vector, size);`

Copie le vecteur `vector` de taille `size` dans les jetons à partir de l'offset `offset`.

- `releaseData (port, count) ;`

Libère les `count` plus anciens jetons acquis dans le canal. Les jetons sont pleins et sont destinés à être lus ensuite par le récepteur.

Côté récepteur :

- `reAcquireData (port, count);`

Acquisition bloquante de `count` jetons pleins du canal connecté au port `port`. Attend que les jetons soient disponibles en nombre suffisant avant de retourner.

- `tryReAcquireData (port, count);`

Acquisition non bloquante de `count` jetons pleins du canal connecté au port `port`. Retourne immédiatement un succès ou un échec d'acquisition.

- `load (port, offset, vector, size);`

Copie le vecteur `vector` de taille `size` à partir des jetons acquis. Peut être en désordre en utilisant l'offset `offset`.

- `releaseRoom (port, count) ;`

Libère les `count` plus anciens jetons acquis dans le canal. Les jetons sont vides et sont destinés à être écrits ensuite par l'émetteur.

- Avantages des interfaces RB/RN

Ces interfaces offrent une synchronisation à gros grains pour un transfert de données à fins grains, d'où un surcoût de synchronisation limité. Aussi, grâce à l'offset, un accès non ordonné aux données est possible pour permettre de réduire le coût de l'utilisation des variables privées. Finalement, les interfaces RB/RN permettent de lire seulement une partie des jetons qui constituent un vecteur de données réduisant ainsi le coût du transfert de données.

- Limitations des interfaces RB/RN

Les interfaces RB/RN sont moins abstraites que l'interface CB. Elles requièrent alors un effort de programmation plus important et rendent les tâches moins réutilisables. De plus, le transfert de données n'est pas très efficace (appel de fonction de l'interface, accès à l'administrateur du canal et finalement calcul de l'adresse). Ainsi, un surcoût de copie pourrait toujours avoir lieu.

- Interfaces DBI et DNI pour respectivement direct blocking in-order et direct non-blocking in-order

- Présentation des interfaces DBI et DNI :

Ces deux interfaces se caractérisent par un accès direct aux données en utilisant des références (des pointeurs) sur les jetons. Ces derniers sont libérés dans le même ordre de leurs acquisitions (d'où le terme in-order). Seules des opérations sur des données de type scalaire sont possibles.

- Primitives offertes par les interfaces DBI et DNI :

Côté émetteur :

`- acquireRoom (port, &token); //interface DBI`

Retourne une référence (`&token`) sur le jeton vide acquis du canal. C'est une primitive bloquante.

`- tryAcquireRoom (port, &token); //interface DNI`

Retourne une référence (`&token`) sur le jeton vide acquis du canal. C'est une primitive non bloquante.

`- token->filed = value;`

Met à jour la valeur du jeton `token` avec la valeur `valeur`. Opération d'écriture.

`- releaseData (port)`

Libère le jeton le plus ancien comme jeton plein sur le canal connecté au port `port`. L'ordre de libération est pris en compte, le plus ancien d'abord.

Côté récepteur :

`- acquireData (port, &token); //interface DBI`

Retourne une référence (`&token`) sur le jeton plein acquis du canal. C'est une primitive bloquante.

`- tryAcquireData (port, &token); //interface DNI`

Retourne une référence (`&token`) sur le jeton plein acquis du canal. C'est une primitive non bloquante.

`- value = token->filed;`

Met à jour la valeur `valeur` avec la valeur du jeton `token`. Opération de lecture.

`- releaseRoom (port) ;`

Libère le jeton le plus ancien comme jeton vide sur le canal connecté au port `port`. L'ordre de libération est pris en compte, le plus ancien d'abord.

- Avantages des interfaces DBI et DNI :

Les interfaces DBI/DNI présentent l'avantage d'une synchronisation à gros grain pour un transfert de données à grains fins. Le coût de la synchronisation est donc diminué. Elles offrent aussi les possibilités d'accès non ordonnés aux jetons acquis et du chargement partiel du contenu d'un jeton. Finalement, elles apportent la possibilité de l'accès direct aux données pour un transfert de données plus efficace.

- Limitations des interfaces DBI et DNI

Les interfaces DBI/DNI sont moins abstraites que les trois précédentes (CB / RB / RN) car elles explicitent les adresses mémoire et rendent donc les tâches moins réutilisables. De plus, elles ne gèrent pas les vecteurs de données.

■ Interfaces DBO et DNO pour respectivement direct blocking out-of-order et direct non-blocking out-of-order

- Présentation des interfaces DBO et DNO :

Les interfaces DBO et DNO se différencient des interfaces DBI et DNI uniquement par la possibilité de libération des jetons dans un ordre différent de celui de leur acquisition.

- Primitives offertes par les interfaces DBO et DNO :

Côté émetteur :

`- acquireRoom (port, &token);`

Retourne une référence (`&token`) sur le jeton vide acquis du canal à partir du port `port`. C'est une primitive bloquante.

`- tryAcquireRoom (port, &token);`

Retourne une référence (`&token`) sur le jeton vide acquis du canal à partir du port `port`. C'est une primitive non bloquante.

`- token->filed = value;`

Met à jour la valeur du jeton `token` avec la valeur `valeur`. Opération d'écriture.

`- releaseData (port, &token)`

Libère le jeton comme jeton plein sur le canal connecté au port `port`. Il n'y a pas un ordre de libération.

Côté récepteur :

`- acquireData (port, &token);`

Retourne une référence (`&token`) sur le jeton plein acquis du canal. C'est une primitive bloquante.

`- tryAcquireData (port, &token);`

Retourne une référence (`&token`) sur le jeton plein acquis du canal. C'est une primitive non bloquante.

`- value = token->filed;`

Met à jour la valeur `valeur` avec la valeur du jeton `token`. Opération de lecture.

`- releaseRoom (port, &token)`

Libère le jeton comme jeton vide sur le canal connecté au port `port`. Il n'y a pas d'ordre de libération.

- Avantages des interfaces DBO et DNO :

Ces interfaces possèdent les mêmes avantages que DBI et DNI, en plus d'une meilleure gestion de la mémoire.

- Limitations des interfaces DBO et DNO

Ces interfaces possèdent les mêmes limitations que DBI et DNI, en plus d'une implémentation plus complexe du canal.

Le tableau suivant récapitule les primitives des interfaces TTL.

Type de l'interface	Primitives de l'interface	
	émetteur :	**récepteur :**
CB: *Combined Blocking*	`Write (port, vector, size);`	`Read (port, vector, size);`

RB:
Relative Blocking
RN:
Relative Non-blocking
	`reAcquireRoom (port, count);`
	`tryReAcquireRoom (port, count);`
	`store (port, offset, vector, size);`
	`releaseData (port, count);`
		`reAcquireData (port, count);`
		`tryReAcquireData (port, count); load (port, offset, vector, size);`
		`releaseRoom (port, count);`

DBI:
Direct Blocking In-order
DNI:
Direct Non-blocking In-order
	`acquireRoom (port, &token);`
	`tryAcquireRoom (port, &token);`
	`token->filed=value;`
	`releaseData (port);`
		`acquireData (port, &token);`
		`tryAcquireData (port, &token);`
		`value=token->filed;`
		`releaseRoom (port);`

DBO:
Direct Blocking Out-of-order
DNO:
Direct Non-blocking Out-of-order
	`acquireRoom (port, &token);`
	`tryAcquireRoom (port, &token);`
	`token->filed=value;`
	`releaseData (port, &token);`
		`acquireData (port, &token);`
		`tryAcquireData (port, &token);`
		`value=token->filed;`
		`releaseRoom (port, &token);`

Tableau IV-1 : Récapitulatif des types d'interfaces TTL et de leurs primitives

- Outre les sept interfaces précédentes relatives à la communication, TTL définit aussi une huitième interface pour la gestion des tâches. Cette interface offre différentes possibilités d'interactions avec l'ordonnanceur. Elle supporte les trois types de tâches suivants : « process », « co-routine » et « actor ».

Une tâche de type « process » possède son propre *thread* d'exécution et n'interagit pas d'une façon explicite avec l'ordonnanceur. La sauvegarde et le changement de contexte sont alors implicites.

Une tâche de type « co-routine » interagit explicitement avec l'ordonnanceur par la primitive `suspend()`. Cette primitive permet de réduire le coût du changement de contexte : elle autorise la tâche à se suspendre quand elle a moins de contexte à sauvegarder. La sauvegarde du contexte reste tout de même implicite.

Une tâche de type « actor » exécute un calcul fini puis retourne à l'ordonnanceur (« fire-exit »), elle prend en charge la sauvegarde de son contexte.

b. Implémentation des interfaces TTL sur des architectures matérielles

Nous avons passé en revue les différentes interfaces offertes par TTL. Ces interfaces, grâce aux primitives qu'elles supportent, offrent un modèle de programmation parallèle qui permet de concevoir le logiciel applicatif. Une fois l'application conçue, l'étape suivante est l'implémentation d'une façon optimisée des primitives de l'API TTL utilisées sur l'architecture matérielle ciblée.

Il y a deux critères pour choisir le type de l'interface qui sera utilisée pour développer une application donnée sur une architecture donnée. Tout d'abord, le type de l'interface doit correspondre aux caractéristiques de l'application (par exemple, pour les applications audio, les interfaces DBO et DNO ne sont pas adéquates, car ces applications n'ont pas besoin de consommer ou de produire des données en désordre).

Dans l'approche TTL, une étape de réécriture du code source de l'application est effectuée pour améliorer l'efficacité de la conception. Cette transformation de code aura lieu uniquement quand une architecture multiprocesseur est ciblée. Elle a pour but de réduire le coût de l'utilisation de la mémoire, le nombre de cycles requis pour la synchronisation, le transfert de données et la génération des adresses. Il ne s'agit pas de transformation dans l'algorithme de l'application mais d'une optimisation de l'utilisation des primitives TTL en vue de leur implémentation sur l'architecture matérielle.

L'implémentation des primitives de l'API sur l'architecture est réalisée actuellement de façon manuelle [Van Der Wolf 04]. Des travaux sont en cours pour aboutir à une implémentation automatique de ces primitives [Van Der Wolf 04] et [Van Der Wolf 05]. D'une manière générale, l'implémentation d'un canal TTL se décompose en deux parties : l'implémentation du buffer du canal et l'administration du canal. L'implémentation du buffer du canal dépend de la configuration de la mémoire dans l'architecture matérielle ciblée, sa version la plus simple est une FIFO circulaire. L'administration du canal gère les jetons du canal. Elle enregistre le nombre total de jetons dans le canal, le nombre de jetons vides et celui de jetons pleins. Elle fournit également le moyen pour acquérir le prochain jeton vide ou plein à partir du canal.

Les primitives TTL peuvent aussi être implémentées en matériel. Cette implémentation est aussi manuelle, elle se fait au cas par cas et dépend des spécificités de l'architecture ciblée (le type et le protocole du réseau de communication).

c. Évaluation du modèle TTL

Le modèle TTL abstrait l'architecture matérielle. Le niveau d'abstraction de la majorité des primitives de l'API des interfaces (à part ceux de l'interface CB) est assez bas : la conception d'applications complexes est plus difficile et les applications conçues sont moins portables. Toutefois, les interfaces du modèle TTL peuvent servir de fondations pour la conception de modèles de programmation plus abstraits. De plus, le modèle TTL est limité aux applications de type flot de données. Une étape de modification du code du logiciel applicatif est nécessaire pour optimiser les aspects relatifs à la communication et la synchronisation et pour améliorer ainsi les performances lors de son exécution.

Actuellement, l'implémentation des primitives TTL est manuelle. L'adaptation automatique proposée comme perspective dans [Van Der Wolf 05], du fait qu'elle passe par une étape d'optimisation où le code source est modifié, risque de compliquer davantage la phase de débogage du système entier.

IV.4.2.3 Le modèle de programmation parallèle MPI

Le modèle de programmation parallèle [MPI] suit l'approche logicielle. Il a été développé pour permettre la programmation parallèle en utilisant une bibliothèque standard de communication. Avant son développement, de nombreuses bibliothèques ont été développées avec chacune des syntaxes différentes pour des sémantiques bien souvent similaires.

MPI signifie *message passing interface*. C'est un standard de bibliothèque de communication développé pour permettre la programmation parallèle à l'aide de processus coopératifs. Le standard MPI existe en deux versions : MPI 1 et MPI 2. MPI 1 permet la communication par passage de messages traditionnel alors que MPI 2 permet en plus l'accès à des mémoires distantes, la parallèlisation des entrées sorties et les processus dynamiques. Dans notre cas, nous nous limitons à l'étude de MPI 1 qui suffit à nos besoins.

MPI est un modèle de programmation parallèle qui appartient à la classe 5, car tous les détails de la gestion du parallèlisme sont à la charge du concepteur du logiciel applicatif. Lorsque deux processus communiquent, l'un demande explicitement l'envoi du message et l'autre demande explicitement sa réception.

a. Interface et API du modèle MPI

L'API du modèle de programmation MPI regroupe 125 [primitives MPI] permettant la communication et la synchronisation entre les processus d'une application. Les processus sont coopératifs et les données sont échangées sous forme de messages. L'envoi et la réception des messages se fait à travers l'utilisation de primitives MPI, le choix de la primitive influe directement sur la synchronisation.

Sélection d'un sous ensemble de primitives MPI pour la conception des systèmes MPSoC

Nous proposons un sous ensemble de 21 primitives MPI (parmi les 125) pour la conception des systèmes embraqués. Ces primitives sont :

- Les primitives de communications point à point bloquantes présentées dans le tableau IV-2.
- Les primitives de communications point à point non bloquantes présentées dans le tableau IV-3.
- Les primitives de communications multipoints présentées dans le tableau IV-4.
- Les primitives de contrôle (`MPI_INIT`, `MPI_FINALIZE`, `MPI_COMM_SIZE`, `MPI_COMM_RANK`).

Nous pensons que cet ensemble est suffisamment général pour supporter plusieurs types d'applications différentes. Nous proposons aussi pour la même raison que seuls les types de données prédéfinis soient supportés par ces primitives.

Dans la suite, nous détaillerons les primitives de communication, les primitives de contrôle et les types de données prédéfinis.

Primitives de communication

Dans le modèle MPI, la communication peut être de type point-à-point ou multipoints.

- Communication point-à-point

La communication point-à-point peut être bloquante ou non bloquante. Les communications bloquantes comprennent 4 primitives pour l'envoi de données et 1 pour la réception des données (tableau IV-2). La primitive `MPI_RECV` est utilisable pour réceptionner les messages envoyés par n'importe quelle primitive d'envoi bloquant.

Envoi de messages	Réception de messages
`MPI_SEND` : Envoi standard bloquant `MPI_BSEND` : Envoi bufférisé bloquant `MPI_RSEND` : *« ready send »* bloquant `MPI_SSEND` : Envoi synchrone bloquant	`MPI_RECV` : Réception standard bloquante

Tableau IV-2 : Primitives bloquantes de communication point à point

Nous présentons plus en détail les primitives `MPI_SEND` et `MPI_RECV` dans l'annexe B de ce mémoire.

Les communications non bloquantes comprennent aussi 4 primitives pour l'envoi de données et 1 pour la réception de données (tableau ci-dessous). Quatre autres primitives permettent de gérer les communications pendantes.

Envoi de messages	Réception de messages	Synchronisation explicite et renseignement
`MPI_ISEND` : Envoi standard non bloquant `MPI_IBSEND` : Envoi bufférisé non bloquant `MPI_IRSEND` : *« ready send »* non bloquant `MPI_ISSEND` : Envoi synchrone non bloquant	`MPI_IRECV` : Réception standard non bloquante	`MPI_WAIT` : Attend l'achèvement d'une communication `MPI_TEST` : Test l'état d'une communication `MPI_PROBE` : Permet de savoir s'il y a des communications pendantes `MPI_CANCEL` : Permet de désactiver une communication

Tableau IV-3 : Primitives non bloquantes de communication point à point

- Communication multipoints

La communication multipoints permet d'établir des communications collectives telles que de la diffusion de messages, de la dispersion, mais aussi du réassemblage et de la concentration de messages. Une primitive de synchronisation explicite permet de faire des points de synchronisations dans un groupe de processus communicants. Les principales primitives existantes de communication multipoints sont présentées dans le tableau ci-dessous.

Envoi et réception de messages	Synchronisation explicite
MPI_BCAST : diffusion MPI_SCATTER : dispersion MPI_REDUCE : réassemblage MPI_GATHER : concentration	MPI_BARRIER : permet de synchroniser un groupe de processus.

Tableau IV-4 : Primitives bloquantes de communication point à point

Primitives de contrôle :

- MPI_INIT et MPI_FINALISE

On ouvre une session MPI en utilisant MPI_INIT. On ferme cette session en utilisant MPI_FINALIZE.

- MPI_COMM_SIZE, MPI_COMM_RANK

MPI_COMM_RANK retourne l'identifiant du processus appelant.

MPI_COMM_SIZE retourne le nombre total de processus pris en charge dans le *communicator* actuel (voir annexe B).

Types de données prédéfinis

MPI contient plusieurs types de données prédéfinis. Mais il est aussi possible de construire ses propres structures de données constituées de compositions de types prédéfinis. Cette construction est effectuée en utilisant des primitives de dérivation de données [primitives MPI]. L'intérêt de la construction de structures de données réside dans la possibilité d'effectuer des envois de messages hétérogènes. Par exemples, on peut avoir besoin d'envoyer un message constitué d'un entier représentant la taille d'une liste, puis une série de données de type « float » constituant cette liste.

Dans la spécification MPI, les types de données prédéfinis sont attachés à un langage de programmation. La spécification MPI précise les types de données associés au langage C et au fortran. Le tableau ci-dessous présente les types prédéfinis pour le langage C.

Types de données		
MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG,	MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG,	MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_BYTE.

Tableau IV-5 : Les différents types de données

b. Implémentation de l'API MPI dans le cadres des systèmes MPSoC

L'implémentation de l'API du modèle de programmation parallèle MPI a fait l'objet de plusieurs travaux :

- [MPICH] [LAM/MPI] [HP-MPI] sont des exemples d'implémentations de l'API du modèle MPI, dans le cadre d'architectures multiprocesseurs classiques. Elles implémentent toutes les primitives de l'API. L'implémentation [MPICH] définie deux couches : une couche portable et une couche dépendante de la plateforme matérielle, appelée ADI pour *abstract design interface* [Gropp 95].
- [McMahon 96] propose deux approches pour porter l'implémentation MPICH de MPI vers des architectures embarquées caractérisées par des ressources mémoires limitées.
  - La première approche consiste à enlever des fonctionnalités (supporter seulement l'envoi standard de données, suppression des types de données utilisateur, de l'interface fortran,...) et à simplifier certaines autres (simplification de l'implémentation de `MPI_INIT`).
  - La seconde approche consiste à concevoir une bibliothèque qui implémente un ensemble réduit de primitives MPI. Cet ensemble est constitué par les primitives suivantes : `MPI_INIT`, `MPI_FINALIZE`, `MPI_COMM_SIZE`, `MPI_COMM_RANK`, `MPI_SEND`, `MPI_RECV`. Puis, d'y ajouter des primitives spécifiques à un contexte prédéterminé, par exemple des primitives de communication non bloquante, de communication multipoints ... pour créer des bibliothèques spécifiques. Toute fois, il faut optimiser l'implémentation de la partie ADI de chaque bibliothèque vis-à-vis de l'architecture ciblée.

Ce travail se limite à une optimisation à gros grains de MPI pour les systèmes embarqués, où seule la bibliothèque utilisée est incluse dans le système final. Une meilleure optimisation (à fins grains) serait alors d'implémenter dans le système final uniquement les primitives utilisées par l'application.

Pour le raffinement d'une application conçue avec le sous ensemble de primitives MPI, nous proposons l'utilisation de l'approche basée sur les graphes de dépendance de services, adoptée dans le flot ROSES (voir paragraphe III.5.2.2). Cette approche permet une implémentation automatique taillée sur mesure (seules les primitives utilisées par l'application sont implémentées). Un exemple d'un tel graphe est donné dans V.4.3.

c. Évaluation du modèle MPI

Le modèle de programmation parallèle MPI, de part l'utilisation de messages pour la communication, semble bien adapté pour les applications de type flot de données. De plus, comme son API est indépendante de l'architecture cible, elle lui permet de simplifier la communication entre des sous-systèmes processeurs hétérogènes. Cependant, cette indépendance nécessite une implémentation manuelle des primitives de l'API en fonction de l'architecture ciblée.

IV.4.2.4 Le modèle de programmation parallèle CORBA

Le modèle de programmation parallèle CORBA, pour *common object request broker architecture*, suit l'approche logicielle. Il est une représentation d'une architecture proposée par l'[OMG] en 1990. L'OMG a pour objectif de faire émerger des standards pour l'intégration et la

conception d'applications distribuées hétérogènes à partir de technologies orientées objet. Les concepts-clés mis en avant sont la réutilisation, l'interopérabilité et la portabilité des composants logiciels.

En CORBA, un système est décrit comme un ensemble de composants interconnectés via une interconnexion CORBA. Les composants peuvent être hétérogènes (écrits avec différents langages de programmation, ciblant différents systèmes d'exploitations ou machine d'exécution etc…).

CORBA est un modèle de programmation parallèle orienté objet. Il appartient à la classe 2 car seuls la décomposition -de l'application en tâches- et la concurrence -entre les tâches- sont explicites.

Les deux sous paragraphes suivants présentent les composants CORBA puis l'interconnexion CORBA.

Les composants CORBA

Un composant CORBA est un objet de l'application qui offre ou requiert des services. Chaque composant est encapsulé par une interface CORBA (figure IV-8). Dans cette interface, une description des différents services requis ou fournis est fixée. Une interface représente un accord de communication entre les différents composants.

Pour la description de ces interfaces, l' [OMG] définit le langage de définition des interfaces IDL (pour *interface description language*). Il permet de définir les différents services fournis ou requis de chaque composant CORBA.

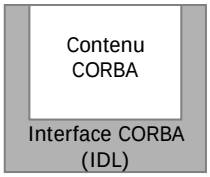


Figure IV-8 : Description du composant CORBA

Le langage OMG-IDL permet d'exprimer, sous la forme de contrats IDL [Geib 97], la coopération entre les fournisseurs et les utilisateurs des services. Il sépare l'interface de l'implantation des objets et en masque les divers problèmes liés à l'interopérabilité, l'hétérogénéité et la localisation de ceux-ci. Un contrat IDL spécifie les types manipulés par un ensemble de composants répartis, c'est-à-dire les types d'objets (ou interfaces IDL) et les types des données échangées entre les objets (figure IV-9). Le contrat IDL isole ainsi les clients et fournisseurs de l'infrastructure logicielle et matérielle les mettant en relation à travers le bus CORBA.

Chaque composant peut exporter certaines de ses fonctionnalités (services) sous la forme d'objets CORBA. Les interactions entre les composants sont alors matérialisées par des invocations à distance des méthodes des objets. La notion client/serveur intervient uniquement lors de l'utilisation d'un objet : le composant implantant l'objet est le serveur, le composant utilisant l'objet est le client. Bien entendu, un composant peut être à la fois client et serveur.

Pour permettre l'exécution de l'application dans un environnement CORBA, les contrats IDL sont projetés en souches IDL dans l'environnement de programmation du client et en squelettes IDL dans l'environnement de programmation du serveur. Le client invoque localement la souche pour accéder aux objets. Elle construit les requêtes, qui vont être transportées par le bus, puis délivrées par celui-ci aux squelettes IDL qui les délégueront aux objets. Ainsi, le langage OMG-IDL est la clé de voûte du bus d'objets répartis CORBA.

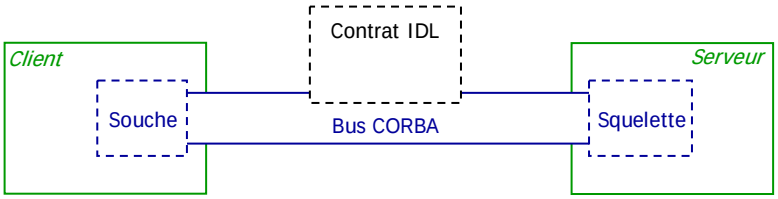


Figure IV-9 : Projection d'un contrat IDL en souche et en squelette

L'interconnexion CORBA

L'interconnexion CORBA, souvent appelée bus CORBA ou bus logiciel, propose un modèle orienté objet client/serveur d'abstraction et de coopération entre les composants répartis.

L'interconnexion CORBA est définie à travers le bus d'objets de CORBA noté ORB (pour *object request broker*). Le ORB est au cœur de l'architecture définie par l' [OMG]. Il est responsable de la mise en relation et de la communication entre tous les composants présents dans cette architecture distribuée. Il prend en charge le dialogue entre les composants serveurs et les différents clients qui s'y connectent. Il rend ce dialogue transparent quelles que soient les plates-formes, systèmes d'exploitation ou langages utilisés.

L'ORB représente une interprétation particulière des interconnexions à un très haut niveau d'abstraction. A ce niveau on peut présenter le modèle de spécification d'un système décrit en CORBA par la figure IV-10 suivante :

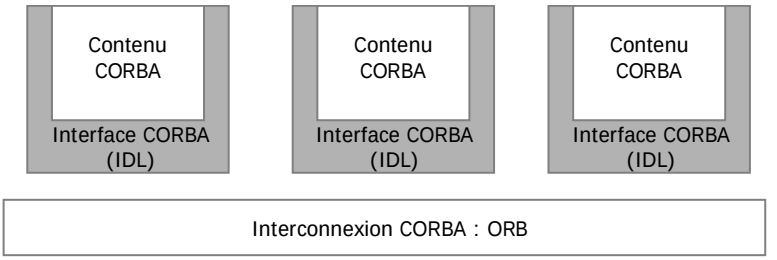


Figure IV-10 : Modèle de spécification d'un système décrit en CORBA

a. Interface et API du modèle CORBA

L'[OMG] propose un catalogue de plusieurs spécifications de CORBA [OMG CORBA SPEC]. Chaque spécification est propre à un contexte d'utilisation donnée. Pour les systèmes embarqués, l'OMG propose la spécification [minimumCORBA]. Cette spécification défini un sous-ensemble des primitives de l'API CORBA adéquat aux systèmes à ressources limitées. OpenFusion [e*ORB] de

Prismetech est un exemple d'une implémentation de cette spécification. Quelques primitives de l'API CORBA, telles que implémentées dans e*ORB sont présentées dans l'annexe D.

IV.5 Conclusion

Dans ce chapitre, nous avons étudié les modèles de programmation parallèle. Dans le premier paragraphe, nous avons donné une définition du terme modèle de programmation parallèle. Dans le second, nous avons proposé un flot pour la conception des interfaces logicielles/matérielles des systèmes MPSoC. Ce flot est basé sur l'utilisation de modèles de programmation parallèle. Les étapes du flot proposé ont été détaillées. Dans le dernier paragraphe, nous avons présenté une classification des modèles de programmation parallèle suivant leurs degrés d'abstraction. Puis, nous avons analysé plus en détail les modèles de programmation DSO/SMP, TTL, MPI et CORBA.

Chapitre V

Analyse d'une expérience de conception : un encodeur vidéo OpenDivX en utilisant le modèle de programmation parallèle MPI et l'architecture multiprocesseur ARM Integrator

V.1 Introduction

Dans ce chapitre nous présentons une analyse d'une expérience de conception réalisée suivant le flot proposé dans le chapitre précédent. Cette expérience est effectuée dans le cadre du projet national « ARCHiFLEX » en collaboration avec deux partenaires (STMicroelectronics et LETI). Le premier objectif du projet « ARCHiFLEX » est le développement d'une architecture flexible et reconfigurable pour environnements multiprocesseurs hétérogènes. Le second objectif est la définition et le développement d'une méthodologie permettant le développement du logiciel applicatif indépendamment de l'architecture matérielle et du système d'exploitation utilisé.

Notre contribution dans ce projet consiste à porter l'application sur l'architecture matérielle ciblée, en raffinant les primitives utilisées de l'API du modèle de programmation parallèle choisi. Notre objectif est l'identification d'un éventuel goulet d'étranglement pour la conception d'un système MPSoC en utilisant le flot de conception détaillé dans le paragraphe IV-3.

La figure V-1 montre l'application du flot lors de cette expérience de conception. L'expérience consiste à développer l'encodeur vidéo [OpenDivX], en utilisant le modèle de programmation parallèle [MPI] pour l'architecture multiprocesseur de la plateforme [ARM Integrator AP]. La spécification initiale de l'application est une implémentation séquentielle de l'encodeur vidéo OpenDivX.

Dans le cadre de cette expérience, l'architecture matérielle ciblée existe déjà. Il s'agit donc d'appliquer seulement la partie du flot qui concerne la conception de la partie logicielle (partie grisée de la figure V-1) : Tout d'abord, le code séquentiel est parallèlisé puis transposé sur la plateforme ciblée. Le résultat de cette étape a été débogué à haut niveau et validé en utilisant [MPICH]. L'étape suivante est la conception du logiciel dépendent du matériel (HdS). Cette étape a été réalisée en utilisant les outils du flot ROSES (présentés dans le paragraphe III.5). Ces outils permettent une conception automatique du HdS et se basent sur l'utilisation d'un graphe de dépendance de services (SDG) annoté par des paramètres extraits de la spécification VADeL de l'application OpenDivX. La dernière étape est le débogage et la validation de l'interface logicielle/matérielle. Elle a tété réalisée en en utilisant la plateforme ARM Integrator AP.

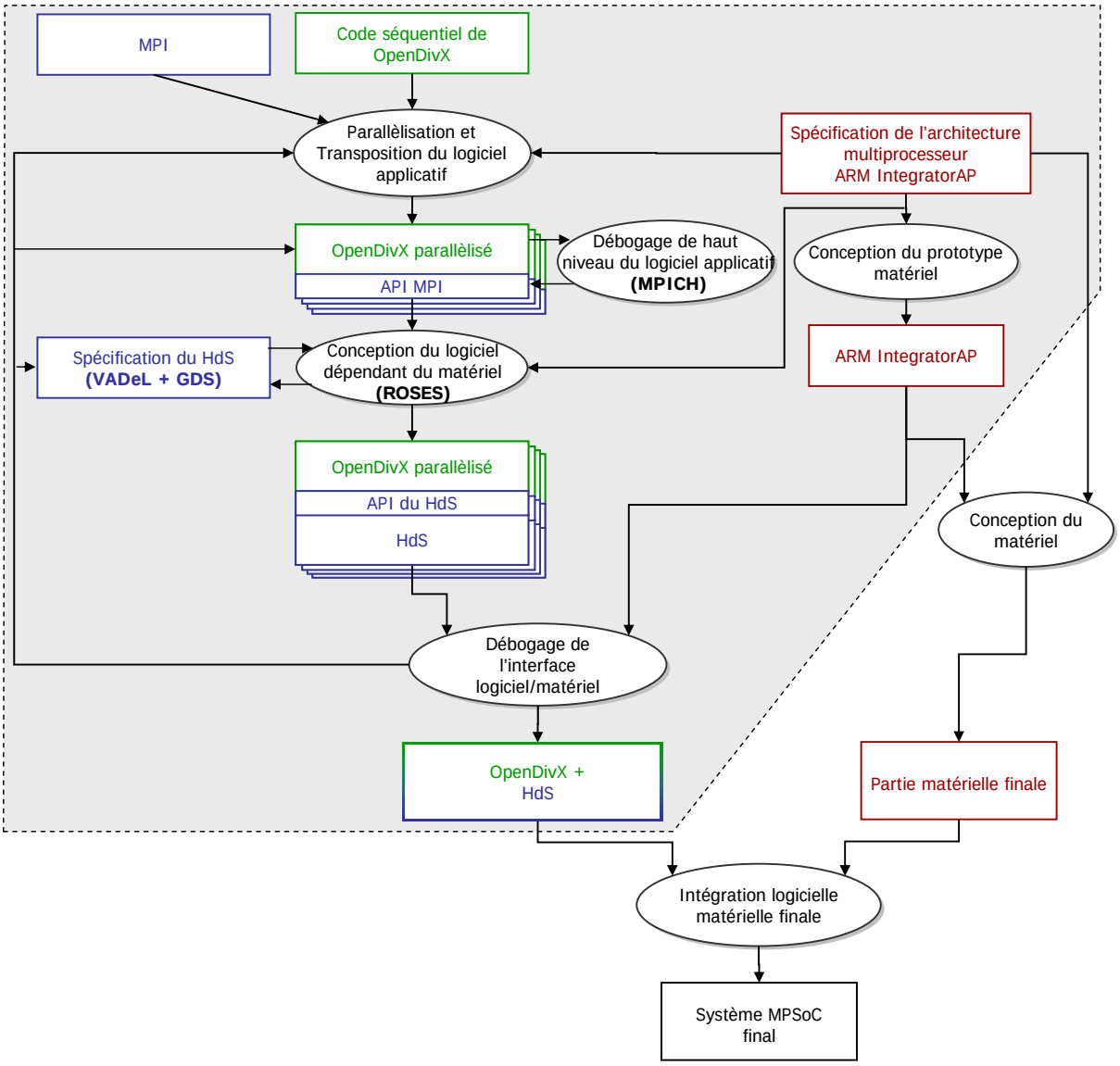


Figure V-1 : Flot de conception pour l'application OpenDivX

Le travail accompli lors de cette expérience a été réalisé en collaboration avec [Paviot 04] et [Sasonkgo 04]. Dans le second paragraphe de ce chapitre, nous détaillons l'application OpenDivX. Puis, dans le troisième paragraphe, nous discutons des choix des architectures logicielles/matérielles. Ensuite, nous détaillons les différentes étapes de la conception réalisées. Dans le cinquième paragraphe, nous présentons une analyse de cette expérience où nous exposons les leçons retenues et les perspectives.

V.2 Présentation de l'application : l'encodeur vidéo OpenDivX

L'encodeur OpenDivX est une application qui a pour but de coder un fichier vidéo en produisant un fichier compressé. Le but étant de réduire la taille du fichier vidéo initial. Le code de l'encodeur utilisé dans cette expérience est un code libre. Il a été développé dans le cadre du [Projet Mayo], qui est à l'origine du standard de compression vidéo actuel [DivX]. L'encodeur OpenDivX supporte uniquement une entrée de taille 176x144 (QCIF) sous le format YUV 4:1:1.

Pour développer une version de cet encodeur pour un système MPSoC, une bonne connaissance de l'application ainsi que des structures de données utilisées est requise. Dans la suite, nous présenterons le format YUV 4:1:1 ainsi que quelques notions de base concernant l'encodage vidéo. Puis nous étudierons les besoins en puissance de calcul de l'application OpenDivX dont on présentera un diagramme fonctionnel.

Le format du flux vidéo en entrée : le format YUV 4:1:1

Dans le flux vidéo, les données sont hiérarchisées selon une manière bien précise. Tout d'abord, la séquence vidéo commence par un code de début de séquence, contenant un ou plusieurs groupes d'images et se termine par un code de fin de séquence. Chaque groupe d'images débute également par un en-tête et comporte une ou plusieurs images.

Nous arrivons ensuite à l'image qui est l'unité élémentaire pour le codage d'une séquence vidéo. Il existe plusieurs façons de représenter une image dont les deux principales sont la représentation par matrices RVB (pour rouge, vert bleu) et la représentation par matrices de luminances et chrominances (notée YUV, où Y est la luminance, U est la chrominance blanc-rouge et V la chrominance blanc-bleu). Ces deux notations sont équivalentes et on utilisera pour l'encodeur OpenDivX la deuxième notation, plus économique en place mémoire.

Dans le format YUV, une image est donc constituée de trois matrices où chaque élément de la matrice représente un pixel. L'essentiel de l'information de l'image est stocké dans la matrice Y, car l'œil est plus sensible à la luminance. Plusieurs sous formats du format YUV existent et on les note selon les caractéristiques de leurs matrices YUV. Ainsi, le format 4:1:1 utilisé dans l'application OpenDivX indique que chaque pixel est échantillonné en luminance (Y), tandis qu'un pixel sur quatre est échantillonné pour la chrominance blanc-rouge (U) et blanc-bleu (V).

V.2.1 Principe de l'encodage vidéo

Dans une séquence vidéo, il existe deux sortes de redondances : la redondance spatiale et la redondance temporelle. L'encodage vidéo (figure V-2 – (c)) se base sur l'élimination de ces deux redondances : l'élimination de la redondance spatiale produit des images I (pour Intra-codé) tandis que l'élimination de la redondance temporelle produit des images P (pour Prédictives).

Le codage spatial (Images I)

La redondance spatiale (figure V-2 – (a)) est la redondance présente dans chaque image prise indépendamment des autres. On peut diminuer cette redondance en codant chaque image séparément en [JPEG] (pour joint photographic experts group). Sans entrer dans les détails de la compression JPEG, l'algorithme de compression d'une image fixe s'effectue en quatre étapes : transformation en cosinus discrète (DCT, pour *discrete cosinus transform*), quantification, zigzag et codage de Huffman.

Typiquement, une image I est intercalée dans le flux vidéo toutes les 10 à 15 images. Avec un débit de 25 à 30 images par seconde, cela veut dire qu'il y a 2 ou 3 images I par seconde dans le flux.

Le codage prédictif (Images P)

Une image P est codée par rapport à l'image précédente. Elle ne code en fait que la différence bloc par bloc (un bloc est une matrice 8x8 pixels) avec l'image précédente. Cette différence est codée spatialement (figure V-2 – (b)), comme les images I.

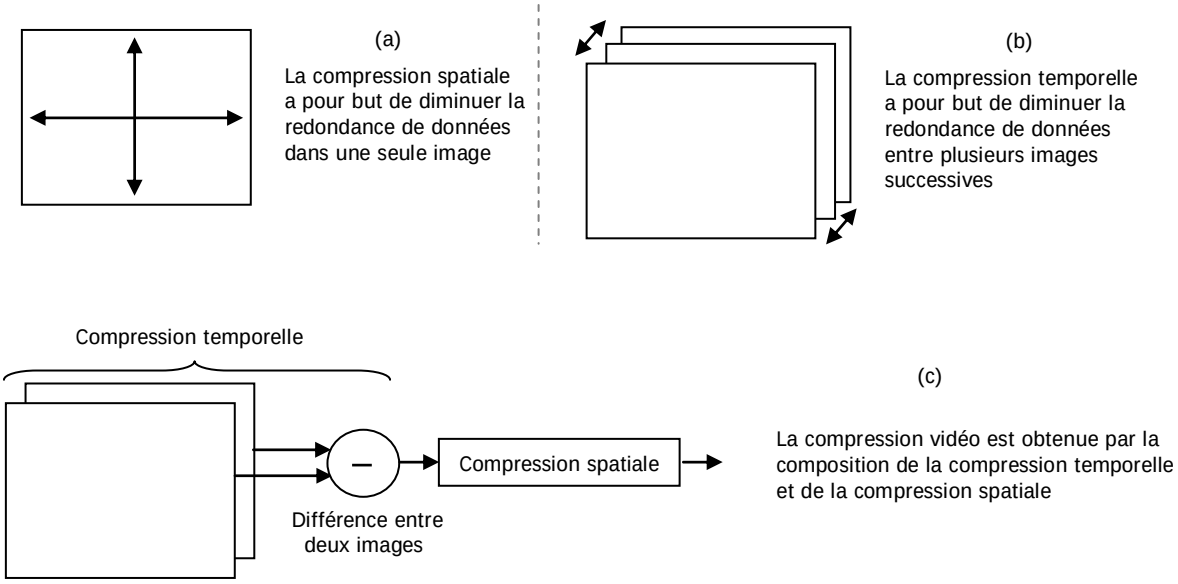


Figure V-2 : (a) Compression spatiale, (b) Compression temporelle et (c) Compression vidéo

La figure V-3 illustre un exemple dans lequel les images P sont très utiles. Nous avons trois images consécutives qui ont le même arrière-plan mais diffèrent par la position d'un animal. Les macroblocs (un macrobloc est une matrice 4x4 blocs) qui représentent l'arrière-plan restent les mêmes d'une image à l'autre, mais ceux qui contiennent l'animal devront être décalés d'une certaine valeur qui est à calculer.

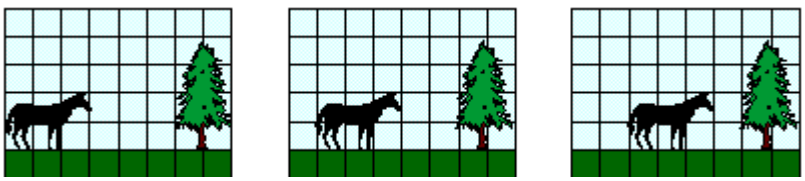


Figure V-3 : Exemple de l'utilité de l'utilisation des images prédictives (P)

Cette technique est appelée la compensation de mouvement. Quant un macrobloc est compressé par cette technique, le résultat contient les informations suivantes :

- Un vecteur de déplacement entre le macrobloc de référence (celui de l'image précédente) et le macrobloc qui va être codé. Ce vecteur rend donc compte du déplacement du macrobloc dans chaque direction par rapport à sa position précédente.
- La différence entre le contenu du macrobloc de référence et le macrobloc qui va être codé. Cette différence est appelée le terme d'erreurs et elle est codée en JPEG.

La figure V-4 montre un flux vidéo compressé. Une image I est codée périodiquement, tandis qu'entre chacune d'entre elles, on intercale des images P. Les limitations de cette approche de codage sont (1) si une erreur se produit lors du codage d'une image P, toutes les images P suivantes contiendront également cette erreur ; (2) et si une image I se perd, le décodage des images P suivantes est impossible jusqu'à ce qu'une nouvelle image I arrive.

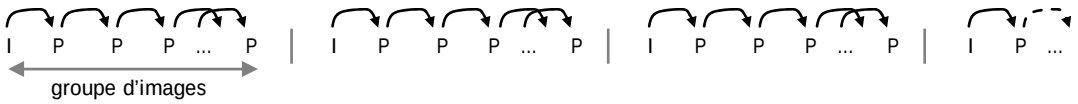


Figure V-4 : Flux vidéo compressé

La compensation de mouvement

Revenons un peu sur le principe de compensation de mouvement. Ce principe est à la base de la génération des images P. La figure V-5 montre le fonctionnement d'un système de compression par compensation de mouvement. Dans cette figure, les rectangles à coins arrondis représentent des étapes de la compression par compensation de mouvement et les rectangles représentent une entrée et/ou une sortie d'une étape. Les rectangles grisés représentent le résultat qui sera encodé dans le fichier résultat.

La première étape est le calcul des vecteurs de mouvement de chaque macrobloc. Cette étape prend comme entrée l'image précédente décompressée notée PD et la nouvelle image actuelle notée A. Elle calcule les vecteurs de mouvement de chaque macrobloc de l'image A. Les vecteurs de mouvements sont ensuite appliqués aux macroblocs de l'image PD pour produire l'image prédictive P. Ensuite, le calcul de la différence entre l'image P et l'image A produit le terme d'erreur. Ce sont les vecteurs de déplacement ainsi que le terme d'erreurs qui seront encodés dans le fichier résultat.

L'application du terme d'erreur sur l'image P produit l'image résultante de la décompression du résultat de la compression de l'image A. Ce résultat servira pour la compression de l'image A+1.

On remarque que dans cette approche d'encodage vidéo, l'encodeur contient lui-même un décodeur. Dans la figure V-5, les étapes du décodeur sont l'application des vecteurs de mouvement et le calcul de la nouvelle image décompressée. On note ici que les seules entrées de ce décodeur sont l'image précédente décompressée, les vecteurs de mouvement et le terme d'erreur.

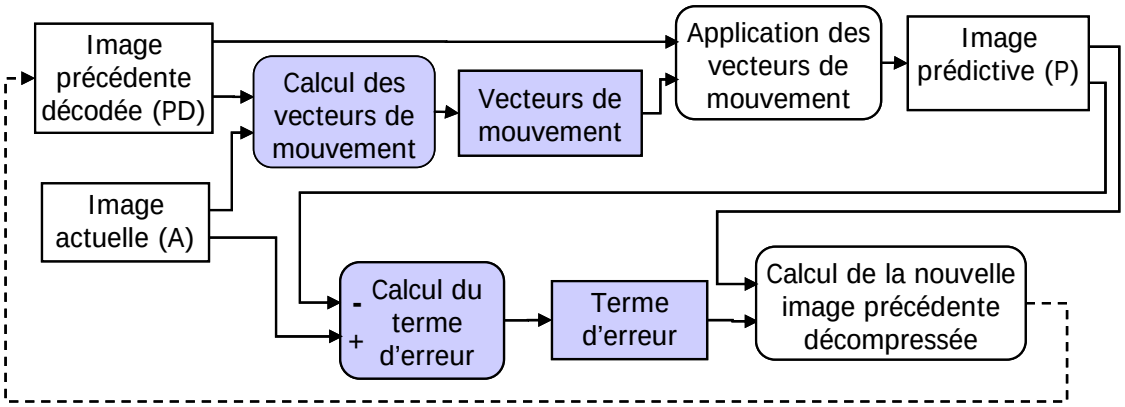


Figure V-5 : Principe de la compression par compensation de mouvement

V.2.2 Besoins en puissance de calcul de l'application OpenDivX

L'encodeur OpenDivX (figure V-6) est initialement écrit sous forme d'un code séquentiel comme la plupart des codes des applications multimédia. La taille totale du code de l'application est de l'ordre de 10K lignes C sur 43 fichiers. Cette application a besoin d'un processeur Pentium III fonctionnant à une fréquence de 1 GHz pour pouvoir encoder une séquence vidéo de 20 images par seconde sous le format QCIF (176x144 pixels). Exécuter cette application sur un système MPSoC est un vrai challenge puisqu'elle a besoin d'une grande puissance de calcul qu'un seul processeur embarqué, comme un ARM7 ou un ARM9, ne peut pas fournir. En effet, l'exécution de cette version séquentielle de OpenDivX sur un processeur ARM9 fonctionnant à 28 Mhz prend 227 secondes pour traiter une séquence de 20. Donc, pour aboutir à une exécution en temps réel, on a besoin d'optimiser le code logiciel de l'application, d'utiliser des accélérateurs matériels et si possible des architectures multiprocesseurs hautement parallèles.

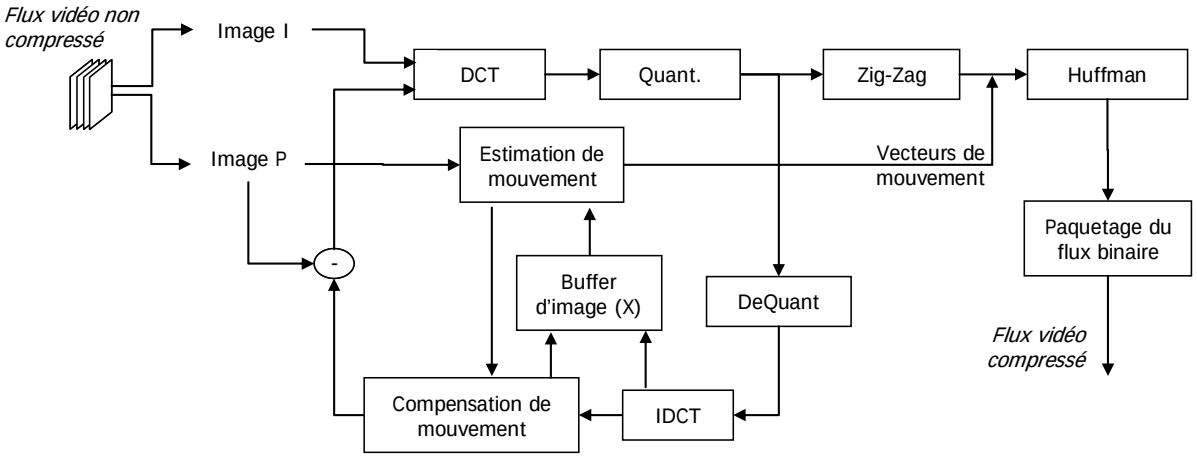


Figure V-6 : Représentation fonctionnelle de l'encodeur OpenDivX

V.3 Choix des architectures logicielles/matérielles pour la conception de l'encodeur OpenDivX

Nous discutons dans cette partie des choix des architectures logicielles/matérielles pour la conception de l'encodeur vidéo OpenDivX. Ces choix englobent le choix du modèle de programmation parallèle et de l'architecture matérielle ciblée.

V.3.1 Choix du modèle de programmation parallèle : MPI

Le modèle de programmation parallèle utilisé lors de la conception de l'application OpenDivX est MPI (voir paragraphe IV.4.2.3). Il a été choisi car il est adéquat aux architectures ciblées dans le projet ARCHiFLEX. En effet, MPI permet d'éviter les problèmes issus d'un adressage différent d'une même zone mémoire au niveau de processeurs hétérogènes, il simplifie alors le développement de l'application. Cependant, le travail qui a été réalisé pour MPI peut être fait avec un autre modèle de programmation parallèle.

V.3.2 Choix de l'architecture matérielle ciblée : ARM Integrator AP

Le choix de la plateforme de prototypage [ARM Integrator AP] est motivé par sa disponibilité chez le groupe SLS. Outre cette disponibilité, l'utilisation d'une plateforme de prototypage existante permet de passer directement à une phase de débogage sans avoir à concevoir un prototype matériel (il faut toutefois configurer la plateforme). La plateforme permet aussi une exécution et un débogage plus rapide qu'un modèle de simulation de l'architecture (qui doit être développé) notamment grâce à la suite logicielle de débogage fournie par ARM (AXD, Code Warrior).

La figure V-7 présente l'architecture de la plateforme ARM Integrator AP. Cette plateforme est composée de deux processeurs ARM7 et de deux processeurs ARM9. Le média de communication utilisé pour connecter les quatre processeurs est un bus AMBA AHB. Chaque processeur est connecté au bus AMBA par un FPGA. Chaque FPGA contient une interface permettant l'utilisation du bus AMBA ainsi qu'un contrôleur mémoire arbitrant l'accès à la mémoire locale du processeur. Grâce à ce contrôleur, il est possible d'accéder à chaque mémoire locale depuis le processeur mais aussi depuis le bus AMBA. Ainsi chaque processeur peut accéder aux mémoires locales des autres processeurs. Ces interfaces permettent aussi d'envoyer un signal d'interruption à tous les processeurs. Des registres permettent de masquer les processeurs auxquels on ne veut pas envoyer d'interruptions. La plateforme possède aussi un FPGA libre connecté au bus AMBA.

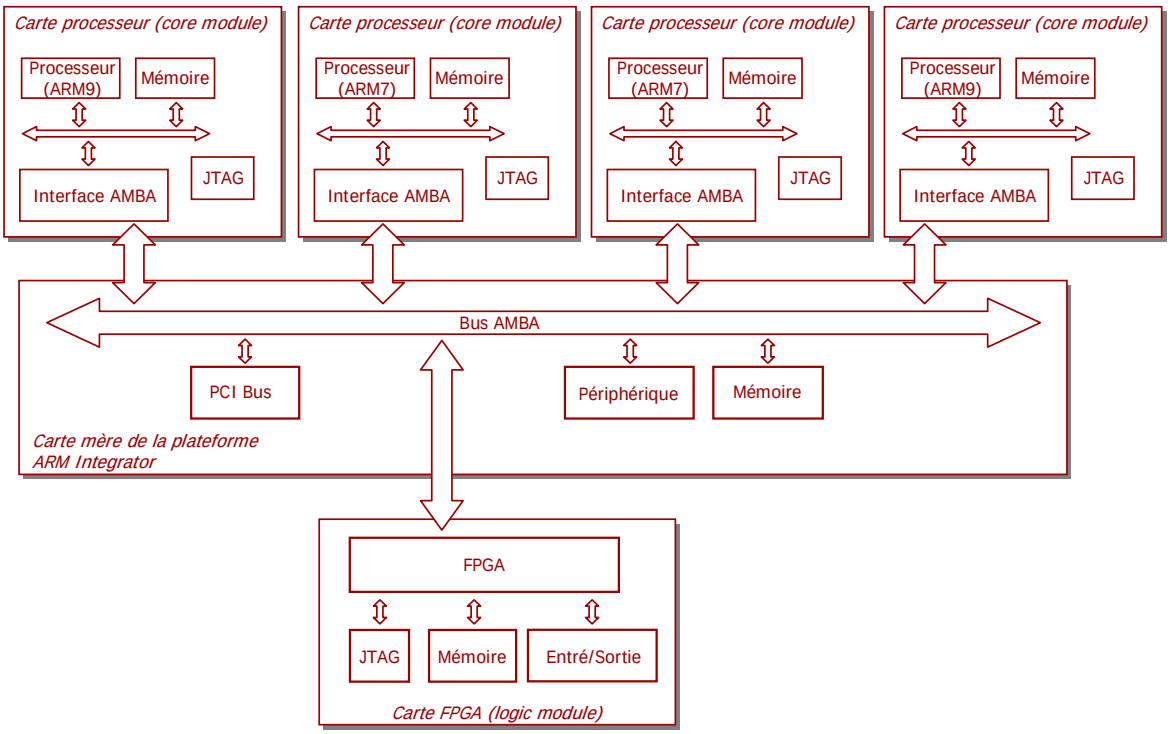


Figure V-7 : Architecture de la plateforme de prototypage ARM Integrator AP

Les contraintes imposées par la plateforme

Un seul FPGA est libre sur cette plateforme. Cependant, il n'est pas placé entre le média de communication (le bus AMBA) et les processeurs. Il n'est donc pas possible de l'utiliser pour implémenter une éventuelle adaptation matérielle de l'interface logicielle/matérielle. Toutefois, il existe une adaptation matérielle fixe entre chaque processeur et le bus AMBA. Elle joue le rôle d'une interface AMBA et offre les possibilités pour un processeur d'écrire dans la mémoire d'un autre processeur et de générer une interruption matérielle chez d'autres processeurs.

La conception de l'interface logicielle/matérielle pour cette plateforme consiste donc à concevoir uniquement l'adaptation logicielle (HdS) et à exploiter l'adaptation matérielle déjà existante.

V.4 Conception des interfaces logicielles/matérielles pour l'encodeur OpenDivX

Dans ce paragraphe, nous présentons l'application du flot proposé dans le paragraphe IV-3 pour la conception des interfaces logicielles/matérielles de l'encodeur OpenDivX (figure V-1). Nous détaillerons dans la suite les différentes étapes relatives à la conception du HdS.

V.4.1 **Parallèlisation de l'application OpenDivX**

V.4.1.1 Profilage de l'application OpenDivX

Pour parallèliser le code séquentiel de l'application pour une architecture composée de quatre processeurs, nous avons établi le profile de l'exécution du code séquentiel. Ce profile nous permet de détecter des goulets d'étranglement du calcul. Pour réaliser cette étape, nous avons utilisé un

simulateur de processeurs ARM. Il en a résulté l'identification de la fonction d'estimation et de compensation de mouvement (MEC) -représentée par la partie grisée de la figure V-8 (a)- comme une bonne fonction candidate pour être parallèlisée : cette fonction représente plus de 65% du cycle de calcul total.

V.4.1.2 Parallèlisation de code de l'application OpenDivX en utilisant MPI

L'architecture ciblée étant composée de quatre processeurs, nous avons créé trois tâches appelées « esclaves » de la fonction MEC. Chacune d'entre elles sera appliquée à un tiers de l'image source. Puis nous avons créé une quatrième tâche appelée « maître » en utilisant la partie restante du code de l'application OpenDivX. La figure V-8 (b) montre quatre tâches et la partie de l'image sur la quelle chacune opère.

La tâche « maître » lit des images à partir du fichier source à encoder. Elle contient le code à exécuter s'il s'agit d'une image à coder en Inter (image I). Dans le cas d'une image P, elle envoie à chaque tâche « esclave » un tiers (ici le premier tiers au premier « esclave », le second au second et le troisième au troisième) pour être traité. Chaque tâche « esclave » calcule alors les vecteurs de mouvement de chaque macro bloc ainsi que la compensation de mouvement. Puis elle envoie les résultats à la tâche « maître » qui effectue, dessus encore, les derniers traitements (compression de huffman) avant de les stocker dans le fichier cible.

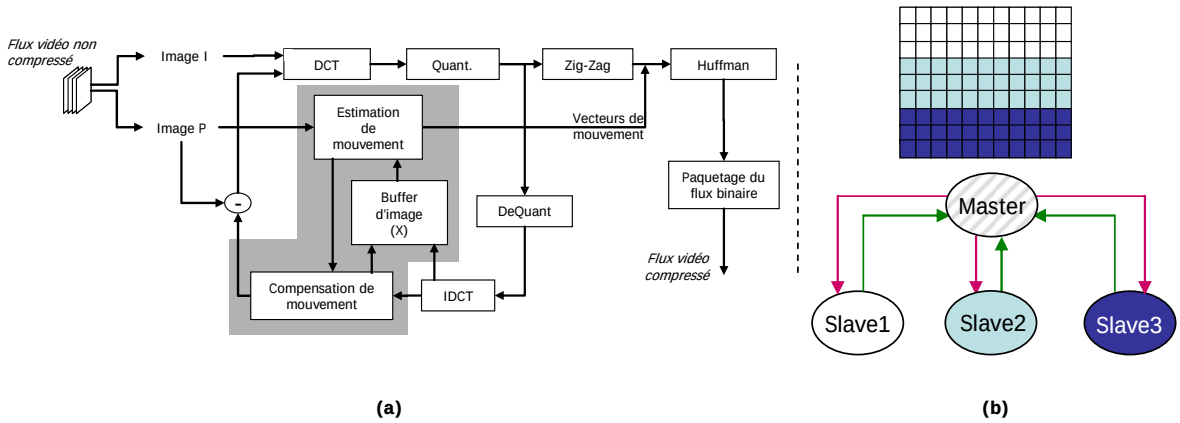


Figure V-8 : Parallèlisation de l'application

Le tableau V-1 détaille les données échangées entre la tâche « maître » et une tâche « esclave » pour la compression d'une séquence vidéo composée par des images dont la taille unitaire est de 38 KB. Sur chaque ligne, on trouve le nom de la structure de données, sa taille et son type. On remarquera que la taille de la majorité des structures de données a été divisée par 3 car il s'agit d'envoyer uniquement un tiers de l'image à chaque tâche « esclave ». Seuls les structures mémorisant l'image précédente (Prev, Prev_u et Prev_v) ont été envoyées en totalité aux différents « esclaves » pour permettre la recherche des vecteurs de mouvement sur l'intégralité de l'image et non pas uniquement sur le tiers. En terme de volume de données

échangées, le « maître » envoi 149 KB (resp. reçoit 25 KB) par image vers (resp. de) chaque « esclave ».

Nom	Taille	Type	Commentaire
Direction : de la tâche maître vers la tâche esclave			
Curr_comp_y	176x144 /3	SInt	Composante Y, image à coder
Curr_comp_u, Curr_comp_v	176x144 /4x3	SInt	Composantes U et V, image à coder
Curr	176x144 /3	SInt	Composante Y, image à coder
Prev	208x176	SInt	Composante Y, image précédente
Prev_u, Prev_v	208x176 /4	SInt	Composantes U et V, image précédente
Tab_int	14	SInt	Entier paramètres
Tab_float	2	Float	Flottants paramètres
Direction : de la tâche esclave vers la tâche maître			
Curr_comp_y	176x144 /3	SInt	Composante Y, compensation mvt
Curr_comp_u, Curr_comp_v	176x144 /4x3	SInt	Composantes U et V, compensation mvt
Mv16_W, Mv16_H	11x9x4 /3	Float	Vecteurs mvt 16x16, composantes W et H
Mv8_W, Mv8_H	11x9x4 /3	Float	Vecteurs mvt 8x8, composantes W et H
Mode_16	11x9 /3	SInt	Mode du codage du MacroBloc (I/P)

Tableau V-1 : Données échangées entre la tâche « maître » et une tâche « esclave »

Le code parallèlisé est validé en utilisant [MPICH] (implémentation de MPI sur Unix) et en utilisant un modèle de simulation de MPI en SystemC noté MPI/SC que nous avons développé. Le code est valide si le résultat de son exécution est identique à celui du code séquentiel. La parallèlisation a requis un effort de trois hommes/semaine. En terme de modifications de code nous avons modifié et créé 1029 lignes de code dans des différents fichiers sources.

V.4.1.3 Primitives MPI utilisées pour la conception de l'application OpenDivX

Dans notre cas, il s'agit d'une communication point à point. Les primitives de communication point à point de MPI sont classables en deux sous-ensembles : celui des communications bloquantes et celui des communications non bloquantes. Pour notre expérience, faute de temps, nous nous limitons aux primitives bloquantes, plus simples et donc plus rapide à raffiner.

Nous avons utilisé les primitives suivantes pour la conception de l'encodeur OpenDivX. Il s'agit des primitives `MPI_SEND`, `MPI_RECV` pour l'échange de messages et des primitives de contrôle `MPI_INIT` et `MPI_FINALISE`.

Pour illustrer l'utilisation des primitives MPI, la figure ci-dessous montre un exemple où la tâche « maître » envoie à la première tâche « esclave » (esclave1) une donnée de taille 176x16x3 entiers du type `MPI_SHORT` (`short int`).

```
1 :   //Au niveau du maître, on envoi des données
2 :   MPI_Send (buff_curr_comp_y1, 176*16*3, MPI_SHORT,SLAVE_1,
3 :              MSG_MASTER_TO_SLAVE1, MPI_COMM_WORLD) ;
4 :
```

```
1 :   //Au niveau du esclave1 on reçoit des données
2 :   MPI_Recv (&buff_curr_comp_y_slave, 176*16*3, MPI_SHORT, MASTER,
3 :               MSG_MASTER_TO_SLAVE1, MPI_Comm_WORLD,& status) ;
4 :
```

Figure V-9 : Exemple d'utilisation des primitives MPI_SEND et MPI_RECV

V.4.2 Transposition de l'application OpendivX sur la plateforme de prototypage ARM Integrator

La phase de transposition correspond pour notre cas à fixer le processeur cible sur lequel s'exécutera chaque tâche. C'est une allocation statistique des tâches. Nous avons choisi d'exécuter la tâche « maître » sur un ARM9 et d'exécuter chacune des trois tâches « esclave » sur un processeur (esclave 1, esclave 2, esclave 3 respectivement sur le processeur ARM7, ARM7 et ARM9).

Aussi, pendant cette étape, nous avons fixé la valeur des paramètres qui guident le raffinement automatique des interfaces logicielles/matérielles. Nous avons fixé, par exemple, le plan mémoire pour chaque sous-système : pour chaque couple émetteur/récepteur, nous avons utilisé une paire de FIFO ; une FIFO de contrôle et une FIFO de données (voir la section suivante pour plus de détails). Si on considère la FIFO de contrôle, appelée `FIFOTag`, on doit fixer la valeur des paramètres présentée dans le tableau V-2 dans le plan mémoire.

D'autres paramètres, dont la valeur doit être fixée à cette étape, sont présentés dans la partie suivante qui détaille leurs utilisations pour la conception du HdS.

Paramètre	Signification
`AddressHeadFIFOTagAddress`	l'adresse où est stockée l'adresse de la tête de la FIFO contrôle
`AddressTailFIFOTagAddress`	l'adresse où est stockée l'adresse de la queue de la FIFO contrôle
`NbElementFIFOTagAddress`	l'adresse où est stocké le nombre d'éléments actuellement dans la FIFO contrôle
`NbElementMaxFIFOTag`	Le nombre d'éléments maximal autorisé dans la FIFO contrôle
`FIFOTagAddressSem`	L'adresse du sémaphore pour l'accès à la FIFO contrôle

Tableau V-2 : Paramètres a spécifiés pour la FIFO contrôle

Une fois que les valeurs de tous les paramètres sont fixées on obtient la spécification VADeL (figure V-10) qui sera utilisée pour le raffinement des primitives de communication. Le code complet de cette spécification en VADeL de l'application OpenDivX est donné dans l'annexe C.

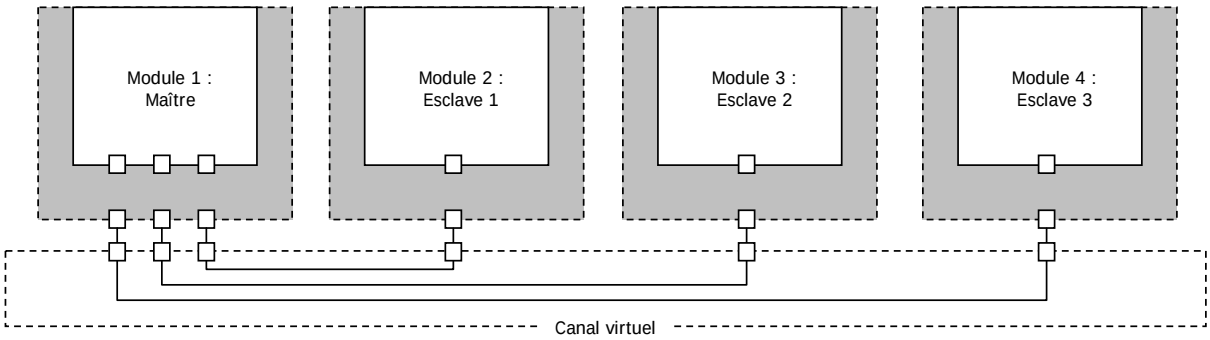


Figure V-10 : Spécification de l'application OpenDivX

V.4.3 Conception du HDS pour le raffinement du modèle de programmation parallèle

Dans cette section, nous détaillons la conception du HdS pour le raffinement des primitives MPI utilisées pour la conception de l'encodeur OpenDivX. Le raffinement consiste à les implémenter sur l'architecture cible. Cette implémentation est 100% logicielle (conception du HdS seulement) à cause des contraintes de l'architecture matérielle (ne permet pas le développement d'une couche d'adaptation matérielle).

Les primitives `MPI_SEND` et `MPI_RECV` utilisées pour la communication dans l'application OpenDivX doivent donc être implémentées sur la plateforme ARM Integrator AP. La figure V-11 montre l'utilisation des FIFO pour l'implémentation d'une paire de primitives `MPI_SEND/MPI_RECV`. Nous avons utilisé et géré deux FIFO (FIFO de contrôle, appelée `FIFOTag` et FIFO de données appelée `FIFOData`) pour chaque sens de communication entre deux tâches :

- Quand la primitive `MPI_SEND` est appelée par la tâche « maître », elle empile les données à envoyer dans la FIFO de données, puis elle met à jour les informations de contrôle (par exemple le nombre total de données dans la FIFO données). Finalement, elle envoie une requête pour émission vers la tâche « esclave ».
- Quand la primitive `MPI_RECV` est appelée par la tâche « esclave », elle vérifie tout d'abord la FIFO de contrôle pour détecter s'il y a des données disponibles. Puis, si tel est le cas, elle achemine les données vers leur destination (espace mémoire réservé par la tâche « esclave »). Sinon, la tâche « esclave » est bloquée en attendant sur un sémaphore le changement de statut de la FIFO de contrôle. La tâche « esclave » sera alors réveillée par une interruption. L'accès aux sections critiques (FIFO donnée, FIFO contrôle) est protégé par des sémaphores.

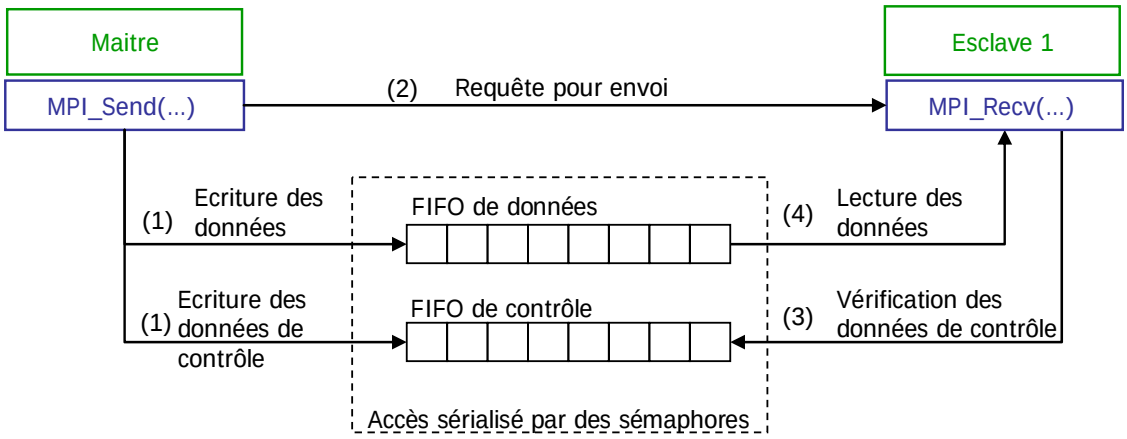


Figure V-11 : Utilisation de deux FIFO pour le raffinement des primitives MPI

Nous avons aussi utilisé des registres pour gérer les FIFO données et contrôle. Ces registres sont :

- 2 registres indiquant le nombre d'éléments contenus dans la FIFO
- 2 registres implémentant des sémaphores
- 2 registres pour l'identificateur d'interruption
- 2 registres contenant l'adresse de Tête de chaque FIFO
- 2 registres contenant l'adresse de Queue de chaque FIFO

La figure V-12 montre le choix d'allocation des ressources de communication. Les FIFO et les registres servant à les contrôler sont implémentés dans la mémoire locale du processeur qui envoie les messages.

Échange de données

La figure V-13 présente le processus d'envoi d'un message. Un message correspond à un ensemble d'éléments de type `datatype`. Si le type de données utilisé est par exemple `MPI_long_long` (un élément fait 64 bits), le message complet fait donc count*64 bits. Ensuite ces éléments sont lus dans le buffer source et découpés en `cells` de 32 bits pour être stockés dans le buffer de communication. Enfin, lorsque le récepteur est prêt à recevoir le message, les `cells` sont envoyées en mode `burst` via le bus AMBA. Chaque `burst` correspond à l'envoi d'un ensemble de `cells` appelé paquet.

Les autres types d'échanges sont des accès mémoire effectués par le processeur émetteur et le processeur récepteur pour contrôler les FIFO et pour écrire dans les registres permettant l'identification des interruptions ou encore pour accéder aux sémaphores.

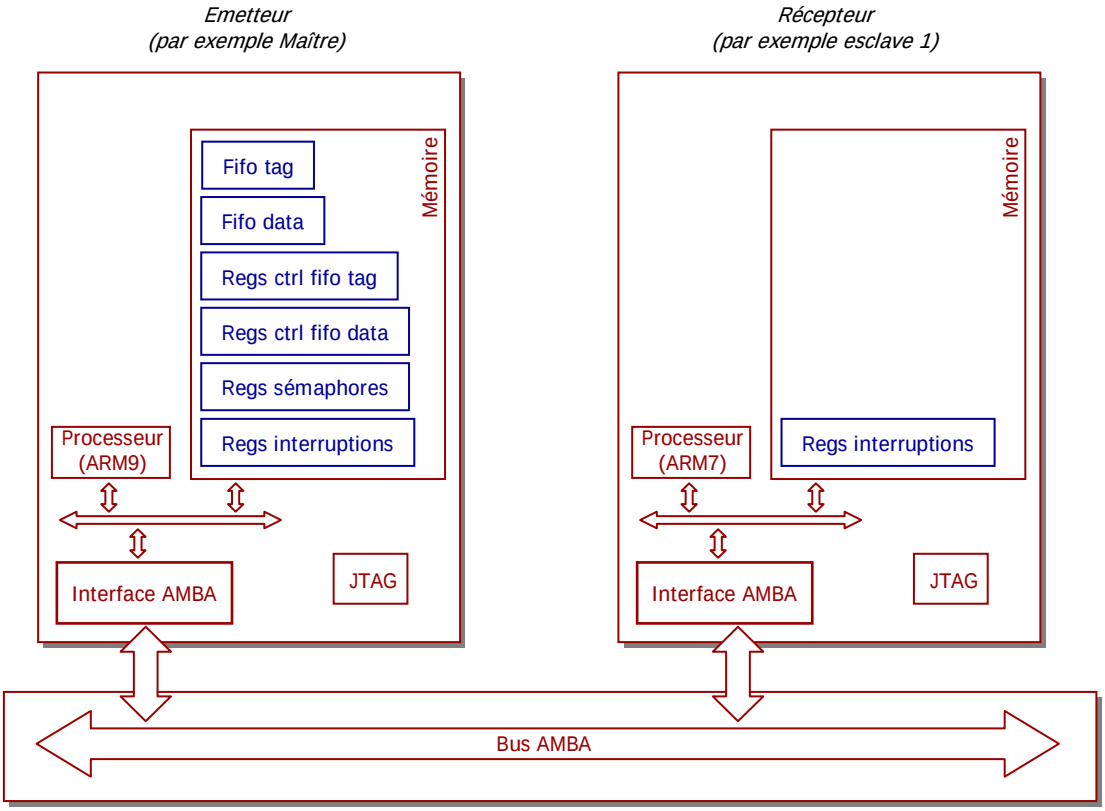


Figure V-12 : Allocation des ressources de communication

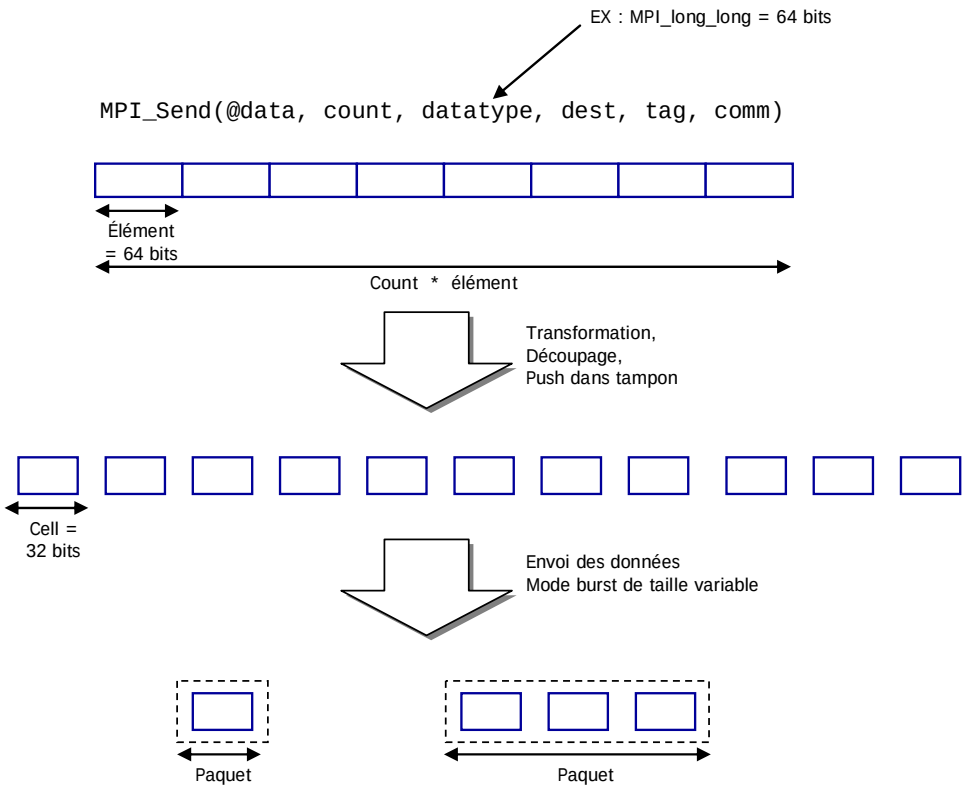


Figure V-13 : Décomposition, transformation et transfert des données

Liste de paramètres :

La figure V-14 reprend la spécification de la tâche « maître » en focalisant sur les paramètres nécessaires à la génération automatique de l'implémentation logicielle de `MPI_SEND` et `MPI_RECV`. Pour implémenter les primitives MPI, nous avons du fixer les paramètres des détails de l'implémentation. Ces paramètres sont :

- `R_FIFOTagAddress` : adresse de la FIFO stockant les étiquettes (*Tag*) des messages.
- `R_AddressHeadFIFOTagAddress`, `R_AddressTailFIFOTagAddress`, `R_NbElementFIFOTagAddress`, `R_NbElementMaxFIFOTag` : ces adresses servent au contrôle de la FIFO d'étiquettes. La FIFO étant circulaire, on a besoin de connaître les adresses des pointeurs de tête et de queue, l'adresse où est stockée le nombre d'éléments contenus ainsi que le nombre maximum d'éléments pouvant être contenus.
- `R_FIFODataAddress` : adresse de la FIFO stockant les données du message.
- `R_AddressHeadFIFODataAddress`, `R_AddressTailFIFODataAddress`, `R_NbElementFIFODataAddress`, `R_NbElementMaxFIFOData` : adresses servant au contrôle de la FIFO de données (les champs ont la même signification que pour la FIFOTag).
- `R_FIFOTagAddressSem`, `R_FIFODataAddressSem` : adresses de zones mémoires servant pour l'implémentation d'un sémaphore permettant de sérialiser l'accès aux régions et données critiques.
- Tous les paramètres commençant par `R_` sont dédiés aux ressources permettant l'implémentation de `MPI_RECV`. Les mêmes paramètres existent en version qui commence par `S_` pour l'implémentation de `MPI_SEND`.
- `ProcessorId`: numéro d'identification du processeur.
- `RegExtItNumberAddress` : adresse du registre où les numéros d'interruption sont récupérés.

Génération du code du HdS

Une fois les valeurs des paramètres fixées, nous avons utilisé les outils du flot ROSES pour la génération du code du logiciel dépendant du matériel (figure V-14 – (d)). La génération est basée sur l'utilisation d'une bibliothèque de composants de l'adaptation logicielle. Cette bibliothèque a été développée en collaboration avec [Paviot 04].

L'utilisation de ce mécanisme de génération basée sur des bibliothèques permet une exploration plus rapide de l'espace de conception du HdS. Il suffit de modifier les valeurs des paramètres (par exemple la taille d'une FIFO) et de générer ensuite la nouvelle implémentation du HdS.

La bibliothèque est organisée sous la forme d'un graphe de dépendance de services (SDG). La figure V-15, montre le SDG pour les primitives MPI_SEND et MPI_RECV. Sur cette figure les éléments sont représentés par des rectangles à coins arrondis et les services par des ovales. Pour simplifier la figure, l'élément `SoftInterrupt` a été représenté deux fois.

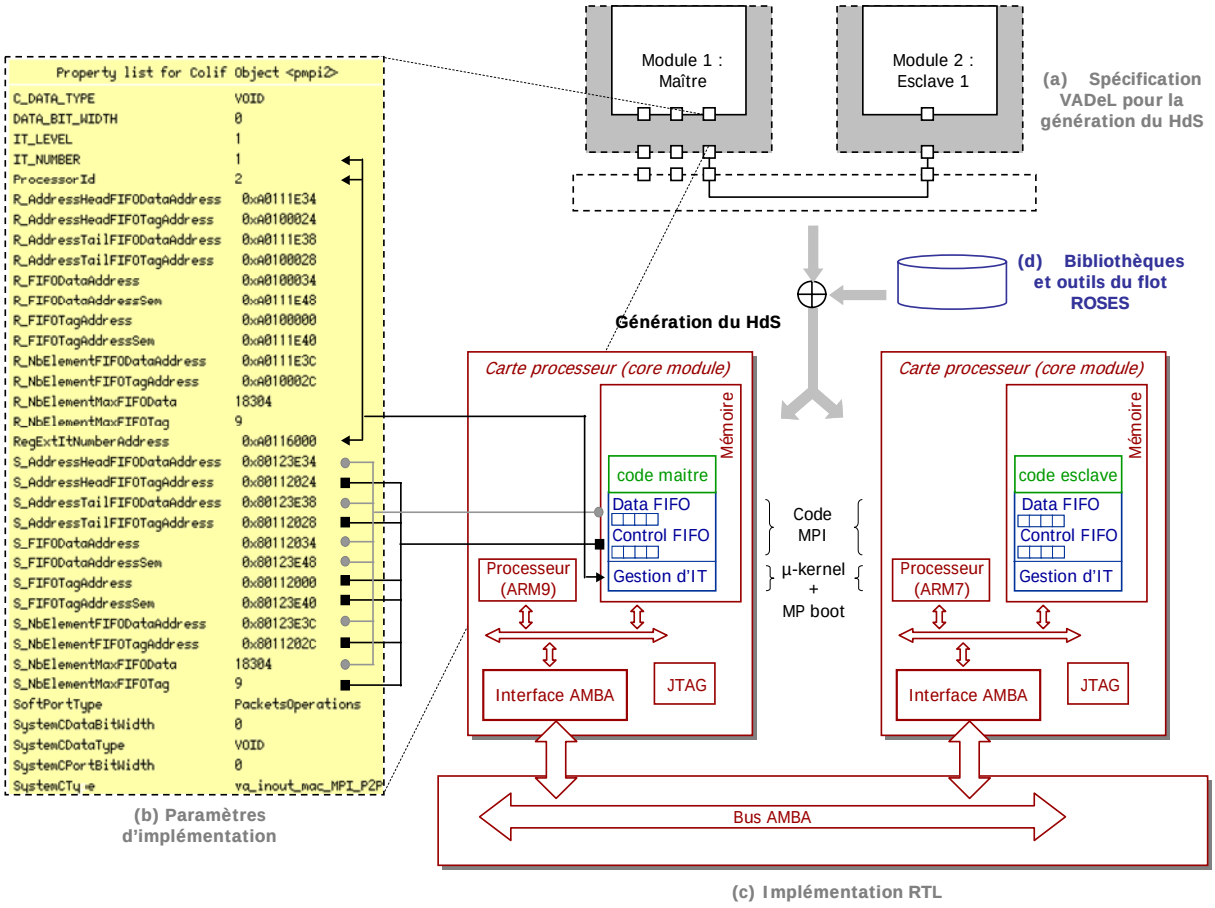


Figure V-14 : Spécification du service de communication MPI_Send

Le GDS peut être découpé en deux parties : une partie qui contient les éléments assurant le comportement de la couche middleware (PPM) et qui se base sur la seconde partie qui assure le comportement des couches système d'exploitation et abstraction du matériel (OS + HAL). Lors de la conception du GDS, nous avons adapté et réutilisé des éléments déjà existants dans les bibliothèques de ROSES pour la partie OS+HAL. Nous avons cependant conçu les éléments de la partie PPM. Nous nous limiterons alors, à la description des éléments et services de la partie PPM, celle de la partie OS + HAL peut être trouvée dans [Gauthier 01].

La primitive `MPI_SEND` est fournie par un composant `send_manager`. Ce composant a besoin des trois services :

- `read_args`, pour le traitement des arguments. Ce service est fourni par l'élément `args_reader`.
- `data_segmentation`, pour segmenter les données comme décrit dans la figure V-13. Ce service est fourni par le composant `data_manager`.
- `data_send`, pour l'envoi des données en utilisant les FIFO de données et de contrôle. Ce service est fourni par l'élément `external_exchange_manager`. Ce dernier utilise les services `write_in_fifo` et `read_from_fifo` fournis par l'élément `fifo_manager`, ainsi que les services de synchronisation `bloc` et `unbloc` fournis par l'élément `synchronizer`. L'élément `fifo_manager` utilise le service `fifo_update` pour la lecture et l'écriture dans les FIFOs, les

services `P` et `V` pour agir sur les sémaphores des FIFOs et le service d'ordonnancement `schedule`.

La primitive `MPI_RECV` utilise les mêmes éléments que `MPI_SEND` à la différence qu'elle a besoin de services de réception `data_recv` et `data_rassembly`.

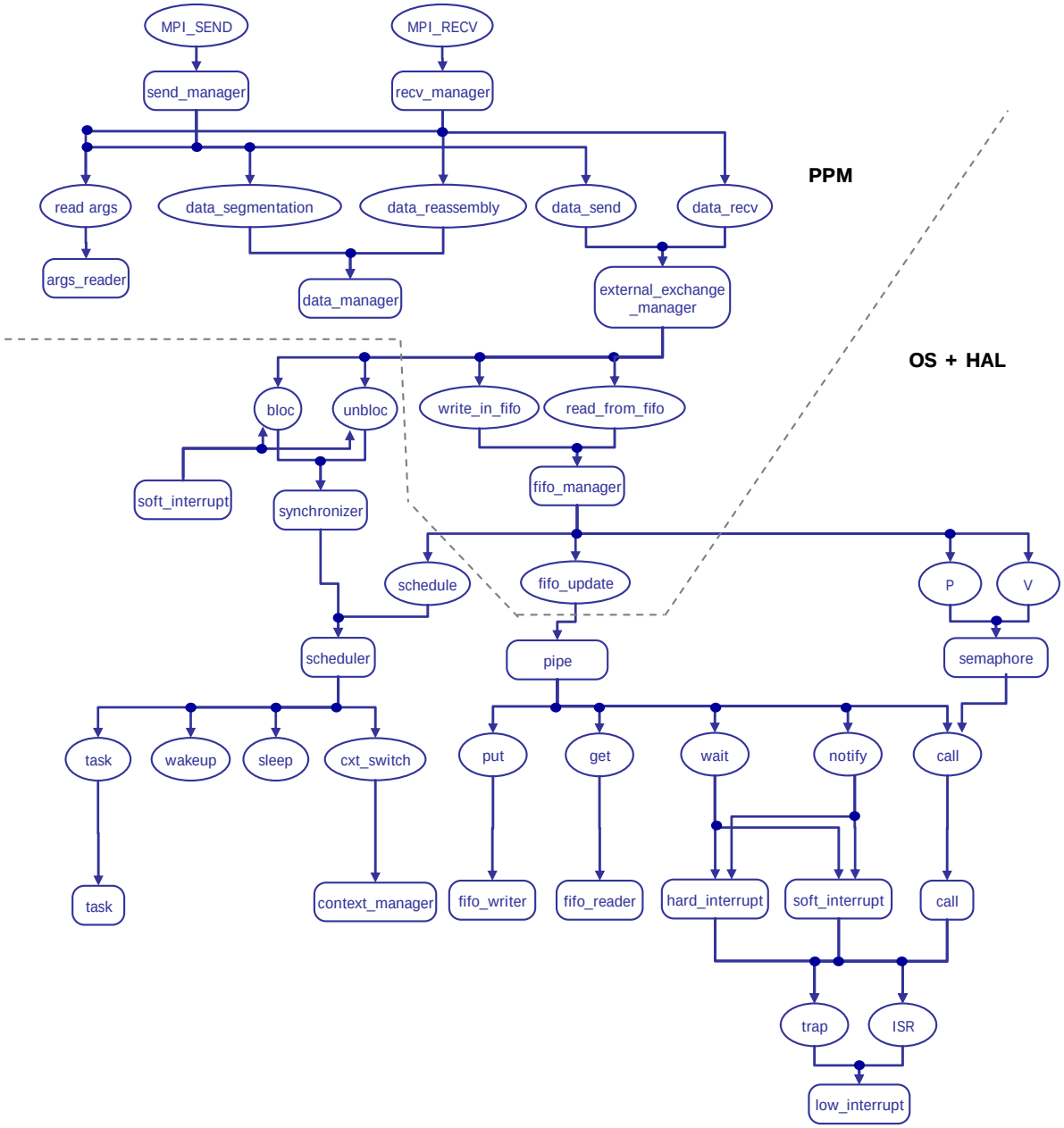


Figure V-15 : Graphe de dépendance de services pour les primitives MPI_SEND et MPI_RECV

Le tableau V-3 montre la taille du code de l'adaptation logicielle générée par l'outil ROSES, pour l'application OpenDivX, en terme de taille de l'exécutable (KB) et de lignes de code (LC).

Code	Maître	Esclave
MPI	3.4 KB, 923 LC	2.9 KB, 892 LC
μ-kernel	2.3 KB, 1069 LC	2.2 KB, 1065 LC
MP boot	0.7 KB, 379 LC	0.7 KB, 379 LC
Total	6.4 KB, 2371 LC	5.8 KB, 2336 LC

Tableau V-3 : Taille du code du l'adaptation logicielle générée

La spécification d'entrée de l'outil ROSES consistait en un ensemble de 30 fichiers contenant la spécification VADeL et le code parallèlisé utilisant des primitives MPI. La spécification VADeL contient en totale 533 paramètres. La difficulté majeure de cette étape est le nombre élevé de paramètres à fixer, qui peut induire à des confusions concernant leurs significations et donc à des bugs dans le code généré.

V.4.4 Débogage de l'interface logicielle/matérielle

Dans cette expérience, concevoir l'interface logicielle/matérielle consiste à développer le logiciel dépendant du matériel puisqu'il s'agit d'un raffinement 100% logiciel. Le HdS est ensuite compilé avec le code de l'application, puis téléchargé sur la plateforme du prototypage (ARM Integrator AP) pour y être exécuté. Lors de cette expérience, nous avons trouvé et fixé plusieurs bugs avant d'obtenir le code correct.

Classification des bugs

Afin d'investiguer les sources d'erreurs et de trouver une stratégie efficace pour le débogage de l'interface logicielle/matérielle, nous avons classé les bugs trouvés sur la plateforme prototypage comme présenté dans la figure V-16.

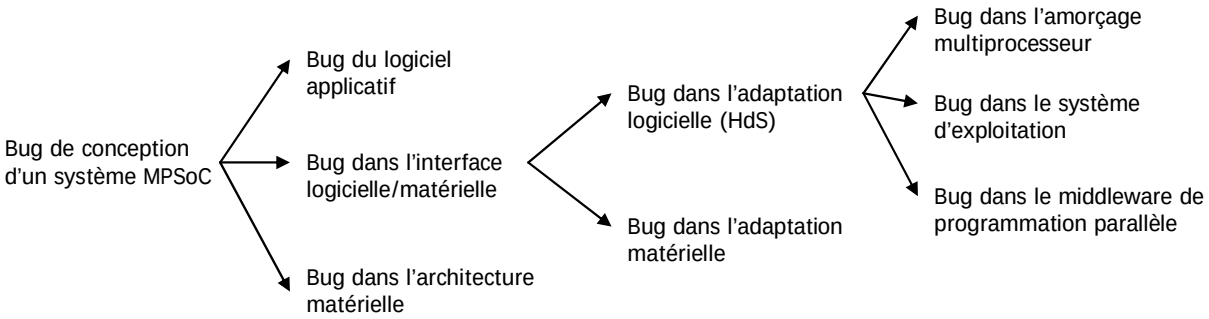


Figure V-16 : Classification des bugs de l'interface logicielle/matérielle

Les bugs dans la conception d'un système MPSoC sont divisés en bugs de l'application logicielle, bugs de l'interface logicielle/matérielle et bugs de l'architecture matérielle. Les bugs de l'application logicielle sont dus à une utilisation de modèle de simulation limitée à MPICH et MPI/SC pour le développement de l'application (y compris la phase de parallèlisation). Par exemple, un bug de dépassement de pile n'est pas détecté par une simulation de haut niveau basée sur MPICH ou MPI/SC car la pile n'est pas modélisée dans cette simulation. De plus, comme l'algorithme

d'encodage dépend des données, un tel bug ne peut être détecté qu'en utilisant une séquence de test spécifique. Donc une simulation d'une durée plus longue est requise ainsi qu'un banc de test plus étendu pour identifier tous les bugs sur la plateforme de prototypage. Nous classons aussi, comme bug d'application logicielle, un bug de librairie C pour manque de fonctions supportées (`malloc`).

Un bug de l'interface logicielle/matérielle peut être un bug du logiciel dépendant du matériel ou un bug de l'adaptation matérielle. Plusieurs types de bugs du logiciel dépendant du matériel existant. Ils peuvent être relatifs au boot multiprocesseur, au micro-noyau du système d'exploitation, ou au code de MPI. Un bug de l'interface matérielle peut être dû à une configuration ou à un accès incorrect à l'architecture matérielle, il résulte généralement d'une mauvaise compréhension de l'architecture matérielle par le concepteur. Par exemple une mauvaise configuration du plan mémoire pour assigner la valeur du registre de contrôle d'interruption.

Un bug de l'architecture matérielle est un bug de la conception matérielle traditionnelle qui inclut tous les bugs de la conception du sous-système (exemple bus, contrôleur d'interruptions, contrôleur DMA) et du réseau de communication (exemple : conception de réseau-sur-puce). Dans cette expérience, comme l'architecture matérielle (plateforme de prototypage) est fixe, les bugs de l'architecture matérielle ne sont pas considérés.

La couche HdS, la principale source des bugs

Le tableau V-4 montre quelques statistiques concernant les bugs trouvés. Pour chaque type de bug le tableau en présente un exemple, le pourcentage de son occurrence et une instance possible de la source du bug. On constate que 78 % des bugs trouvés sur la plateforme de prototypage sont des bugs relatifs à l'interface logicielle/matérielle.

Dans le tableau V-4, les bugs relatifs à l'environnement de conception représentent les différences entre les outils de conception des applications logicielles et les outils de conception des MPSoC. Par exemple la différence de gestion de types de données entre le compilateur GCC, utilisé pour le développement de l'application et le compilateur ARMCC utilisé pour la compilation sur le processeur cible.

Pour chaque bug identifié, on doit recompiler le code logiciel corrigé (dans le cas d'erreur de code logiciel) ou reconfigurer la plateforme (dans le cas d'une mauvaise valeur de configuration de la plateforme). Puis, on doit recharger le code exécutable de nouveau sur la plateforme pour une nouvelle exécution et un nouveau cycle de débogage.

Type	Exemple	%	Source du Bug

Application	Calcul dépendant des données.	5	Taille de la pile insuffisante.
	Bug au niveau de la bibliothèque C.	12	Manque dans la bibliothèque C utilisée.
Boot multiprocesseur	Le boot n'est pas synchronisé entre les différents processeurs.	12	Mauvaise configuration et initialisation de la plateforme.
Micro noyau	Perte de quelques interruptions.	13	Problème au niveau de la gestion des interruptions imbriquées au niveau des ISR.
	Mauvais niveau de priorité.	5	Mauvaise utilisation des niveaux d'interruption.
	Le changement de contexte ne fonctionne pas correctement.	5	Problème au niveau du multitâche dans la pile système/IRQ
Modèle de programmation parallèle	Valeur incorrecte du compteur de FIFO qui cause un deadlock.	13	Scénario de communication non implémenté.
Interface matérielle	Le résultat obtenu (vidéo compressée) n'est pas correct.	30	Mauvaise affectation dans le plan mémoire.
Environnement de conception	Exécution anormale d'une partie du code C.	5	Différente gestion des types de données par les différents compilateurs 'armcc' et 'gcc'

Tableau V-4 : Le détail des bugs trouvés

La figure suivante représente graphiquement le tableau V-4. On remarque nettement que la majorité des bugs sont relatifs au HdS de l'interface logicielle/matérielle.

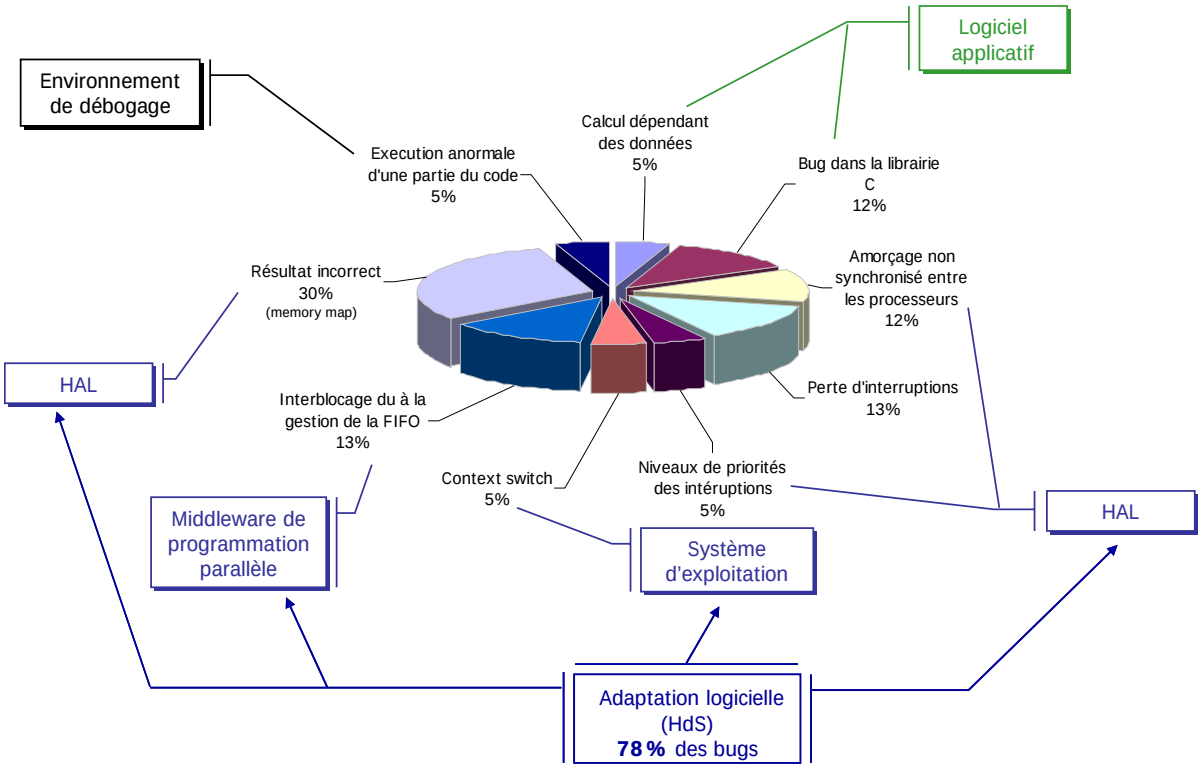


Figure V-17 : Représentation en camembert du pourcentage des bugs

Il a fallu une période de sept semaines pour une équipe de trois concepteurs pour identifier tous les bugs.

V.5 Leçons retenues

Dans ce paragraphe, nous présentons les principales leçons retenues de l'expérience de conception de l'encodeur OpenDivX en utilisant MPI pour l'architecture ARM Integrator.

V.5.1 Résultats de l'expérience de conception

Le code généré

Le tableau suivant montre la taille du code généré pour le processeur exécutant la tâche « maître » et un processeur implémentant une tâche « esclave ». Tous les chiffres présentés sont exprimés en octets. Pour chaque couche de la partie logicielle, trois types d'informations sont données. Le premier type donne la taille du code, le deuxième la taille des données en lecture/écriture et le dernier la taille des données en lecture seule. Ce tableau permet d'observer les différences de tailles de codes entre deux tâches n'utilisant pas le même nombre de port `MPI_SEND/MPI_RECV`. La tâche « maître » utilise trois ports alors que les tâches « esclaves » n'en utilisent qu'une. On peut noter que le code de la couche HAL ne varie pas. Ceci est tout à fait normal puisque la cible ne change pas. Pour la couche OS, une légère variation est observée. Cela correspond à l'ajout de deux files d'attente de tâches bloquées. Elles sont utilisées pour les blocages sur les deux ports supplémentaires utilisés par la tâche « maître ».

		Maître	Esclave
MPI	Taille de code	923	829
	Données en RW	3028	2772
	Données en RO	436	148
Système d'exploitation	Taille de code	443	439
	Données en RW	712	712
	Données en RO	88	48
HAL	Taille de code	1005	1005
	Données en RW	1884	1884
	Données en RO	84	84

Tableau V-5 : Taille du code généré pour le maître et un esclave

Les performances du code parallèlisé

Le Tableau V-6 contient des mesures réalisées sur le temps d'exécution des primitives `MPI_SEND` et `MPI_RECV`. Elles ont été réalisées en utilisant des compteurs intégrés dans la plateforme ARM Integrator AP. Le processeur utilisé est un ARM9 cadencé à 120 MHZ. Le tableau présente les temps mesurés pour plusieurs tailles de messages et plusieurs tailles de buffers de communication. Avec ces mesures, on observe que dès qu'un message a une taille supérieure au buffer de communication, le temps d'exécution augmente très rapidement. Ceci est dû au temps ajouté par le blocage de la tâche et son déblocage, ainsi que l'augmentation du nombre d'échanges pour le contrôle des FIFO.

	Taille du message en octets	**MPI_Send**				**MPI_Recv**			
Taille du buf. en octets		64	256	1024	4096	64	256	1024	4096
Temps d'exécution en micro secondes	16	113	114	114	113	194	193	193	193
	64	123	124	122	123	298	275	276	276
	256	3288	152	158	161	950	628	606	606
	1024	15287	5060	308	306	3535	2269	1950	1928
	4096	63218	24097	12250	897	13874	8817	7559	7243

Tableau V-6 : Temps d'exécution des primitives MPI_Send et MPI_Recv

Les performances observées sont plutôt mauvaises puisque l'on obtient une bande passante moyenne de 0,18 MB/s. Cette lenteur de communication est explicable par plusieurs raisons :

- Le contrôle des FIFO réalisé en logiciel est trop coûteux en échange de données de contrôle et interruptions.
- L'implémentation de l'arbitrage des interruptions en logiciel prend beaucoup de temps.

- Après vérification l'utilisation des `burst` n'était pas effective : le `burst` incrémental ne fonctionne pas sur la plateforme.

L'implémentation 100% logicielle est trop lente. Cela montre seulement qu'une implémentation avec autant de logiciel ne permet pas d'obtenir de bonnes performances. Si par exemple, les FIFO avaient été implémentées en matériel, une bonne partie des accès mémoires et interruptions auraient été évitées. Cette conclusion est peut-être un peu caricaturale puisque cette implémentation est extrême. Mais elle montre que la recherche d'un bon partitionnement logiciel matériel est nécessaire pour atteindre les contraintes imposées lors de la réalisation d'une application.

Cependant, le tableau V-6 démontre la faisabilité de l'exploration de l'architecture des interfaces logicielles/matérielles. Ces chiffres peuvent être utilisés pour annoter un modèle de simulation permettant de déterminer la meilleure taille du `buffer` de communication selon un compromis performances/taille mémoire requise.

V.5.2 Évaluation du choix de MPI

Cette expérience a été réalisée dans le cadre du projet ARCHiFLEX qui vise des architectures multiprocesseurs hétérogènes interconnectées via un réseau sur puce. La motivation du choix de MPI est son adéquation au type de l'architecture matérielle ciblée dans le projet, MPI permettant à des processeurs hétérogènes de communiquer d'une façon uniforme en évitant les problèmes issus des différents adressages de la mémoire, spécifiques à chaque processeur. Aussi, l'application d'encodage vidéo se prête bien à une parallèlisation de type SIMD (3 esclaves ayant le même code) et où la communication se fait par échange de messages.

MPI est donc bien adapté pour l'application et pour les architectures matérielles ciblées dans le projet ARCHIFLEX.

Cependant, l'architecture ciblée dans cette expérience (ARM Integrator, plateforme disponible d'où son choix) possède une caractéristique fondamentale qui la différencie des architectures ciblées dans le cadre du projet : l'interconnexion entre les processeurs se fait à travers un seul bus AMBA, d'où un échange de messages séquentiel entre les différents processeurs sur ce bus. Cette limitation de l'architecture a pour conséquence directe une dégradation des performances surtout que le volume de données à échanger entre les processeurs est élevé (des matrices représentant des images). Un choix de modèle de programmation parallèle basé sur la mémoire partagée, tel que [OpenMP] par exemple, aurait permis peut-être de meilleures performances, surtout que la plateforme offre des facilités de gestion de mémoire partagée (alias mémoire).

Le choix du modèle de programmation parallèle doit tenir compte de l'architecture matérielle si la performance du système est le principal critère de sélection (d'autres critères peuvent être le coût du système en temps de conception ou en surface). En effet, si on utilise un modèle de programmation parallèle de haut niveau basé sur une communication par échange de messages et

que l'architecture ciblée offre des facilités de mémoire partagée (concurrence d'accès à la mémoire gérée par matériel), le raffinement du modèle de programmation sur l'architecture engendre des copies mémoire supplémentaires qui dégradent les performances (au lieu d'utiliser directement la donnée dans la mémoire partagée on la copie vers une autre zone dans la même mémoire avant de l'utiliser).

Ceci peut paraître contradictoire avec le fait qu'une conception basée sur un modèle de programmation parallèle fait abstraction de l'architecture matérielle. Mais, pour des performances optimales, on a besoin de considérer en paire le modèle de programmation parallèle et l'architecture matérielle qui se prête le mieux à son raffinement.

V.5.3 Analyse du cycle de conception de l'application OpenDivX

Dans cette partie, nous analysons le cycle de conception de l'application OpenDivX, pour identifier les avantages et les limitations du flot proposé en terme de temps de conception.

Le tableau V-7 présente la durée de chaque étape de conception ainsi que la taille du code écrit ou généré. On remarque que pour cette expérience, le débogage sur la plateforme de prototypage a coûté plus de 58% du cycle de conception totale, en majorité à cause des bugs au niveau de l'interface logicielle/matérielle.

Étape de conception	Durée de l'étape de conception	Nombre de lignes de code
Parallèlisation et validation	3 semaines	1029, code écrit à la main
Spécification pour la génération du HdS	2 semaines	729, code écrit à la main
Génération du HdS	**5 minutes**	9336, code généré automatiquement
Débogage sur la plateforme de prototypage	**7 semaines**	40 bugs corrigés

Tableau V-7 : Analyse du cycle de la conception pour l'application OpenDivX

Ici, deux points méritent d'être signalés :

- Le coût de la conception du matériel n'est pas pris en considération puisqu'on a utilisé une plateforme de prototypage lors de cette expérience. Dans le cas où une conception de la partie matérielle a lieu, le coût du débogage de l'interface logicielle/matérielle peut devenir plus important que celui de cette expérience. En effet, plusieurs symptômes sont communs au matériel et au logiciel d'où une difficulté accrue pour identifier la source d'un bug et une période de temps plus longue pour le corriger. Puis, comme les travaux de [Verfaillie 03] le montrent, le débogage peut être toujours considéré comme l'étape la plus coûteuse en temps, même si la conception du matériel est prise en compte.

Le second point concerne la valeur ajoutée de la génération automatique du logiciel dépendant du matériel. L'outil ROSES a été utilisé pour la génération de 9379 lignes de code divisées comme suit :

- 2371 lignes de code pour la tâche « maître »,
- 2336 lignes de code pour chaque tâche « esclave ».

La génération du HdS a duré uniquement 5 minutes. C'est un gain énorme en temps de conception puisque la durée du développement manuel du HdS est estimée à 31 semaines ![2]

Chronologie du cycle de conception

Dans cette expérience, l'utilisation du modèle de programmation parallèle MPI a permis la parallèlisation de l'application en même temps que le développement des bibliothèques adéquates pour le raffinement du modèle sur l'architecture matérielle (figure V-18). La phase de débogage a commencé dès la cinquième semaine. La correction de chaque bug impliquait des modifications portant sur la spécification du HdS ou/et sur la bibliothèque des éléments du HdS, puis une nouvelle génération du code du HdS.

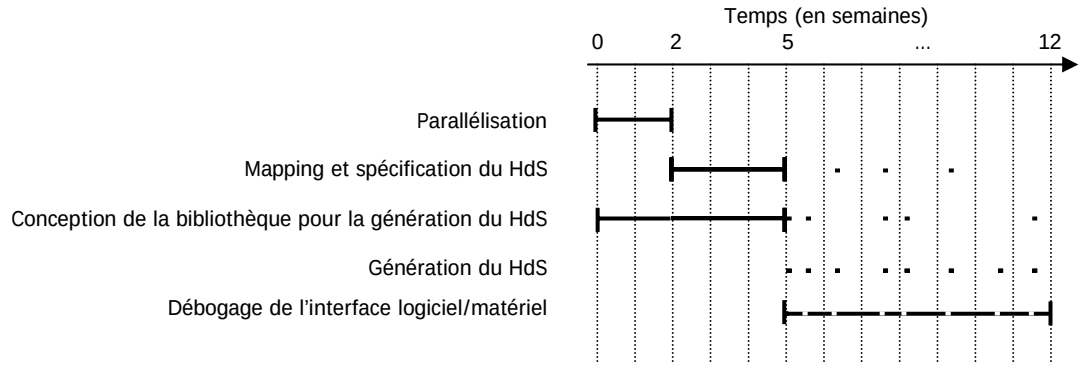


Figure V-18 : Chronologie des étapes de la conception de l'application OpenDivX

V.5.4 Le débogage, un goulet d'étranglement

Lors de l'analyse du cycle de conception, nous avons clairement identifié que le débogage est un goulet d'étranglement de la conception (58% du cycle totale de conception) [Youssef 04]. Dans ce paragraphe, nous présentons une explication de ce coût ainsi qu'une approche pour réduire le temps nécessaire au débogage.

Environnements de débogages utilisés

Dans la figure V-19, la pile de gauche montre l'architecture logicielle matérielle au niveau RTL. Dans cette expérience, on a quatre sous-systèmes processeurs. La pile du milieu, montre les modèles utilisés pour la validation du code de l'application parallèlisée. Il s'agit de MPICH ou du modèle MPI en SystemC que nous avons développé. Toutefois, ces deux modèles de simulation ne

[2] En supposant une production de 50 lignes de code par jour

permettent pas de valider le code du HdS car ils l'abstraient. La pile de droite montre le modèle de validation final de code logiciel total (code de l'application parallèlisée + HdS). Nous avons utilisé la plateforme de prototypage ARM Integrator pour la validation du code logiciel. On remarque alors, que tout le code du HdS ainsi que son interaction avec le code de l'application sont débogués pour la première fois directement sur la plateforme (sans utilisation de modèle de simulation). Ce débogage du HdS comme un tout complique l'identification de la source des bugs.

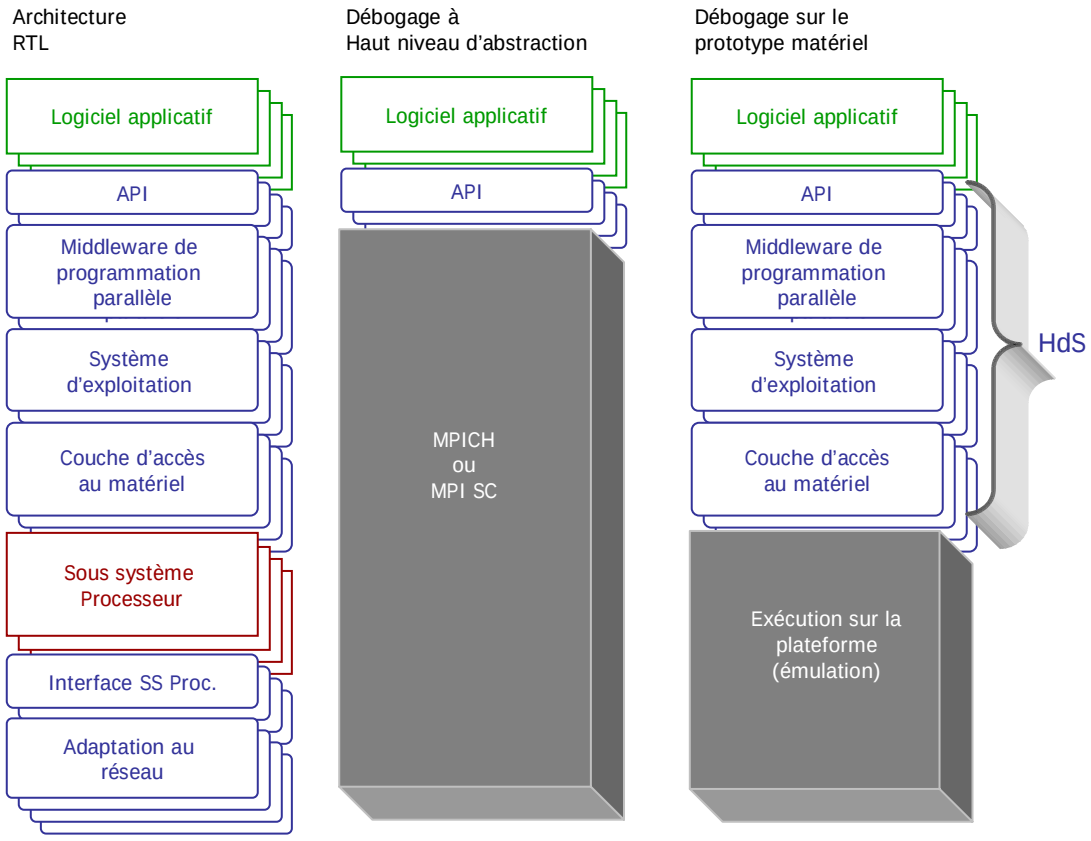


Figure V-19 : Modèles de simulation de l'architecture logicielle/matérielle

L'identification de la source d'un bug est un travail complexe

La figure V-20 explicite la difficulté pour l'identification des origines des bugs. La majeure partie des bugs est relative à l'interface logicielle/matérielle. Par exemple, si un bug est dû à une mauvaise gestion des FIFO au niveau de la couche middleware de programmation parallèle :

- L'origine du bug peut-être alors une valeur incorrecte dans la spécification du HdS (par exemple une valeur incorrecte pour l'adresse de la case mémoire contenant le nombre d'éléments actuellement dans la FIFO).
- L'origine du bug peut aussi se situer dans le code d'un composant de la bibliothèque qui gère la FIFO (par exemple, ne pas utiliser un sémaphore avant de modifier la variable contenant le nombre d'éléments actuellement dans la FIFO).

Comme tous les bugs ne se limitent pas à la gestion de la FIFO, d'autres exemples de bugs au niveau de l'interface logicielle/matérielle sont possibles. Pour chaque bug, deux sources

(spécification du HdS, bibliothèque de composants de HdS) sont potentiellement à son origine. D'où une vraie difficulté à identifier la source du bug.

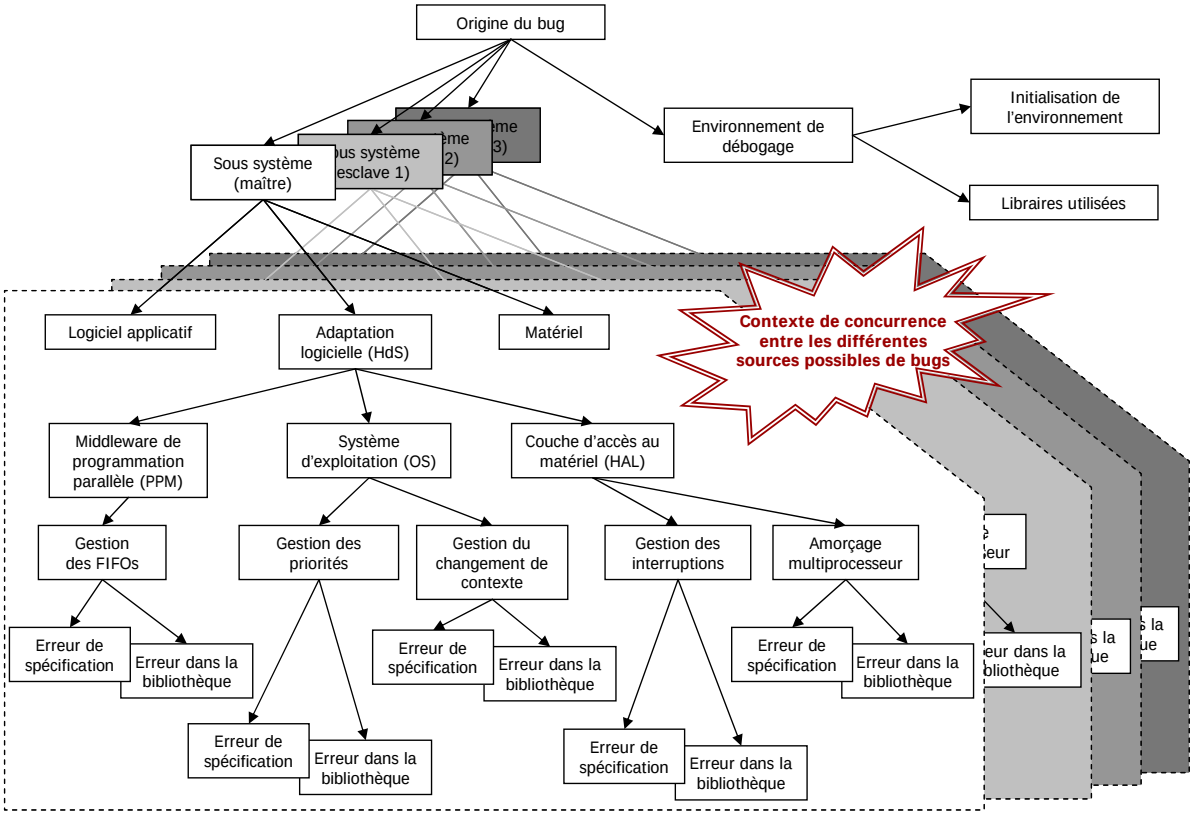


Figure V-20 : Sources éventuelles d'un bug

Cette difficulté est accentuée par deux facteurs :

- Tous les processeurs exécutent le code de l'application en concurrence, un bug au niveau du code d'un processeur (un processeur « esclave ») peut être dû à un disfonctionnement au niveau d'un autre processeur (processeur « maître »). Un problème de synchronisation est un exemple d'un tel bug.
- La limitation des outils de débogage livrés avec la plateforme : ces outils sont conçus pour un débogage monoprocesseur, ils ne permettent pas une exécution pas à pas du code de l'application sur les quatre processeurs (au niveau de l'architecture) en même temps. Ces outils ne supportent pas un débogage non-intrusif : le débogage est intrusif, il modifie le comportement de l'application et ne permet pas la régénération des bugs en utilisant un même scénario d'exécution.

Origines des bugs

- Les bugs issus d'une erreur au niveau de la spécification du logiciel dépendant du matériel sont dus à une mauvaise maîtrise de l'architecture matérielle : le concepteur doit gérer un grand nombre de paramètres (159 pour l'application OpenDivX) et connaître l'architecture matérielle ciblée dans ses moindres détails. Ce problème est d'autant plus significatif quand l'architecture matérielle et l'application logicielle deviennent plus complexes. Pour simplifier cette étape, un outil

d'aide pour l'exploration de l'espace des paramètres doit être fourni au concepteur, pour permettre de détecter des mauvaises combinaisons de paramètres et d'identifier les valeurs des paramètres qui donnent une meilleure performance pour un coût moins élevé.

- Les bugs issus d'une erreur au niveau de la bibliothèque sont dus au fait que la bibliothèque n'a pas été validée pour tous les scénarios d'exécution possibles avant son utilisation pour la conception de l'application OpenDivX. Une fois la bibliothèque mûre, l'utilisation d'un outil de génération automatique du logiciel dépendant du matériel permet d'accélérer significativement le cycle de conception (une écriture manuelle du HdS pourrait entraîner un nombre moins élevé de bugs mais elle rallongera considérablement le cycle de conception).

V.5.4.1 Stratégies pour limiter le coût et la durée de la phase de débogage

La plateforme de prototypage utilisée pour la conception de l'encodeur OpenDivX permet une simulation fiable et rapide puisqu'elle supporte une exécution précise au cycle près. Cependant, pour des systèmes plus complexes, une plateforme de prototypage spécifique à l'application peut être requise. La conception d'une telle plateforme requière une certaine période de temps incompatible avec la contrainte d'un délai de mise sur le marché réduit. C'est pourquoi on doit transférer le débogage de l'interface logicielle/matérielle de la plateforme de prototypage vers les environnements de débogages qui sont disponibles plutôt dans le cycle de conception.

Une stratégie possible est alors d'utiliser les environnements basés sur la cosimulation logicielle/ matérielle. Plusieurs bugs peuvent alors être identifiés avant que la plateforme de prototypage ne soit prête. Les deux types d'environnement de cosimulation suivant peuvent être utilisés pour le débogage de l'interface logicielle/matérielle [Clouard 02] :

- La cosimulation logicielle/matérielle approximative au niveau cycle. Cette cosimulation utilise un simulateur de jeu d'instructions qui encapsule un modèle de la mémoire du processeur et un modèle transactionnel du matériel. Elle permet une simulation rapide (~100Kcycles/sec). Cependant, ce n'est pas une simulation précise au niveau cycle puisque la mémoire du processeur est modélisée à l'intérieur du simulateur de jeu d'instructions. D'où le fait qu'une contention au niveau de la mémoire du processeur (par exemple entre un contrôleur DMA et un accès du processeur) n'est pas simulée au niveau cycle près [Kriaa 02] [Nicolescu 02-A].
- La cosimulation logicielle/matérielle précise au niveau cycle. Cette cosimulation utilise un simulateur de jeu d'instructions sans modèle de mémoire et un modèle transactionnel du matériel. Elle est plus lente que la première (seulement ~1Kcycles/sec). La différence réside dans le fait que la mémoire du processeur est modélisée comme un composant matériel, externe au simulateur du jeu d'instructions.

L'utilisation d'un simulateur de jeu d'instructions (ISS), dans les deux types d'environnement de cosimulation, permet le débogage du logiciel dépendant du matériel qui contient du code assembleur spécifique au processeur ciblé.

Pour exploiter la cosimulation logicielle/matérielle approximative au niveau cycle, nous classifions les bugs de l'interface logicielle/matérielle en bug purement fonctionnel et bug dépendant du temps.

- Les bugs purement fonctionnels sont ceux qui peuvent être détectés par une simulation, fonctionnelle ou temporelle, même si les informations temporelles ne sont pas correctes. Par exemple, les bugs de librairies C, les fonctions de base du logiciel dépendant du matériel comme la routine service d'interruption.
- Les bugs dépendants de temps sont ceux qui peuvent être détectés seulement si une séquence particulière ordonnée des événements a lieu lors de l'exécution du système. Des exemples sont un bug du démarrage multiprocesseur, la gestion des priorités imbriquées au niveau du système d'exploitation, etc... Ces derniers bugs peuvent être détectés en utilisant une cosimulation logicielle/matérielle au niveau cycle ou bien une émulation.

Toutefois, plusieurs bugs dépendants du temps peuvent aussi être détectés comme étant des bugs de type fonctionnel, si les bancs de tests sont développés afin de forcer un ordre spécifique des événements qui engendre le bug.

Pour exploiter une cosimulation logicielle/matérielle précise au niveau cycle, nous classifions tout d'abord les bancs de tests en deux types :

- Les bancs de tests qui ont une durée courte. Ils incluent les tests pour le démarrage multiprocesseur et les tests qui remplacent le code complexe des applications par un code plus simple, pour le débogage de l'interface logicielle/matérielle.
- Les bancs de tests qui ont une durée longue. Ils peuvent être utilisés pour tester la fiabilité du système après une longue période d'exécution, pour détecter par exemple des problèmes après plusieurs images.

Même si la cosimulation logicielle/matérielle précise au niveau cycle est lente, elle peut être exploitée pour exécuter les tests de courte durée. Cette cosimulation permettra d'identifier la majorité des bugs du logiciel dépendant du matériel, comme les bugs du modèle de programmation parallèle, du système d'exploitation, du démarrage multiprocesseur et de l'interface matérielle qui interagit avec le logiciel dépendant du matériel.

Après l'utilisation des deux environnements de cosimulation, quand la plateforme de prototypage est prête, elle peut-être utilisée pour exécuter les bancs de tests de longue durée.

Une autre stratégie peut être adoptée pour réduire la durée de la phase de débogage. Cette stratégie est basée sur l'utilisation des modèles de simulation. La figure suivante montre l'utilisation des modèles de simulation pour le débogage du logiciel dépendant du matériel [Yoo 03-A] [Yoo 03-B] [Bouchhima 06]. Ces modèles sont, de gauche à droite, de moins en moins abstrait. Le passage d'un modèle à un autre permet de déboguer le comportement de la nouvelle couche qu'il rend explicite. Ainsi par exemple, le troisième modèle de simulation à partir de la gauche permet

de trouver les bugs relatifs à la couche système d'exploitation. Ces bugs, comme détaillés dans le tableau V-4 représentent 40% des bugs. Les trois modèles du milieu, utilisés séquentiellement auraient pu permettre l'identification de jusqu'à 83 % des bugs de l'interface logicielle/matérielle dans le cas de l'application OpenDivX.

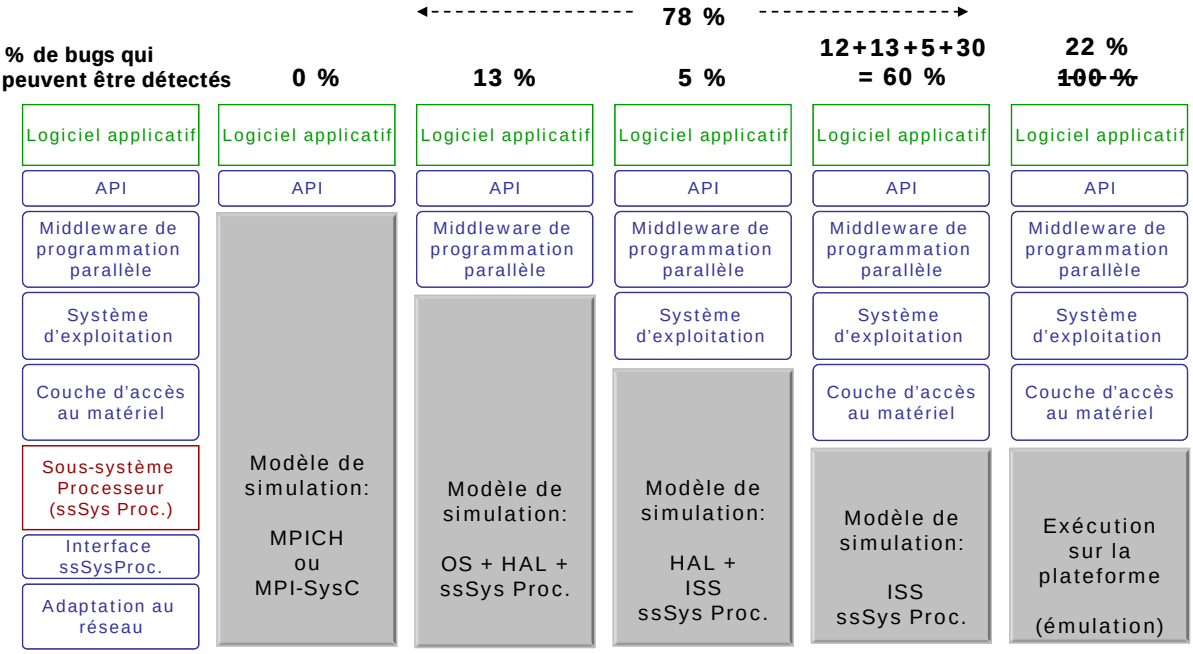


Figure V-21 : Utilisation des modèles de simulation pour le débogage des interfaces logicielles/matérielles

V.5.5 Perspectives

Cette expérience est le premier essai réalisé dans le groupe SLS, pour l'implémentation d'un encodeur vidéo sur une architecture MPSoC. Elle a permis de mettre en évidence la non adéquation de l'algorithme de l'application OpenDivX pour un environnement embarqué :

- Le code séquentiel est lourd pour un seul processeur : 227 secondes sont requises sur un processeur ARM9 fonctionnant à 28 MHZ pour encoder un flux vidéo d'une seconde contenant 20 images. Il faudrait alors une fréquence de 227x28 = 6356 MHZ sur un ARM9 pour atteindre le temps réel.
- Le code parallèlisé ne fait pas mieux. Il est même plus long (9 minutes pour un seconde de donnés vidéo) notamment à cause de problèmes de congestion et de non optimisation lors des accès au bus partagé AMBA. Certes, une meilleure utilisation du bus (activation du mode *burst*) permettra d'améliorer la performance vis-à-vis du code séquentiel à condition de gérer aussi les problèmes de contention.

Toutefois, nous pensons que le code de la partie calcul, non pas la partie communication, est le facteur le plus pénalisant pour atteindre les performances permettant un encodage en temps réel. Ce code étant écrit pour une exécution dans un environnement « standard » et non pas embarqué (multiples copies mémoires lors d'échanges de structures de données entre fonctions, utilisation

intensive d'opérations sur des flottants...). Porter certaines fonctions en matériel (DSP) permettrait d'alléger le coût de la partie calcul. Dans ce contexte, des outils comme le générateur d'unité de traitement automatique [GAUT] [Martin 93] peuvent être utilisés.

Nous avons aussi constaté lors de cette expérience la non-adéquation de l'architecture matérielle (un bus partagé) avec une caractéristique importante de l'application ciblée (grand besoin de bande passante). Cette non-adéquation a une grande influence négative sur les performances de l'application.

Cette expérience a révélée l'utilité de l'utilisation d'une API d'un modèle de programmation parallèle lors du développement du logiciel applicatif, à savoir (1) le développement conjoint du logiciel applicatif, de l'architecture cible et de l'interface logicielle/matérielle ; (2) facilité l'exploration de l'interface logicielle/matérielle et (3) la portabilité du logiciel applicatif vis-à-vis du matériel permettant le ciblage de différentes architectures matérielles.

Ce travail a servi alors pour le développement d'un nouvel encodeur vidéo [Bonaciu 06 – A] [Bonaciu 06 – B] :

- Nouveau code de calcul mieux adapté à un environnement embarqué.
- Utilisation d'un modèle de programmation parallèle (modèle interne au groupe, semblable à MPI) pour le développement du logiciel applicatif.
- Utilisation du flot ROSES pour le ciblage d'une plateforme matérielle dédiée et configurable [Han 04] adaptée à l'encodage vidéo.

V.6 Conclusion

Dans ce chapitre, une expérience de conception a été présentée et analysée. L'expérience consistait à développer l'encodeur vidéo OpenDivX en utilisant le modèle de programmation parallèle MPI pour cibler l'architecture matérielle ARM Integrator AP. L'expérience avait pour but de tester le flot présenté dans le chapitre précédent.

Dans le second paragraphe, l'application OpenDivX a été détaillée. Le principe de la compression par compensation de mouvement a été expliqué. Dans le troisième paragraphe, les choix des architectures logicielles/matérielles ont été discutés. L'architecture et les spécificités de la plateforme ARM Integrator AP ont été présentées. Dans le quatrième paragraphe, l'application des différentes étapes du flot, pour la conception de l'encodeur OpenDivX, a été exposée. Dans le cinquième paragraphe, une analyse de cette expérience, où les leçons retenues et les perspectives ont été exposées, a été proposée. Le flot utilisé a permis la réduction du temps de conception, grâce à la génération automatique d'interfaces logicielles/matérielles et à l'utilisation du modèle de programmation parallèle pour la conception du logiciel en parallèle avec celle du matériel. Le débogage des interfaces logicielles/matérielles a été identifié comme étant un goulet

d'étranglement dans le flot proposé. Des stratégies pour accélérer le débogage ont alors été explicitées.

Chapitre VI

Analyse d'une seconde experience de conception : Radio définie par logiciel en utilisant le modèle de programmation parallèle CORBA et une architecture matérielle spécifique

VI.1 Introduction

Dans le chapitre précédent, nous avons réalisé une expérience de conception, suivant le flot présenté dans le paragraphe IV.3, en utilisant le modèle de programmation MPI. Dans ce chapitre, nous présentons une seconde expérience de conception en utilisant le modèle de programmation parallèle [CORBA]. Ce travail s'inscrit dans le cadre du projet européen MERCED, qui a pour but de promouvoir la conception des applications à partir d'un ensemble de composants hétérogènes.

Notre contribution dans ce projet consiste à porter l'application sur l'architecture matérielle ciblée, en raffinant les primitives utilisées de l'API du modèle de programmation parallèle CORBA. Notre objectif est double : l'identification d'un éventuel goulet d'étranglement pour la conception d'un système MPSoC en utilisant le flot détaillé dans le paragraphe IV.3 (tout comme l'objectif de l'expérience précédente) et l'étude et la conception d'applications à partir d'un modèle de programmation parallèle plus abstrait que MPI.

Lors de la réalisation de ce travail, l'état d'avancement du projet MERCED ne nous permettait pas de disposer du code source du logiciel applicatif, ni d'une description détaillé de l'architecture matérielle. Nous nous sommes basés alors sur un modèle à gros grains de l'application et de l'architecture ciblée pour étudier le raffinement du modèle de programmation parallèle utilisé.

La figure VI-1 montre l'application du flot (proposé dans IV.3) lors de cette expérience de conception. L'expérience consiste à développer une radio défini en logiciel (SDR), en utilisant le modèle de programmation parallèle CORBA pour une architecture multiprocesseur spécifique.

Dans le cadre du projet MERCED, le travail de conception à réaliser porte sur tout le flot présenté dans la figure VI-1. Pour des contraintes de temps, seul les étapes de la partie grisée de la figure VI-1 ont été réalisées dans le cadre de cette thèse. Ces étapes sont la parallèlisation puis la transposition de l'application SDR et le débogage en haut niveau de l'application parallèlisée. Ces étapes ont requis une étude des primitives de l'API CORBA ainsi que le développement d'un modèle de simulation pour le débogage de haut niveau (il s'agit du modèle CORBA-SYSTEMC). L'étape de conception du logiciel dépendent du matériel a été entamée. Elle sera présentée comme perspective de ce travail.

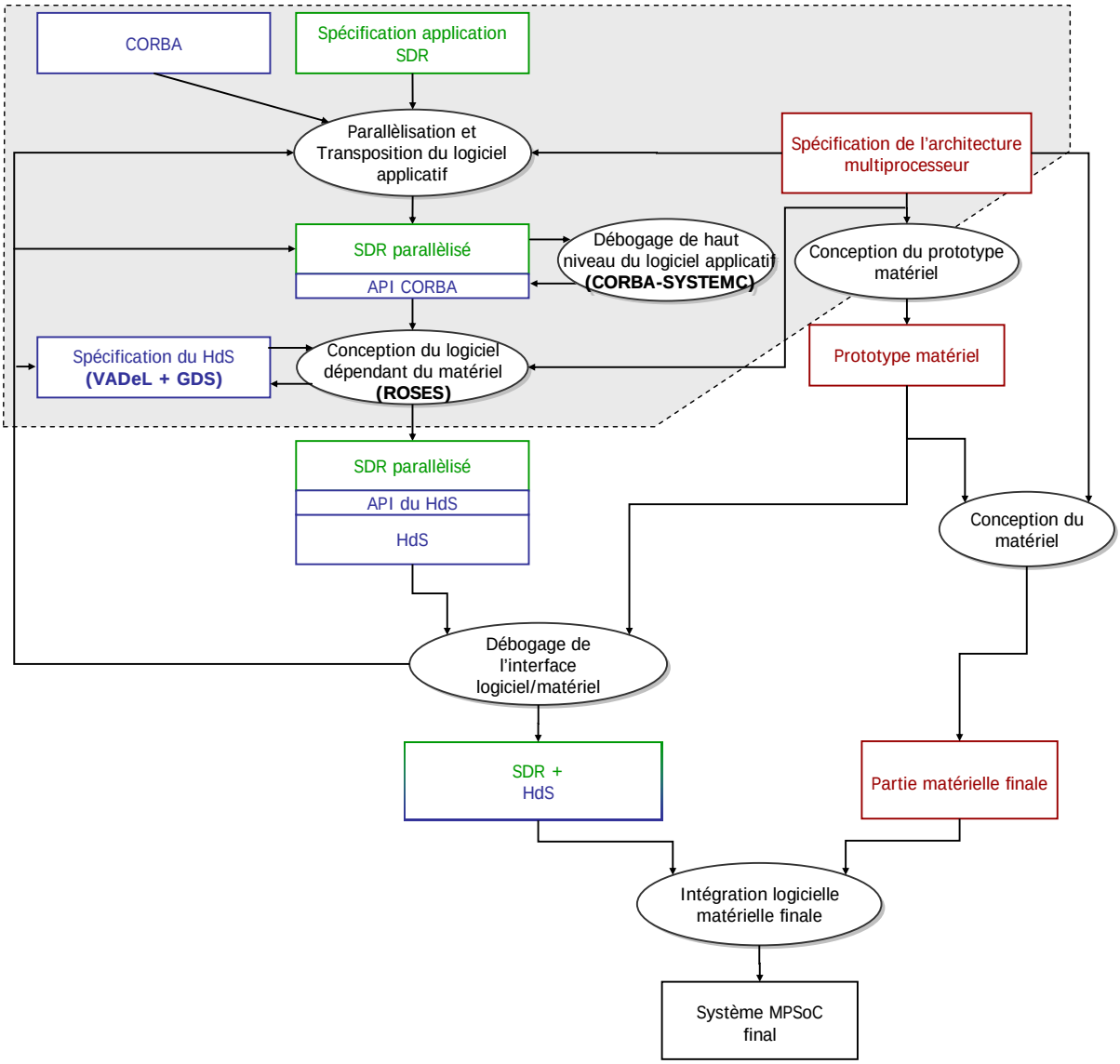


Figure VI-1 : Flot de conception pour l'application SDR

Le travail présenté dans ce chapitre a été réalisé en collaboration avec [Kriaa 05]. Dans le premier paragraphe nous présentons l'application radio définie par logiciel. Dans le second paragraphe, nous discutons des choix des architectures logicielles/matérielles. Nous y présentons le modèle de programmation parallèle CORBA. Dans le troisième paragraphe, nous détaillons le travail réalisé pour la conception des interfaces logicielles/matérielles basée sur le raffinement du modèle CORBA. Nous y discutons de la parallèlisation de l'application SDR en utilisant le modèle CORBA, puis nous y étudions les primitives de l'API CORBA en vue de leur raffinements, ensuite nous y proposons un raffinement des ces primitives sur un modèle SystemC. Dans le quatrième paragraphe nous présentons les perspectives de ce travail.

VI.2 Présentation de l'application radio définie par logiciel (SDR)

La radio définie par logiciel, notée SDR pour *software defined radio*, représente une radio complètement configurable qui peut être programmée par logiciel. C'est une technologie clé pour

la communication mobile sans fil. Le terme « radio logicielle » réfère à une collection de technologies et à une architecture standard qui permettent de séparer l'application (ondes radio) de la plateforme matérielle (figure VI-2). Il s'agit d'implémenter en logiciel le traitement spécifique à une forme d'onde donnée et en matériel le traitement commun à toutes les formes d'ondes radio. Grâce à cette approche, les nouvelles applications et les corrections des programmes existants peuvent être téléchargées sur un système existant. Ainsi, elle permet de surmonter la nécessité de faire des modifications matérielles pour chaque type d'onde radio.

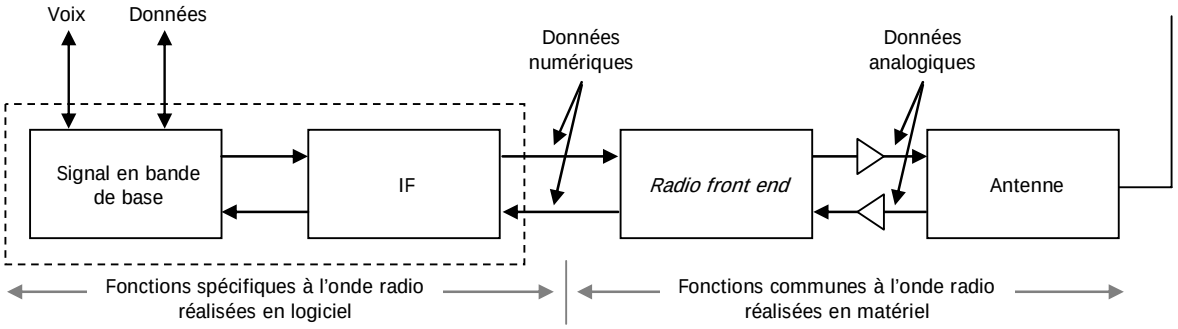


Figure VI-2 : Notion de radio logicielle – séparation entre le traitement logiciel spécifique à une forme d'onde donnée et le traitement matériel commun à toutes les formes d'ondes radio.

Une radio est généralement caractérisée par une variété de fonctions permettant la conversion de la voix et des données à partir/vers (d)'un signal de fréquence radio. Ces fonctions sont :

- Traitement de signal fréquence radio analogique (par exemple, amplification/désamplification, filtrage, etc...).
- Modulation et démodulation de l'onde avec une éventuelle correction d'erreur.
- Traitement du signal bande de base (*baseband*). Par exemple, routage vers les périphériques, ajout de protocoles, etc...

Dans une radio logicielle, les fonctions de modulation et de démodulation sont définies en logiciel. Du côté émetteur, ceci équivaut à un envoi d'onde générée comme des signaux numériques puis convertit en signaux analogiques. Du côté du récepteur, l'onde analogique reçue est extraite, convertie, démodulée pour produire un signal numérique.

La figure VI-3 présente les différentes fonctions qui ont lieu depuis une entrée de l'utilisateur (voix ou donnée) vers le signal fréquence radio produit et vice-versa. On y distingue trois parties :

- Partie noire, où figurent les éléments qui sont en contact avec le signal radio et doivent seulement supporter des informations cryptées,
- Partie rouge, où figurent les éléments qui sont en contact avec les utilisateurs via des interfaces de données et de voix. Ici l'information n'est plus cryptée.
- Partie de gestion du cryptage (appelée *infosec*). Elle crypte le signal en cas d'envoi, ou le décrypte en cas de réception. Cette partie définie distinctement la partie noire et la partie rouge.

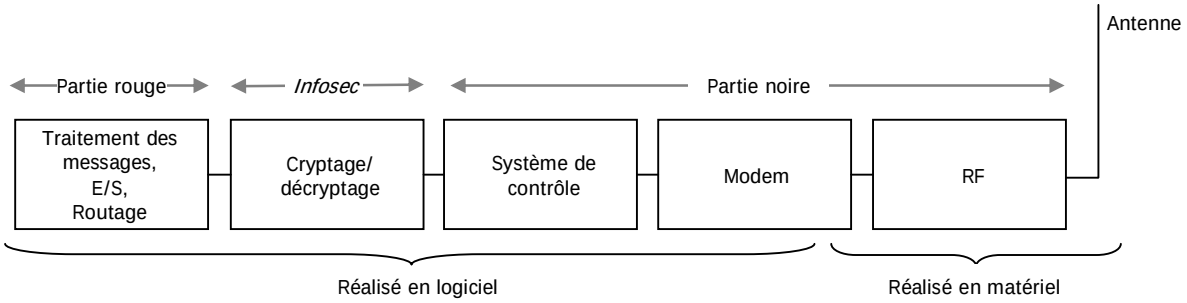


Figure VI-3 : Différentes fonctions d'un radio logicielle

Ce travail porte sur la partie noire, plus précisément, nous allons travailler sur la partie modem et système de contrôle de la partie noire.

VI.3 Choix des architecture logicielles/matérielles pour la conception de l'application SDR

Nous présentons dans ce chapitre les choix des architecteurs logicielles/matérielles pour la conception de l'application SDR. Ces choix englobent le choix du modèle de programmation parallèle et le choix de l'architecture matérielle ciblée. Nous présentons tout d'abord le modèle de programmation parallèle CORBA qui a été utilisé pour la conception de l'application SDR. Puis nous présentons brièvement l'architecture matérielle ciblée (présentation à gros grains) car cette architecture est encore en phase de développement.

VI.3.1 **Choix du modèle de programmation parallèle : CORBA**

Pour cette seconde expérience de conception, le modèle de programmation parallèle choisi pour la conception du logiciel applicatif est CORBA (voir paragraphe IV.4.2.4). Ce choix a été décidé par nos partenaires industriels du projet MERCED, car il permet la conception de l'application sous forme d'un ensemble de composants hétérogènes coopérant.

VI.3.1.1 Principe de fonctionnement d'une application CORBA

Le principe de fonctionnement d'une application décrite sous forme de composants CORBA suppose que tous les composants (clients et serveurs) se connectent à l'ORB. Du Côté serveur, l'enregistrement de tous les objets offrant les différents services doit se faire. Ainsi, lorsqu'un client requiert un service, l'ORB effectue une recherche sur l'ensemble des objets déjà enregistrés pour trouver l'objet offrant le service requis.

L'enregistrement des objets est un mécanisme complexe. Il assure la transparence de la recherche et de l'exécution au concepteur de l'application CORBA. Pour cette raison, du côté du serveur par exemple, le squelette défini un adaptateur portable d'objet noté « POA » (pour *portable object adaptor*). Un POA est alors associé à chaque objet, il gère l'exécution des services

appelés sur cet objet. D'autres services peuvent être requis pour assurer la transparence de la recherche de différents objets, tel que le service de nommage.

VI.3.2 Présentation de l'architecture matérielle ciblée

La figure VI-4 représente une vue simplifiée de l'architecture matérielle cible. Cette architecture est composée principalement par :

- Un FPGA.
- Deux processeurs de traitement de signal (de type [TI C5510] fonctionnant à 200 MHz).
- Un processeur (MPC 860 fonctionnant à 80 MHz).
- 160 Kmots (16 bits) de mémoire RAM (avec un adressage uniforme pour les données et le code).
- 2 M x 32 bits Flash
- 4 M x 32 bits RAM
- Un convertisseur analogique/numérique (FI = 400 KHz)
- Un convertisseur numérique/analogique (FI = 400 KHz)

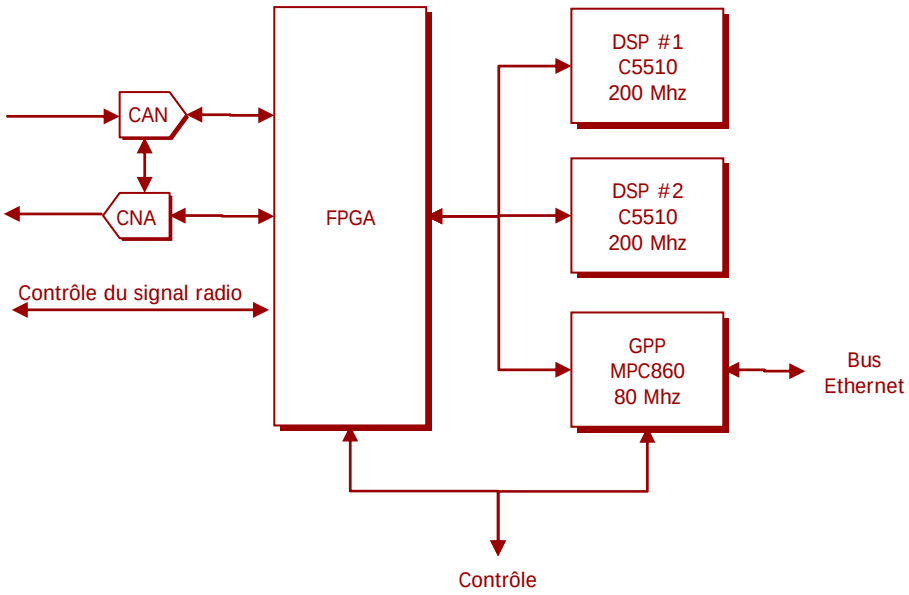


Figure VI-4 : Vue simplifiée de l'architecture matérielle ciblée

Cette architecture est développée spécifiquement aux besoins de la partie noire de la radio logicielle. Le premier DSP est chargé d'exécuter le traitement modem. Le second est dédié au traitement transmetteur et le FPGA joue le rôle d'un coprocesseur de communication. Le processeur GPP assure un traitement de contrôle.

VI.4 Conception des interfaces logicielles/matérielles pour l'application SDR

Nous présentons dans ce paragraphe le travail qui a été réalisé pour la conception des interfaces logicielles/matérielles de l'application SDR. Vu l'état d'avancement du projet MERCED, ce travail ne couvre que les premières étapes du flot (voir figure VI-1). Nous présentons tout d'abord la parallèlisation de l'application SDR et son implémentation sur l'environnement d'exécution [e*ORB]. Puis nous détaillons le raffinement des primitives de l'API CORBA, utilisées dans l'application SDR, pour une simulation de haut niveau basée sur SystemC. Les buts de cette simulation sont (1) étudier le raffinement du modèle CORBA sur une architecture matérielle et (2) avoir une estimation des performances en haut niveau permettant une exploration de l'espace de conception des interfaces logicielles/matérielles.

VI.4.1 **Parallèlisation de l'application SDR : Implémentation sur e*ORB**

VI.4.1.1 Présentation de e*ORB

e*ORB, pour *embedded object request broker*, est un modèle d'exécution de CORBA commercialisé par [PrimsTech]. C'est une implémentation de l'ORB CORBA pour les systèmes embarqués, compatible avec le standard [minimumCORBA] de [OMG] et avec l'architecture de communication logicielle définie par [JTRS] pour les radios logicielles. [PrismTech] présente e*ORB comme étant une implémentation (du ORB CORBA), de haute performance et de petite taille optimisée pour les systèmes à ressources réduites.

Nous avons utilisé e*ORB comme premier environnement d'exécution pour la conception du logiciel applicatif. Il nous permet d'écrire et de valider une application CORBA en utilisant l'implémentation de e*ORB des primitives de l'API CORBA.

VI.4.1.2 Parallèlisation de l'application SDR en utilisant le modèle d'exécution e*ORB

Lors de la réalisation de cette expérience, nous ne disposions pas du code de l'application SDR, car il était encore en cours de développement chez nos partenaires. Nous avons donc conçu un modèle à gros grains de l'application SDR. Notre objectif est l'étude du raffinement des primitives de l'API CORBA qui sont utilisées par ce modèle et qui seront réutilisée par le code final de l'application.

L'architecture matérielle ciblée est principalement composée de trois unités de traitement (2 DSP et 1 GPP). Nous avons alors développé une application SDR à base de trois composants CORBA, chaque composant simulera le fonctionnement d'une unité de traitement. Les composants sont `DSP_DTX`, `DSP_Modem` et le `GPP`. Dans ce modèle le composant `DSP_Modem` joue le rôle d'un serveur tandis que les deux autres composants jouent le rôle de clients qui requirent des services différents. La figure IV-5 – (a) montre le modèle de spécification en CORBA de l'application SDR.

Dans ce modèle, une interface CORBA écrite en IDL est associée à chaque composant. Ce modèle n'est pas un modèle exécutable. Le passage vers l'environnement d'exécution e*ORB se fait par la projection des interfaces IDL en squelette et en souches. Cette projection est réalisée automatiquement par l'outil de génération d'interfaces `idlcpp`, fournit avec l'environnement e*ORB. La figure IV-5 – (b) montre le modèle d'exécution de l'application SDR basée sur l'environnement d'exécution e*ORB. La génération des squelettes/souches fait apparaître des nouvelles primitives de l'API CORBA et des primitives de l'API du système d'exploitation. Les nouvelles primitives de l'API CORBA ne sont pas utilisées directement par les composants, mais c'est les squelettes/souches qui les utilisent. De même, les primitives de l'API OS ne sont pas directement utilisées par les composants pour préserver leur portabilité et leur indépendance vis-à-vis du système d'exploitation.

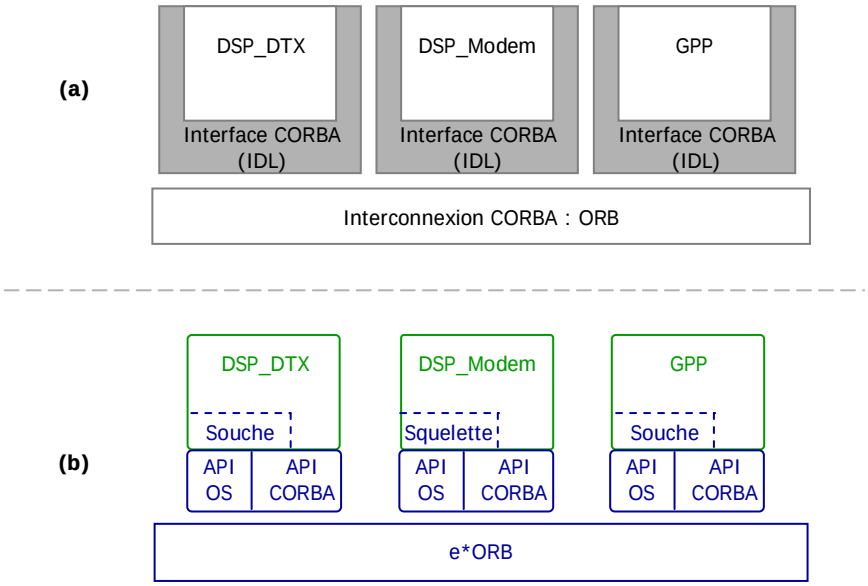


*Figure VI-5 : (a) Modèle de spécification et (b) Modèle d'exécution basé sur e*ORB de l'application SDR*

La figure VI-6 montre le code du fichier de description de l'interface entre le composant `DSP_DTX` et le composant `DSP_Modem`. Ici le composant `DSP_Modem` est le serveur, le composant DSP_DTX est le client. L'interface entre les deux composants porte sur les services « `decode` » et « `shutdown` ». Le premier service permet au `DSP_DTX` de récupérer le résultat d'un décodage paramétré par deux variables (« `var_one` » et « `var_two` ») et réalisé au niveau du `DSP_Modem`. Le second service est utilisé par le `DSP_DTX` pour informer `DSP_Modem` qu'il n'a plus besoin de ses services et que donc le serveur peut s'arrêter.

Cette interface est certes très abstraite par rapport à une application SDR réelle, mais elle est suffisante pour nous permettre d'identifier un sous ensemble de primitives de l'API CORBA qui doit être impérativement raffiné sur l'architecture pour permettre l'exécution d'une vraie application SDR.

```
1 :         interface DTX_Modem
2 :         {
3 :            short decode(in short var_one, in short var_two);
4 :            oneway void shutdown();
5 :         };
```

Figure VI-6 : Description en langage IDL de l'interface entre le DSP_DTX et DSP_Modem

VI.4.1.3 Les primitives de l'API CORBA utilisées pour la conception de l'application SDR

La conception du modèle d'exécution de l'application SDR basé sur l'environnement e*ORB nous a permis de fixer la liste des primitives de l'API CORBA que l'application utilise. Cette liste est donnée par la figure VI-7. La description de la fonction de chaque primitive de l'API, ainsi que de ses paramètres sont présentés dans l'annexe C.

```
1: CORBA::ORB_ptr CORBA::ORB_init ( int & argc,   char ** argv )

3: void CORBA::ORB::run (CORBA::Long timeout_msec)

5: CORBA::Object_ptr CORBA::ORB::resolve_initial_references ( const CORBA::ObjectId &
     id  EORB_ENV_ARGN )

8: CORBA::Object_ptr IORUtil::read ( const CORBA::ORB_ptr orb, char * name
     EORB_ENV_ARGN )

11: void IORUtil::write (const CORBA::ORB_ptr orb,  const CORBA::Object_ptr objref,
     char * name EORB_ENV_ARGN )

14: PortableServer::ObjectId_ptr PortableServer::POA::activate_object (
     PortableServer::Servant p_servant EORB_ENV_ARGN )

17: PortableServer::POA_ptr PortableServer::POA::_narrow ( CORBA::Object_ptr obj
     EORB_ENV_ARGN )

20: void CORBA::ORB::shutdown ( CORBA::Boolean wait_for_completion EORB_ENV_ARGN )

22: EORB::Codec::RequestId CORBA::Object::invoke_request ( CORBA::String opname,
     CORBA::ULong oplen,  CORBA::OperationMode mode,  EORB::Codec::Param * putParams,
     CORBA::ULong putElems,EORB::Codec::Param * getParams,   CORBA::ULong getElems
     EORB_ENV_ARGN )

27: void EORB::Codec::Request::get_args ( EORB::Codec::Param * params, CORBA::ULong
     elems EORB_ENV_ARGN )

30: void EORB::Codec::Request::put_args ( EORB::Codec::Param * params, CORBA::ULong
     elems EORB_ENV_ARGN )

33: CORBA::Boolean CORBA::Object::_is_a ( const char * logical_is_type EORB_ENV_ARGN )

35: CORBA::Object_ptr CORBA::Object::_duplicate ( CORBA::Object_ptr obj )
```

Figure VI-7 : Liste des primitives de l'API CORBA utilisées pour la conception de l'application SDR

VI.4.2 Raffinement des primitives de l'API CORBA pour une simulation en haut niveau basée sur SystemC : définition du modèle d'exécution CORBA-SystemC

Nous ne disposons pas de l'architecture matérielle finale, ni d'une description détaillée de celle-ci. Le raffinement des primitives de l'API CORBA utilisées par l'application SDR cible alors un modèle de simulation à gros grains de l'architecture matérielle, décrit en SystemC. Ce raffinement conduit à la définition d'un modèle d'exécution de CORBA en SystemC, que nous appelons CORBA-SystemC. Ce modèle est utile pour l'apport que SystemC rajoute par rapport à la simulation fonctionnelle basée sur e*ORB. En fait, le modèle CORBA-SystemC permet une simulation des composants à un haut niveau d'abstraction (niveau fonctionnel comme e*ORB). Puis, en ajoutant des annotations temporelles dans le code des composants, cette simulation nous permettra d'avoir une estimation de performance à un haut niveau d'abstraction [Bouchhima 04] [Posadas 04].

Nous présenterons dans la suite le modèle CORBA-SystemC plus en détails. Puis, le modèle d'exécution équivalent à e*ORB implémenté en SystemC. Dans la dernière partie, nous détaillerons l'implémentation des primitives de l'API CORBA en SystemC et nous présenterons un exemple d'utilisation de ces primitives.

VI.4.2.1 Présentation du modèle d'exécution CORBA-SystemC

Le modèle d'exécution CORBA-SystemC consiste à définir un environnement d'exécution autre que l'environnement d'exécution e*ORB pour les composants CORBA. Le nouvel environnement d'exécution doit contenir un raffinement (une implémentation) des primitives de l'API CORBA utilisées par l'application SDR. Le but étant de permettre l'exécution de l'application SDR sur le modèle CORBA-SystemC sans modifier son code source.

Pour développer le modèle CORBA-SystemC, le travail se décompose en deux parties :

- La première consiste à envelopper chaque composant CORBA par un module en SystemC, qu'on notera SC_CORBA_XX_MODULE (XX est à remplacer par le rôle du module, voir paragraphe VI.4.2.2). Ce module définira une interface pour communiquer avec d'autres modules et un contenu pour exécuter le comportement du composant CORBA. C'est à travers l'interface du module qu'on exécutera les différentes primitives de l'API CORBA.
- La deuxième partie est la définition de l'équivalent d'un ORB CORBA implémenté en SystemC. C'est l'environnement d'exécution dans le modèle d'exécution global [Kriaa 05]. Il est noté ORB_SYSTEMC. Il permettra l'interconnexion entre les différents modules SC_CORBA_XX_MODULE. C'est au niveau du ORB_SYSTEMC que les différentes primitives de l'API CORBA présentées dans VI.4.1.3 seront implémentées.

A la suite de cette partie, nous détaillons le modèle CORBA-SystemC. Nous présentons tout d'abord une description du module SC_CORBA_XX_MODULE puis une description de l'interconnexion ORB_SYSTEMC.

VI.4.2.2 Description du module SC_CORBA_XX_MODULE

Un module SC_CORBA_XX_MODULE est composé de deux parties, un contenu et une interface. Nous allons tout d'abord détailler l'interface puis le contenu.

a. Interface du module SC_CORBA_XX_MODULE

Dans un module SC_CORBA_XX_MODULE, le composant est enveloppé par une interface abstraite. Cette interface définie un ensemble de ports (internes et externes) et des services associés aux ports.

Les ports internes offrent les services définis par les primitives présentées dans le paragraphe VI.4.1.3. L'interface contient un seul port externe, il s'agit d'un port décrit en SystemC et nommé CORBA_Port.

Le port externe permettra l'interconnexion avec l'implémentation de l'ORB en SystemC, notée ORB_SYSTEMC. Sur chaque port externe, nous associons un paramètre dont la valeur sert pour différencier un composant offrant des services (paramètre = 0) d'un composant qui requière des services (paramètre = 1). Par exemple, dans un programme on peut écrire comme suit : CORBA_Port port(1).

Nous avons utilisé SystemC 2.0 pour implémenter l'interface d'un module SC_CORBA_XX_MODULE. En fait, dans SystemC 2.0 on peut associer un ensemble de primitives à un port. Cet ensemble de primitives est alors « l'interface » du port. Nous avons alors défini une « interface » notée CORBA_if qui définit l'ensemble des primitives de l'API CORBA décrit dans le paragraphe VI.4.1.3 et on l'a associée au port CORBA_Port. Quand un composant CORBA appelle une primitive de l'API CORBA, l'appel est transformé de façon implicite en un appel de la même primitive sur le port SystemC Corba_Port.

L'implémentation des primitives de l'API sur le port CORBA est définie et exécutée au niveau de l'interconnexion « ORB_SYSTEMC » (voir le paragraphe V.4.2.2).

b. Contenu du module SC_CORBA_XX_MODULE

Le contenu d'un module SC_CORBA_XX_MODULE implémente le composant CORBA ainsi que le squelette et/ou la souche. Nous avons défini trois types de modules. Les types dépendent du rôle du composant CORBA :

- Pour un module enveloppant un composant fournissant un service, nous avons défini le type de module SC_CORBA_SERVER_MODULE.
- Pour un module enveloppant un composant requérant un service, nous avons défini le type de module SC_CORBA_CLIENT_MODULE.
- Pour un module enveloppant un composant ayant un double rôle, nous avons défini le type de module SC_CORBA_CLISER_MODULE.

Pour l'application SDR seuls les deux premiers types ont tété utilisés. La figure VI-8 montre la description du module encapsulant le composant DSP_DTX.

Description des modules à rôle unique

Un module à rôle unique encapsule un seul composant CORBA. Nous définissons alors un processus SystemC qui implémente le composant CORBA et le squelette et/ou la souche.

La figure suivante présente le modèle SystemC d'une telle description.

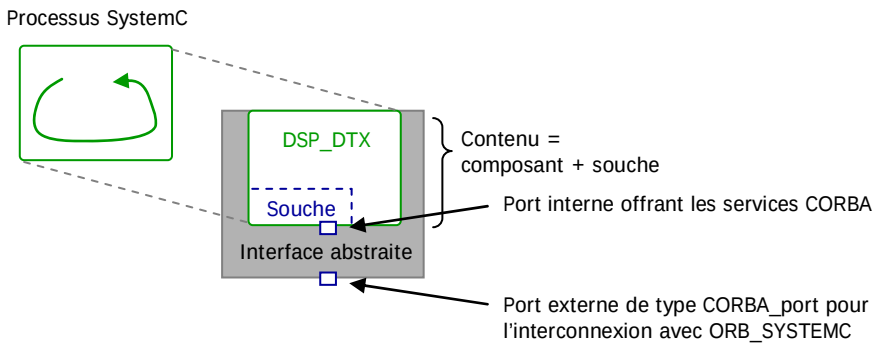


Figure VI-8 : *Module SC_CORBA_CLIENT_MODULE encapsulant le composant DSP_DTX*

Description des modules ayant double rôle

Un module à double rôle encapsule deux composants CORBA, chacun réalisant un rôle. Nous définissons alors deux processus SystemC différents, un processus pour le composant serveur et un autre pour le composant client. Chacun des composants opère sur une interface dédiée. L'interface du module comporte alors deux ports de type `CORBA_Port`. Le modèle SystemC d'un tel module est donné par la figure suivante.

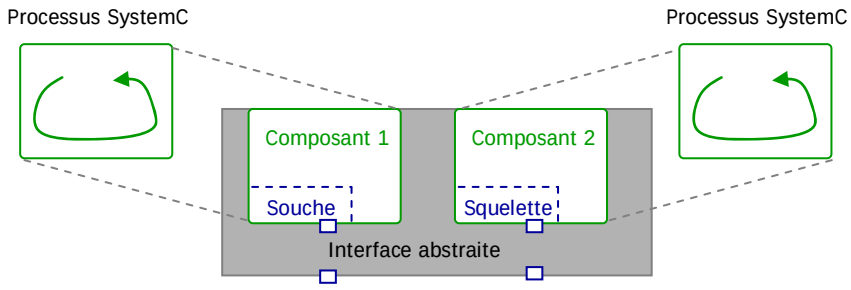


Figure VI-9 Description d'un module de type SC_CORBA_CLISER_MODULE en SystemC

VI.4.2.3 Description de l'interconnexion ORB_SYSTEMC

L'interconnexion `ORB_SYSTEMC` définie une implémentation de la politique de communication de l'ORB CORBA en utilisant SystemC. Elle représente, en fait, l'environnement d'exécution.

L'interconnexion ORB_SYSTEMC est définie comme étant un canal SystemC nommé ORB_Channel. Ce canal implémente toutes les primitives décrites par l'interface CORBA_if associée aux différents ports externes de type CORBA_port. Plusieurs modules SC_CORBA_XX_MODULE peuvent être connectés au canal à la fois.

L'interconnexion ORB_SYSTEMC contient des structures spécifiques qui permettent de garder la notion de transparence de communication entre les différents composants CORBA clients et serveurs.

L'implémentation des différentes primitives de l'interface CORBA_if dans l'interconnexion ORB_SYSTEMC est présentée dans la partie VI.4.2.5.

VI.4.2.4 Modèle d'exécution de l'application SDR basé sur CORBA-SystemC

La description du modèle d'exécution de l'application SDR est obtenue en assemblant les modules SC_CORBA_XX_MODULE définis ci-dessus autour de l'environnement d'exécution définis par ORB_SYSTEMC. La figure VI-10 montre le modèle d'exécution.

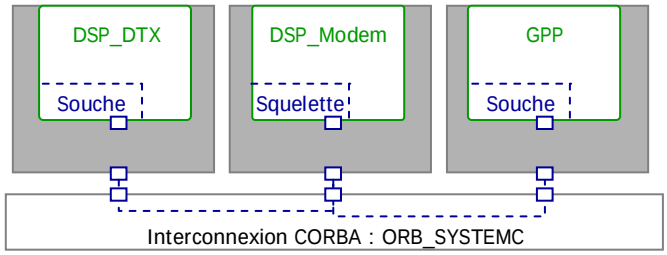


Figure VI-10 : Modèle d'exécution de l'application SDR basé sur CORBA-SystemC

Le code repris dans la figure VI-11 présente la description du modèle d'exécution de l'application SDR.

```
1:   int sc_main (int argc , char *argv[])
2:   {
3:    // déclaration des composants  CORBA et leurs conteneurs
4:    corba_proc dsp_modem, dsp_dtx, gpp;
5:    dsp_modem = main_DSP_Modem;
6:    dsp_dtx = main_DSP_DTX;
7:    gpp = main_GPP;
8:
9:    // Definie e*ORB_SystemC Channel
10:   ORB_Channel orb_sysc;
11:
12:   // Definition des modules SystemC_CORBA
13:   SC_CORBA_SERVER_MODULE  *Ser_mod;
14:   SC_CORBA_CLIENT_MODULE  *Cli_mod;
15:   SC_CORBA_CLIENT_MODULE  *Cli_mod1;
16:    …


19:   // Encapsulation des modules CORBA en SystemC
20:   Ser_mod = new SC_CORBA_SERVER_MODULE( "DSP_MODEM", dsp_modem);
21:   Cli_mod = new SC_CORBA_CLIENT_MODULE( "DSP_DTX", dsp_dtx);
22:   Cli_mod1= new SC_CORBA_CLIENT_MODULE( "GPP", gpp);

24:   // Interconnexion entre les différents modules et le canal
25:   (*(Ser_mod->P_server))(orb_sysc);
26:   (*(Cli_mod->P_client))(orb_sysc);
27:   (*(Cli_mod1->P_client))(orb_sysc);

28:   //Début de l'exécution
29:   sc_start(-1);
30:   return (0);
31:  }
```

Figure VI-11 : Code du modèle d'exécution global

VI.4.2.5 Implémentation des primitives de l'API CORBA utilisées par l'application SDR dans l'interconnexion ORB_SystemC

Avant d'implémenter les primitives de l'API CORBA utilisées par l'application SDR, nous les avons classées en trois types suivant la décomposition présentée dans la figure VI-12 :

- Le premier type correspond aux primitives des services de l'application définies dans l'interface décrite en IDL. Ces primitives sont utilisées par le concepteur de l'application CORBA pour faire des invocations de services distants.
- Le second type correspond aux primitives relatives à l'initialisation de l'ORB ou pour avoir des références sur des objets distants. Ce second type de primitives est aussi utilisé par le concepteur de l'application CORBA.
- Le troisième type correspond aux primitives de l'API CORBA utilisées pour implémenter les primitives relatives à l'interface. Ce type de primitives n'est pas utilisable directement par les composants CORBA, en fait, ces primitives sont cachées pour le concepteur de l'application CORBA, car utilisées uniquement par le squelette ou la souche.

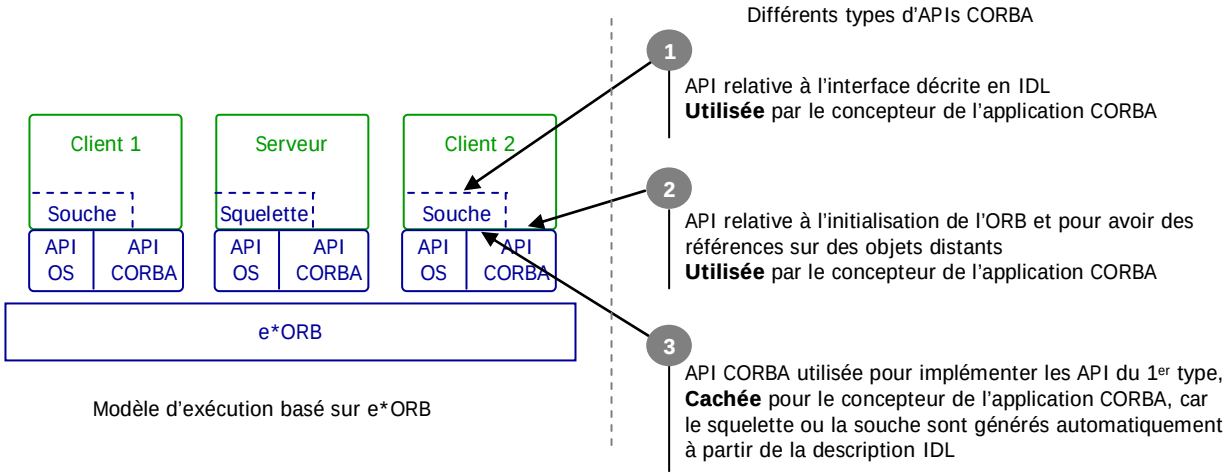


Figure VI-12 : Modèle d'exécution d'un système décrit en CORBA et différents types d'APIs CORBA

Les primitives de l'API du premier type sont implémentées dans le squelette/souche et utilisent les primitives des l'APIs du second et du troisième type. Nous allons donc juste nous intéresser aux APIs du second et du troisième type. Dans la suite nous qualifierons l'API du second type par API utilisateur car il l'utilise et celle du troisième type par API squelette/souche car elle est utilisée uniquement par squelette/souches.

Pour l'implémentation des primitives de ces deux types d'APIs, nous distinguons deux catégories. La première est celle de primitives de l'API de communication et la seconde est celle des primitives de l'API de cohérence. L'ensemble des primitives des APIs de la première catégorie représente le plus petit ensemble de primitives d'APIs CORBA que nous devons implémenter dans l'interconnexion ORB_SystemC pour assurer le fonctionnement d'un modèle d'exécution en SystemC équivalent à celui d'e*ORB. La deuxième catégorie est celle des primitives de l'API dont nous avons besoin pour permettre l'exécution des composants (avec squelette/souche) sans modification de leurs codes source.

Le tableau suivant présente la classification des APIs. Nous présentons les types en colonnes et les catégories en lignes.

Catégorie 1 : API de communication	
Primitives de type 2	Primitives de type 3

`PortableServer::ObjectId_ptr PortableServer::POA::activate_object()`
 	
`void CORBA::ORB::run ()`
	`EORB::Codec::RequestId CORBA::Object::invoke_request()`
	
	`void EORB::Codec::Request::get_args()`
	
	`void EORB::Codec::Request::put_args()`

Catégorie 2 :API de cohérence	
Primitives de type 2	Primitives de type 3

`CORBA::ORB_ptr CORBA::ORB_init()`
`CORBA::Object_ptr CORBA::ORB::resolve_initial_references()`
`PortableServer::POA_ptr PortableServer::POA::_narrow ()`
`CORBA::Object_ptr IORUtil::read()`
`void IORUtil::write()`
	`CORBA::Boolean CORBA::Object::_is_a()`
	`CORBA::Object_ptr CORBA::Object::_duplicate ()`

Tableau VI-1 : Classification des APIs CORBA

La figure VI-13 présente un diagramme de l'interaction -chronologiquement dans le temps- entre un ORB_SYSTEMC, un client (ici DSP_DTX) et un serveur (ici DSP_Modem). L'invocation des principaux APIs CORBA utilisées par le client ou le serveur y est illustrée. Dans cette figure, nous avons identifié onze étapes. Pour chaque étape, la description de ce que fait l'ORB_SYSTEMC (respectivement client ou serveur) est donnée dans un rectangle plein (respectivement ombrée).

Les onze étapes sont les suivantes : la première primitive est exécutée au niveau du serveur, il s'agit d'activer un objet offrant un service (1), l'objet et le serveur sont alors mémorisés dans l'ORB_SYSTEMC (2). La seconde primitive exécutée, est aussi du coté serveur, consiste à se mettre en attente des requêtes des clients (3) qui lui seront acheminés par l'ORB_SYSTEMC.

L'appel d'un service au niveau du composant client est transformé (via le squelette) en appel de la primitive `invoke_request` (5), les paramètres de cette primitive permettent d'identifier le service requis au niveau de l'ORB_SYSTEMC (6). Aussi, *via* `invoke_request`, les paramètres en entrée ainsi qu'une structure des paramètres en sortie sont aussi envoyés à l'ORB_SYSTEMC. Une fois la requête transférée au serveur par l'ORB_SYSTEMC, le serveur demande alors les différents paramètres en entrée du service via la primitive `get_args` (7). L'ORB_SYSTEMC lui transfère les paramètres en entrée (8) et le serveur peut donc exécuter le service requis par le client. Il envoi le résultat en utilisant `put_args` (9). L'ORB_SYSTEMC reçoit le résultat et le transfère au client qui a fait la requête (10). Finalement en (11), le client reçoit le résultat de sa requête de service.

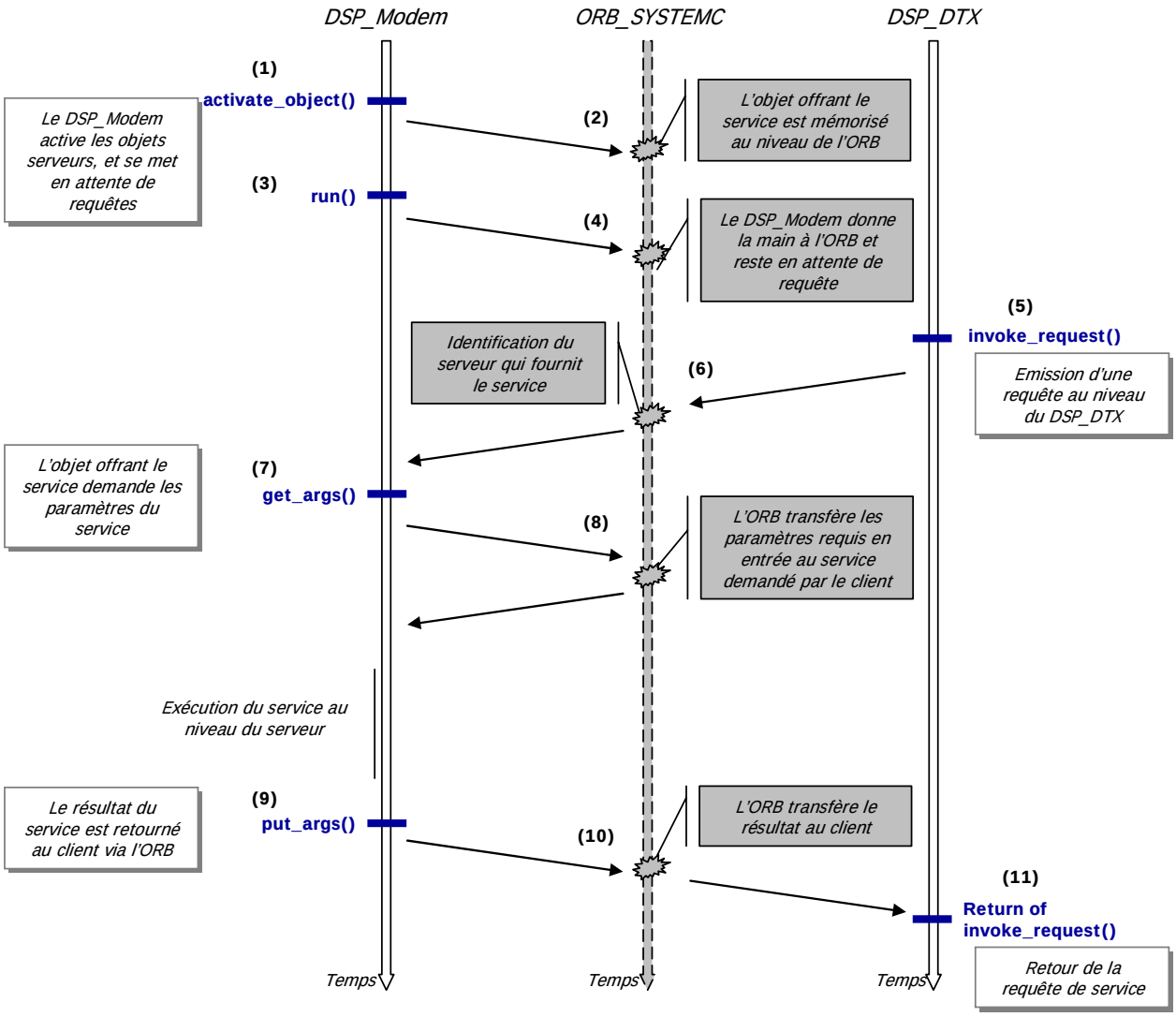


Figure VI-13 : Diagramme des interactions entre l'ORB_SYSTEMC, un serveur et un client

VI.5 Perspectives

Le travail réalisé pour l'application SDR a été décrit dans les paragraphes précédents. Il s'agissait principalement de l'étude de modèle de programmation CORBA et à son raffinement sur un modèle de simulation en SystemC. Dans ce paragraphe, nous présentons les perspectives de ce travail et nous présenterons quelques idées pour les réalisées.

VI.5.1 Choix de l'implémentation de l'ORB pour un modèle de l'architecture ciblée

Un modèle de l'architecture matérielle finale remplacera l'ORB_SYSTEMC. On aura alors besoin d'une implémentation de l'ORB sur ce modèle. Cette implémentation peut être réalisée en logiciel ou en matériel

- une implémentation logicielle permet une affection dynamique des tâches aux ressources de traitements, le rôle de l'ORB sera alors de décider du choix de l'affectation en vue de l'obtention

des meilleurs performances et la gestion des requêtes de services (communication, transfert des données et des résultats).

- Une implémentation en matérielle permet seulement une affectation statique des tâches, avec des performances meilleures (économie du temps de sélection de l'unité de traitement cible).

VI.5.2 Vers un raffinement automatique du modèle CORBA

Une autre perspective de ce travail est l'ajout, aux bibliothèques du flot ROSES, des composants adéquats (éléments et services) pour le raffinement automatique du modèle de programmation parallèle CORBA. La figure suivante présente une proposition d'un graphe de dépendance de services. Dans ce graphe, les ovales représentent les services. Les rectangles à coins arrondis représentent les éléments. La partie de gauche (respectivement de droite) représente la partie du graphe pour un composant serveur (respectivement client). La partie en bas permet la génération du système d'exploitation. Une proposition de paramètres relatifs à la configuration de l'élément gestionnaire d'appels distribué est aussi présentée.

Le graphe de dépendances de services présenté supporte uniquement les primitives `activate_object`, `get_args`, `put_args` et `invoke_request`. Par exemple, le service `invoke_request` a besoin de l'élément `request_processor` qui implémente un appel de requête en utilisant un service d'extraction d'arguments et un service d'appel de requête distribuée.

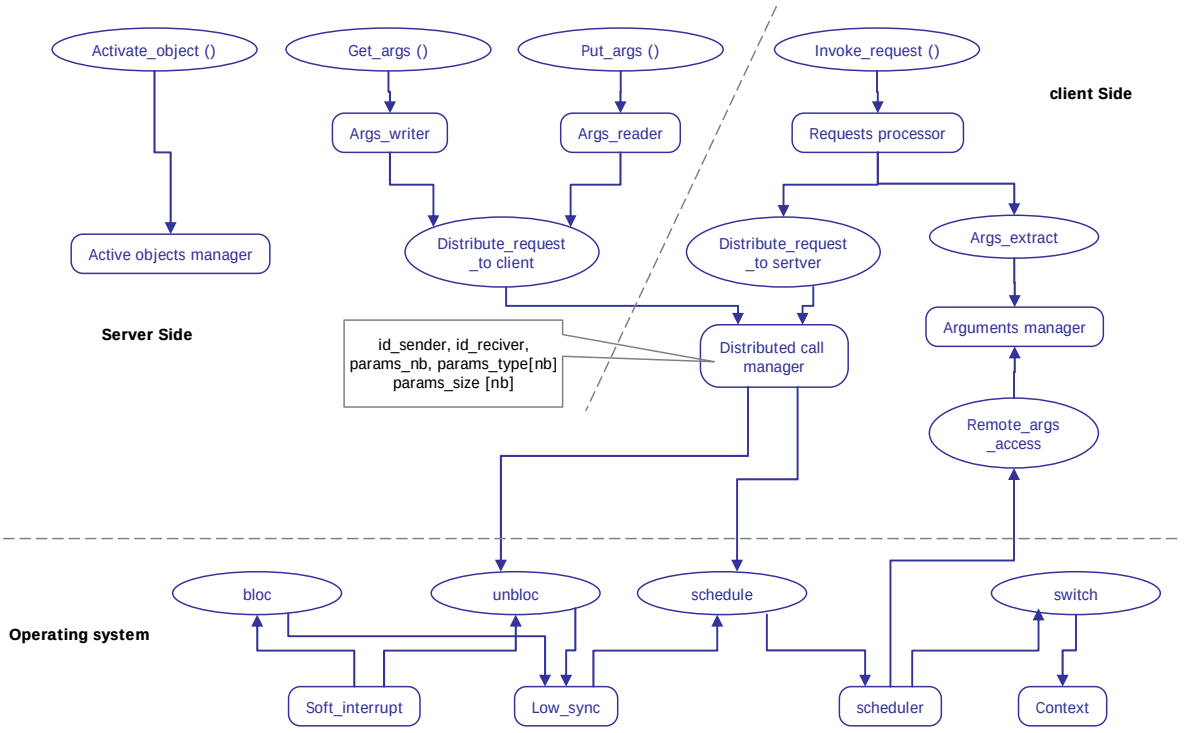


Figure VI-14 : Proposition d'une organisation de la bibliothèque pour le raffinement automatique du modèle Corba

VI.5.3 Estimation de performances

VI.5.3.1 Établissement des contraintes de performance en haut niveau d'abstraction

Le développement d'un modèle de simulation temporelle en SystemC permet d'avoir une estimation de la performance de l'application SDR à un très haut niveau d'abstraction. Les besoins de l'application en terme de qualité de service, combinés à une simulation temporelle permettent de fixer des limites pour le temps d'exécution du système final. Ces contraintes doivent être respectées lors du raffinement des primitives CORBA pour assurer la qualité de service requise.

VI.5.3.2 Estimation des performances à différents niveaux d'abstractions en utilisant un modèle abstrait de l'architecture

L'estimation des performances de l'application SDR peut être réalisée à différents niveaux d'abstraction, pour permettre un raffinement graduelle de modèle de programmation parallèle.

Annotation temporelle de l'application

La première étape est l'annotation temporelle de l'application SDR. Pour cette étape, nous proposons d'utiliser un ISS du processeur ciblé pour annoter des parties de calcul du code de l'application SDR avec leurs temps d'exécution. Cette information est primordiale pour l'estimation de performance de l'application à différents nivaux d'abstraction.

Estimation de performances au niveau système d'exploitation

Pour avoir une estimation de performance au niveau système d'exploitation, il faut (1) développer un modèle de simulation au niveau système d'exploitation, puis (2) raffiner les primitives CORBA pour ce modèle de simulation et (3) utiliser un code du logiciel applicatif déjà annoté. Cette estimation renseigne sur la qualité du raffinement des primitives CCORBA au niveau système d'exploitation, un bon raffinement respecte la qualité de service requise fixée dans VI.5.3.1 et réduit au maximum le coût de l'implémentation (la taille mémoire requise par exemple).

Estimation des performances au niveau de la couche HAL

Dans cette étape, il faut réduire encore le niveau d'abstraction et décider de l'implémentation du système d'exploitation. Il faut aussi disposer d'un modèle de simulation de l'architecture au niveau HAL (c'est-à-dire qui implémente l'API du niveau HAL). On peut alors avoir une information sur les performances d'un système très proche du système finale [Yoo 03-B]. Il s'agit en fait de l'avant dernier raffinement des primitives CORBA.

VI.6 Conclusion

Nous avons présenté dans ce chapitre une seconde expérience de conception. Il s'agissait de développer une radio définie par logiciel (SDR) en utilisant le modèle de programmation parallèle

CORBA et une architecture matérielle spécifique. L'application SDR a été présentée dans le second paragraphe. Dans le troisième paragraphe, nous avons exposé les choix des architectures logicielles/matérielles. Dans le quatrième paragraphe, nous avons discuté de la conception des interfaces logicielles/matérielles. Vu l'avancement du projet, nous avons raffiné le modèle CORBA vers un modèle en SystemC de l'architecture matérielle. La première étape était la conception d'un modèle exécutable de l'application SDR, en utilisant l'environnement d'exécution e*ORB. La seconde étape était l'étude et la classification des primitives de l'API CORBA utilisées dans ce modèle exécutable. La troisième étape était l'implémentation de ces primitives dans un nouvel environnement d'exécution basé sur SystemC. Dans le cinquième paragraphe nous avons détaillé quelques perspectives futures de ce travail.

Chapitre VII

Conclusions et perspectives

VII.1 Conclusions

Le besoin d'une méthodologie de conception basée sur une approche plus abstraite pour la conception des systèmes MPSoC est bien ressenti par le monde industriel et celui de la recherche. Dans cette optique, l'utilisation des modèles de programmation parallèle en vue de l'automatisation de leur raffinement sur différentes architectures matérielles est une approche prometteuse.

Dans le second chapitre, une étude de l'architecture des interfaces logicielles/matérielles pour les systèmes monopuces multiprocesseurs a été réalisée. Cette architecture a été décomposée en deux adaptations, l'adaptation logicielle et l'adaptation matérielle. L'adaptation logicielle a été structurée en couches pour des besoins de flexibilité et de portabilité. Les couches constituant l'adaptation logicielle étaient le middleware de programmation parallèle, le système d'exploitation et la couche d'accès au matériel. L'adaptation matérielle était constituée de l'adaptateur de sous-système processeur et de l'interface de communication avec le réseau.

Dans le troisième chapitre, les approches utilisées pour l'implémentation de l'adaptation logicielle ont été étudiées. Les deux approches classiques qui sont l'approche matérielle et l'approche logicielle ont été présentées et discutées. La partie la plus importante de l'adaptation logicielle étant le système d'exploitation, une étude des différentes architectures de système d'exploitation pour MPSoC a été exposée. La meilleure architecture semblait être l'architecture en réseau de système d'exploitation. Cette architecture consistait à concevoir un système d'exploitation spécifique à chaque unité d'exécution. Une analyse des solutions système d'exploitation commerciaux pour MPSoC a été réalisée. Il en a résulté que ces solutions étaient généralement conçues pour supporter un maximum d'applications et d'architectures ; elles offraient alors des fonctionnalités supplémentaires non requises par l'application. Les outils et méthodes d'adaptation de ces solutions aux besoins de l'application et aux caractéristiques de l'architecture matérielle ont été discutés. Aussi, dans ce chapitre l'approche de conception élaborée au groupe SLS a été présentée. Cette approche était basée sur la génération automatique de l'adaptation logicielle et de l'adaptation matérielle via un assemblage de composants.

Dans le quatrième chapitre, une analyse des modèles de programmation parallèle a été exposée. Aussi, un flot de conception de MPSoC basé sur l'utilisation des modèles de programmation parallèle a été présenté. Les modèles de programmation parallèle ont été classés suivant leur degré d'abstraction. Des modèles de programmation pour MPSoC déjà utilisés en industrie ont été détaillés. Le flot pour la conception des interfaces logicielles/matérielles des systèmes MPSoC proposé, se distinguait par une conception automatique de l'interface logicielle/matérielle, sans pour autant être limité à une classe d'architectures matérielles unique.

Dans le cinquième chapitre, une première expérience de conception a été détaillée. Il s'agissait de la conception d'un encodeur vidéo OpenDivX pour l'architecture matérielle multiprocesseur ARM Integrator, en utilisant le modèle de programmation parallèle MPI. Les différentes étapes du flot

utilisé pour la conception de cette application ont été présentées. La conception du logiciel dépendant du matériel a été explicitée. Cette expérience a montré que le débogage des interfaces logicielles/matérielles était un goulet d'étranglement pour la conception des systèmes monopuces, avec un coût en terme de pourcentage de cycle de conception avoisinant 60%. Des approches pour remédier à ce problème ont été présentées, notamment l'utilisation des modèles de simulation et de cosimulation. Cette expérience de conception a aussi montré l'importance de l'automatisation de la conception, avec une conception du logiciel dépendant du matériel qui ne durait que quelques minutes. L'utilisation du modèle de programmation parallèle MPI a permis de commencer la conception du logiciel applicatif et des composants du logiciel dépendant du matériel en même temps.

Dans le sixième chapitre, la première partie d'une expérience de conception d'une radio logicielle basée sur le modèle CORBA a été détaillée. Tout d'abord, le modèle CORBA a été détaillé, puis un modèle d'exécution de CORBA en SystemC a été présenté. Les perspectives de ce travail ont été présentées.

VII.2 Perspectives

L'objectif initial de ce travail consistait en l'abstraction des interfaces entre la partie logicielle et la partie matérielle d'un système MPSoC, en vue de leur génération automatique. Deux expériences ont été réalisées afin d'investiguer la complexité de la conception des interfaces logicielles/matérielles. Deux modèles de programmation parallèle, MPI et CORBA, de différents niveaux d'abstraction ont été utilisés.

Nous prévoyons que le niveau d'abstraction va encore être élevé dans le futur [Jerraya 05], notamment pour contrer la complexité croissante du logiciel et du matériel. Actuellement, le modèle au niveau transaction (TLM), est en train d'être accepté comme niveau d'abstraction plus élevé que le RTL. TLM sera utilisé pour aider la conception de l'interface logicielle/matérielle en accélérant la simulation de tout le système MPSoC [Clouard 02].

Les systèmes embarqués peuvent avoir des architectures matérielles hautement parallèles. Alors, après le niveau TLM, nous prévoyons que la tendance sera d'adapter les modèles de programmation parallèle comme un nouveau niveau d'abstraction pour la conception des systèmes MPSoC. Ces modèles sont requis pour décrire le parallèlisme interne à l'application. En terme de méthodologie de conception, l'utilisation des modèles de programmation parallèle signifie principalement que l'application va être décrite d'une façon neutre vis-à-vis de l'architecture, c'est-à-dire, elle n'est pas prédéterminée par rapport à une implémentation logicielle ou matérielle spécifique. En terme d'abstraction, il y a les six niveaux présentés dans le quatrième chapitre (voir le paragraphe IV.4.1) : interconnexion, synchronisation, communication, transposition, décomposition, concurrence.

La tendance pour l'utilisation des modèles de programmation pour la conception des systèmes MPSoC est de commencer par les modèles les plus explicites vers les moins explicites. Dans ce

contexte, les premiers modèles a être utilisés pour les systèmes MPSoC seront des modèles comme l'interface d'échange de messages, MPI, ou l'interface pour la communication par mémoire partagée, OpenMP. En terme d'abstraction, ces modèles fournissent une abstraction de l'interconnexion tout en fixant les détails de synchronisation, de communication, de transposition, de décomposition et de concurrence. En terme d'élévation du niveau d'abstraction, ces modèles présentent une continuation intéressante, puisque le niveau TLM consiste principalement en une abstraction de l'interface et donc l'abstraction prochaine est celle de l'interconnexion. L'étape suivante est d'abstraire le protocole de communication en adoptant des modèles comme SDL. Ici, on notera que de tels modèles existaient déjà depuis des années dans d'autres domaines d'applications comme les télécommunications. Certains travaux ont essayé d'appliquer des modèles comme SDL [Daveau 01] pour la conception des SoC. Cependant, l'adoption de tels modèles pour la conception des MPSoC se fera dans la durée : les concepteurs doivent tout d'abord bien maîtriser les technologies clé comme les interfaces logicielles/matérielles, le réseau sur puce, etc... Telles technologies permettront le développement des outils clés d'aide à la conception à base de ces modèles comme le partitionnement et la transposition automatique des tâches.

ANNEXES

VIII.1 Annexe A : Expérience de conception d'une adaptation matérielle

Cette expérience rentre dans le cadre de l'implémentation en matériel d'un gestionnaire d'entrées/sorties. Ce gestionnaire sera utilisé pour l'implémentation d'une version améliorée de l'application d'encodage vidéo [OpenDivX] sur la plateforme [ARM Integrator AP].

Pour les besoins de cette application, trois modules sont implémentés en matériel (voir figure VIII-1). Ces modules sont : un module qui lit un flux vidéo en entrée, un module qui écrit le résultat de compression vidéo et un troisième qui gère la communication entre les deux premier et les quatre processeurs de la plateforme ARM Integrator AP. Les processeurs exécutent le logiciel qui réalise la compression.

L'adaptation matérielle qui a été conçue est notée « adaptateur AMBA » dans la partie grisée de la figure VIII-1. Elle réalise un arbitrage pour l'accès au bus AMBA, entre les trois modules matériels. L'arbitrage d'accès au bus est nécessaire car la plateforme ne supporte qu'un seul composant connecté en tant que « maître » sur le bus AMBA à partir du FPGA. Hors, les trois composants ont un comportement « maître » vis-à-vis di bus AMBA. : les modules input et output accèdent au bus AMBA pour communiquer avec les ports série (UART). Le contrôleur De communication accède au bus quand à lui pour envoyer/recevoir des données vers/de chaque processeur.

La conception de cette adaptation matérielle est complexe car les choix des priorités entre les différents modules lors de l'arbitrage de l'accès au bus affectent directement les performances de l'application.

Lors de cette expérience de conception d'une adaptation matérielle, nous avons ressenti la complexité et l'importance de la phase de débogage et de validation de l'adaptation matérielle avant de la transposer sur le FPGA cible. En fait, la visibilité dont nous disposons de l'adaptation matérielle une fois transposée sur le FPGA se résume à des *leds*. Il faut avoir préalablement configuré l'automate de l'adaptateur AMBA conçu pour qu'il affiche sur les *leds* son état actuel.

D'une façon plus générale, les difficultés et les bugs de conception que nous avons rencontré lors de cette expérience mettent en évidence la difficulté et la complexité de la conception des interfaces logicielles/matérielles : les concepteurs doivent avoir des compétences pluridisciplinaires, notamment dans notre cas, relatives à la fois aux techniques de conception du logiciel et du matériel.

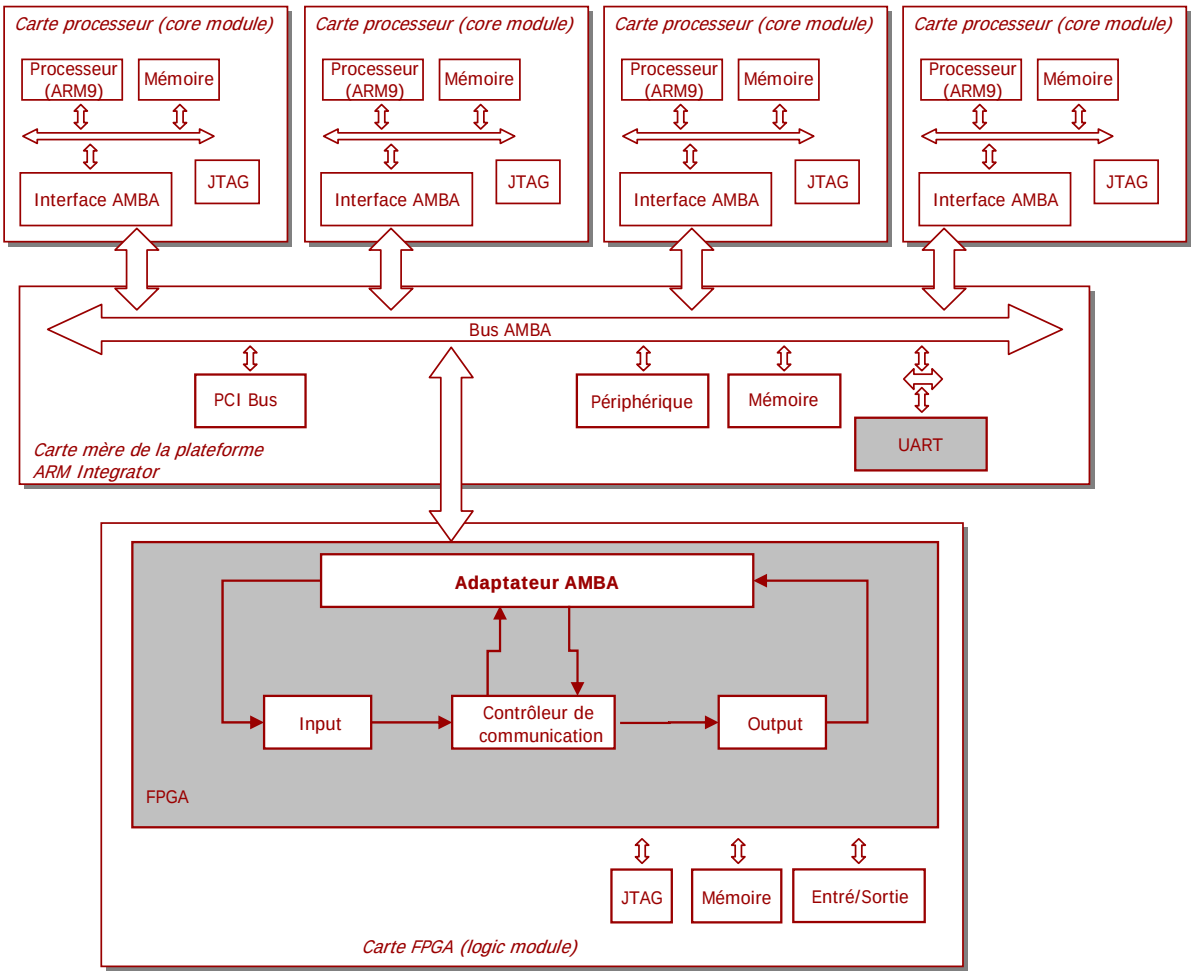


Figure VIII-1 : Gestionnaire d'entrées/sorties dans la plateforme ARM IntegratorAP

VIII.2 Annexe B : Présentation détaillée des primitives MPI_SEND et MPI_RECV

Envoi standard de message : MPI_SEND

L'envoi standard consiste simplement à copier le message en dehors de son emplacement source. Il peut être décliné en deux versions : une version avec buffer et une version sans buffer :

- Pour la version sans buffer, le message est envoyé directement au processus destinataire. Les données ne sont copiées que si le récepteur est prêt. S'il ne l'est pas, le processus émetteur est bloqué.
- Pour la version avec buffer, le message est copié dans un buffer intermédiaire de communication. La primitive `MPI_SEND` standard avec buffer est donc indépendante de l'appel de la primitive `MPI_RECV`. Si le buffer intermédiaire de communication n'est pas suffisant pour stocker l'intégralité du message, le processus émetteur est bloqué en attente de sa libération.

Le comportement des deux versions de l'envoi standard de messages peut être représenté par les graphes de transition de la figure VIII-2.

La syntaxe d'un appel MPI_SEND est la suivante :

`MPI_SEND(buf, count, datatype, dest, tag, comm)` , avec

- `Buf` : adresse du buffer destination
- `Count` : nombre d'éléments constituant le message à recevoir
- `Datatype` : type des éléments
- `Dest` : identificateur du processus récepteur
- `Tag` : identificateur unique du message dans le `comm`
- Comm : identificateur du « communicator » (il représente un ensemble de processus numérotés auxquels on attache des modes de communications).

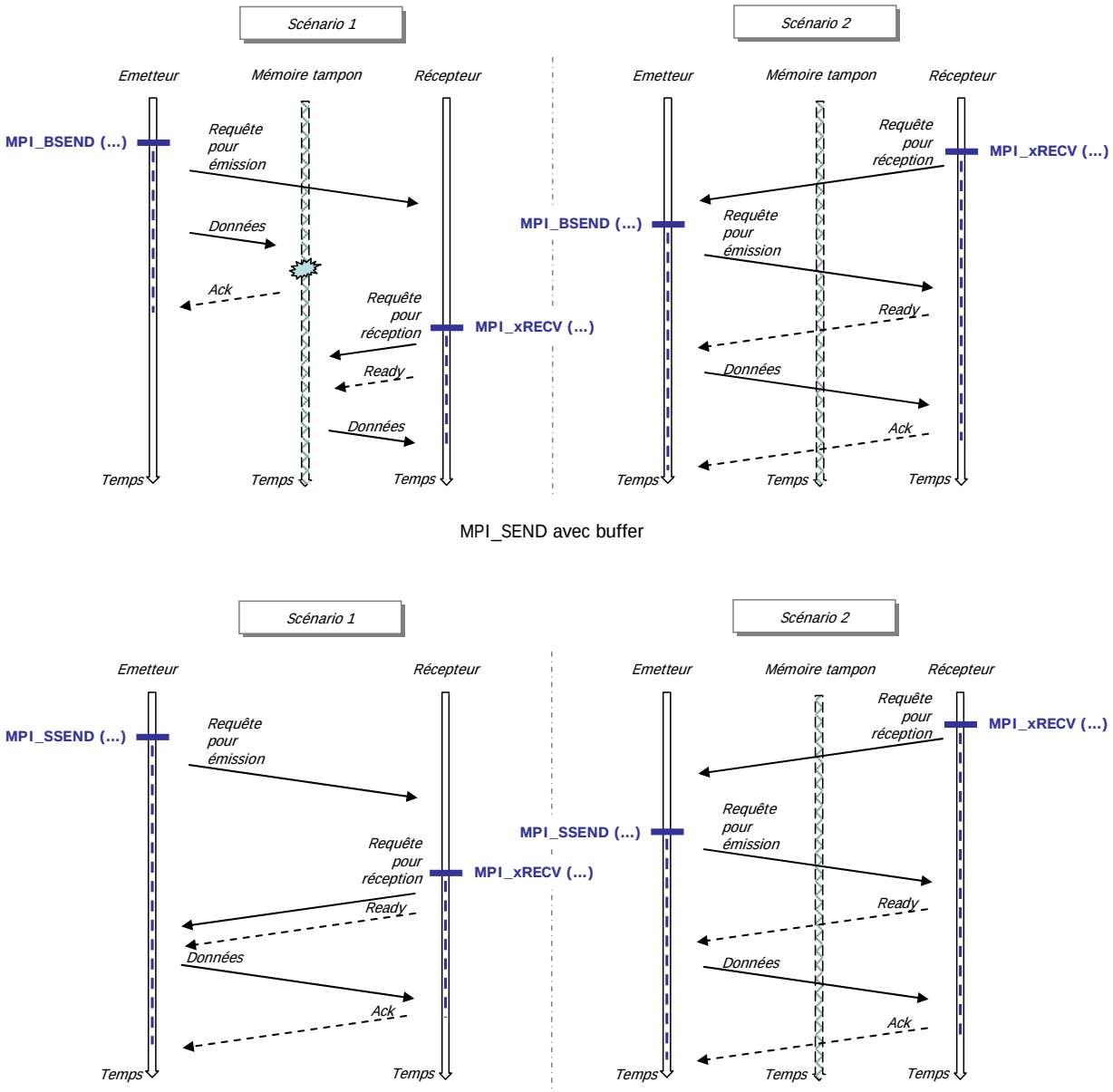


Figure VIII-2 : Différents graphes de transition de MPI_SEND

Réception de messages : MPI_RECV

`MPI_RECV` est utilisé pour recevoir les messages envoyés par `MPI_SEND`, `MPI_BSEND`, `MPI_RSEND`, `MPI_SSEND`. L'appel `MPI_RECV` peut être fait indépendamment de l'appel du `MPI_SEND`. Si le processus émetteur n'est pas prêt pour envoyer le message, le processus récepteur est bloqué.

La syntaxe d'appel de la primitive `MPI_RECV` est la suivante :

`MPI_RECV(buf, count, datatype, source, tag, comm) , avec`

`Source` : identificateur du processus émetteur, les autres champs ont la même signification que pour `MPI_SEND`.

VIII.3 Annexe C : Spécification VADeL de OpenDivX

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <vadel.h>
#include "../appli_types.h"
#include "../Master/Vmaster.h"
#include "../Slave1/Vslave1.h"
#include "../Slave2/Vslave2.h"
#include "../Slave3/Vslave3.h"

/*****************************************************************************
                    definition of the virtual channels
*****************************************************************************/
VA_CHANNEL(vc1)
{
 va_ch_mac_MPI_P2P     mpi_channel;
  VA_CCTOR(vc1)
   {
    VA_CEND
   };
};

VA_CHANNEL(vc2)
{
 va_ch_mac_MPI_P2P     mpi_channel;
  VA_CCTOR(vc2)
   {
    VA_CEND
   };
};

VA_CHANNEL(vc3)
{
 va_ch_mac_MPI_P2P     mpi_channel;
  VA_CCTOR(vc3)
   {
    VA_CEND
   };
};

/*****************************************************************************
                                    main program
*****************************************************************************/
int sc_main(int argc, char *argv[])
{
     /*---------------------------- VADeL initialization  ----------------------------*/
     va_init();

     /*----------------------- instanciate virtual channels  -------------------------*/
     vc1 VC1("VirtualChannelMPI1");
     vc2 VC2("VirtualChannelMPI2");
     vc3 VC3("VirtualChannelMPI3");

     /*-------------- instanciate virtuals modules & sc_modules. --------------*/
```

```
vmaster vmaster("VMaster");
vslave1 vslave1("VSlave1");
vslave2 vslave2("VSlave2");
vslave3 vslave3("VSlave3");
/*********************************************************************/
/*---------------------------------- Parameters -------------------------*/
/*********************************************************************/

/*----------------------Parameters for Virtual Nets--------------------------*/
VC1.level = "DL";
VC2.level = "DL";
VC3.level = "DL";

/*------------------- Parameters for Virtual Ports -----------------------------*/
vmaster.VPmpi1.level = "DL";
vmaster.VPmpi2.level = "DL";
vmaster.VPmpi3.level = "DL";
vslave1.VPmpi1.level = "DL";
vslave2.VPmpi1.level = "DL";
vslave3.VPmpi1.level = "DL";
vmaster.VPmpi1.C_DATA_TYPE = "VOID";
vmaster.VPmpi2.C_DATA_TYPE = "VOID";
vmaster.VPmpi3.C_DATA_TYPE = "VOID";
vslave1.VPmpi1.C_DATA_TYPE = "VOID";
vslave2.VPmpi1.C_DATA_TYPE = "VOID";
vslave3.VPmpi1.C_DATA_TYPE = "VOID";

/*------------------- Parameters for Virtuals Modules -------------------------*/

/* Virtual Master */
vmaster.CPUName          = "CpuARMIntegrator";
vmaster.level            = "DL";
vmaster.ItNumberAddress  = "0x00136000";
vmaster.ItTypeAddress    = "0x00136040";
vmaster.RamBase          = "0x00000000";
vmaster.RamLimit         = "0x08000000";
vmaster.FileBase         = "0x06000000";
vmaster.IrqStack         = "0x06000000";
vmaster.SvcStack         = "0x05FFFC00";
vmaster.HeapBase         = "0x00300000";
vmaster.UsrStack         = "0x04300000";

/* Virtual  Slave1 */
vslave1.CPUName          = "CpuARMIntegrator";
vslave1.level            = "DL";
vslave1.ItNumberAddress  = "0x00116000";
vslave1.ItTypeAddress    = "0x00136040";
vslave1.RamBase          = "0x00000000";
vslave1.RamLimit         = "0x08000000";
vslave1.FileBase         = "0x06000000";
vslave1.IrqStack         = "0x06000000";
vslave1.SvcStack         = "0x05FFFC00";
vslave1.HeapBase         = "0x00300000";
vslave1.UsrStack         = "0x04300000";

/* Virtual  Slave2 */
vslave2.CPUName          = "CpuARMIntegrator";
vslave2.level            = "DL";
vslave2.ItNumberAddress  = "0x00116000";
```

```
vslave2.ItTypeAddress    = "0x00136040";
vslave2.RamBase          = "0x00000000";
vslave2.RamLimit         = "0x08000000";
vslave2.FileBase         = "0x06000000";
vslave2.IrqStack         = "0x06000000";
vslave2.SvcStack         = "0x05FFFC00";
vslave2.HeapBase         = "0x00300000";
vslave2.UsrStack         = "0x04300000";

/* Virtual  Salve3 */
vslave3.CPUName          = "CpuARMIntegrator";
vslave3.level            = "DL";
vslave3.ItNumberAddress  = "0x00116000";
vslave3.ItTypeAddress    = "0x00136040";
vslave3.RamBase          = "0x00000000";
vslave3.RamLimit         = "0x08000000";
vslave3.FileBase         = "0x06000000";
vslave3.IrqStack         = "0x06000000";
vslave3.SvcStack         = "0x05FFFC00";
vslave3.HeapBase         = "0x00300000";
vslave3.UsrStack         = "0x04300000";

/*---------------------------- Parameters for  Modules ----------------------------*/
vmaster.Mmaster->setColifParam("module_level","DL");
vslave1.Mslave1->setColifParam("module_level","DL");
vslave2.Mslave2->setColifParam("module_level","DL");
vslave3.Mslave3->setColifParam("module_level","DL");

/*------------------------- Parameters for Modules Ports ----------------------*/

/* Internal Ports  ---------------------------------  */

/* @ Master Module */
/* Master Port pmpi1 for communication with Slave1*/
vmaster.Mmaster->pmpi1.C_DATA_TYPE                        = "VOID";
vmaster.Mmaster->pmpi1.SoftPortType                       = "PacketsOperations";
vmaster.Mmaster->pmpi1.R_FIFOTagAddress                   = " 0x90100000";//size 9x32b
vmaster.Mmaster->pmpi1.R_AddressHeadFIFOTagAddress        = " 0x90100024";
vmaster.Mmaster->pmpi1.R_AddressTailFIFOTagAddress        = " 0x90100028";
vmaster.Mmaster->pmpi1.R_NbElementFIFOTagAddress          = " 0x9010002C";
vmaster.Mmaster->pmpi1.R_NbElementMaxFIFOTag              = "9";
vmaster.Mmaster->pmpi1.R_FIFODataAddress                  = " 0x90100034";//size 9107x32b
vmaster.Mmaster->pmpi1.R_AddressHeadFIFODataAddress       = " 0x90111E34";
vmaster.Mmaster->pmpi1.R_AddressTailFIFODataAddress       = " 0x90111E38";
vmaster.Mmaster->pmpi1.R_NbElementFIFODataAddress         = " 0x90111E3C";
vmaster.Mmaster->pmpi1.R_NbElementMaxFIFOData             = "18304";
vmaster.Mmaster->pmpi1.R_FIFOTagAddressSem                = " 0x90111E40";
vmaster.Mmaster->pmpi1.R_FIFODataAddressSem               = " 0x90111E48";

vmaster.Mmaster->pmpi1.S_FIFOTagAddress                   = " 0x80100000";//size 9x32b
vmaster.Mmaster->pmpi1.S_AddressHeadFIFOTagAddress        = " 0x80100024";
vmaster.Mmaster->pmpi1.S_AddressTailFIFOTagAddress        = " 0x80100028";
vmaster.Mmaster->pmpi1.S_NbElementFIFOTagAddress          = " 0x8010002C";
vmaster.Mmaster->pmpi1.S_NbElementMaxFIFOTag              = "9";
vmaster.Mmaster->pmpi1.S_FIFODataAddress                  = " 0x80100034";//size 9107x32b
vmaster.Mmaster->pmpi1.S_AddressHeadFIFODataAddress       = " 0x80111E34";
vmaster.Mmaster->pmpi1.S_AddressTailFIFODataAddress       = " 0x80111E38";
vmaster.Mmaster->pmpi1.S_NbElementFIFODataAddress         = " 0x80111E3C";
vmaster.Mmaster->pmpi1.S_NbElementMaxFIFOData             = "18304";
```

```
vmaster.Mmaster->pmpi1.S_FIFOTagAddressSem                = " 0x80111E40";
vmaster.Mmaster->pmpi1.S_FIFODataAddressSem               = " 0x80111E48";
vmaster.Mmaster->pmpi1.ProcessorId                        = "1"; //Slave1 id
vmaster.Mmaster->pmpi1.RegExtItNumberAddress              = " 0x90116000";
vmaster.Mmaster->pmpi1.IT_NUMBER                          = "0" ;
vmaster.Mmaster->pmpi1.IT_LEVEL                           = "1" ;

/* Master Port pmpi2 */
vmaster.Mmaster->pmpi2.C_DATA_TYPE                        = "VOID";
vmaster.Mmaster->pmpi2.SoftPortType                       = "PacketsOperations";
vmaster.Mmaster->pmpi2.R_FIFOTagAddress                   = " 0xA0100000";//size 9x32b
vmaster.Mmaster->pmpi2.R_AddressHeadFIFOTagAddress        = " 0xA0100024";
vmaster.Mmaster->pmpi2.R_AddressTailFIFOTagAddress        = " 0xA0100028";
vmaster.Mmaster->pmpi2.R_NbElementFIFOTagAddress          = " 0xA010002C";
vmaster.Mmaster->pmpi2.R_NbElementMaxFIFOTag              = "9";
vmaster.Mmaster->pmpi2.R_FIFODataAddress                  = " 0xA0100034";//size 9107x32b
vmaster.Mmaster->pmpi2.R_AddressHeadFIFODataAddress       = " 0xA0111E34";
vmaster.Mmaster->pmpi2.R_AddressTailFIFODataAddress       = " 0xA0111E38";
vmaster.Mmaster->pmpi2.R_NbElementFIFODataAddress         = " 0xA0111E3C";
vmaster.Mmaster->pmpi2.R_NbElementMaxFIFOData             = "18304";
vmaster.Mmaster->pmpi2.R_FIFOTagAddressSem                = " 0xA0111E40";
vmaster.Mmaster->pmpi2.R_FIFODataAddressSem               = " 0xA0111E48";

vmaster.Mmaster->pmpi2.S_FIFOTagAddress                   = " 0x80112000";//size 9x32b
vmaster.Mmaster->pmpi2.S_AddressHeadFIFOTagAddress        = " 0x80112024";
vmaster.Mmaster->pmpi2.S_AddressTailFIFOTagAddress        = " 0x80112028";
vmaster.Mmaster->pmpi2.S_NbElementFIFOTagAddress          = " 0x8011202C";
vmaster.Mmaster->pmpi2.S_NbElementMaxFIFOTag              = "9";
vmaster.Mmaster->pmpi2.S_FIFODataAddress                  = " 0x80112034";
vmaster.Mmaster->pmpi2.S_AddressHeadFIFODataAddress       = " 0x80123E34";//size 9107x32b
vmaster.Mmaster->pmpi2.S_AddressTailFIFODataAddress       = " 0x80123E38";
vmaster.Mmaster->pmpi2.S_NbElementFIFODataAddress         = " 0x80123E3C";
vmaster.Mmaster->pmpi2.S_NbElementMaxFIFOData             = "18304";
vmaster.Mmaster->pmpi2.S_FIFOTagAddressSem                = " 0x80123E40";
vmaster.Mmaster->pmpi2.S_FIFODataAddressSem               = " 0x80123E48";

vmaster.Mmaster->pmpi2.ProcessorId                        = "2"; //Slave2 id
vmaster.Mmaster->pmpi2.RegExtItNumberAddress              = " 0xA0116000";
vmaster.Mmaster->pmpi2.IT_NUMBER                          = "1" ;
vmaster.Mmaster->pmpi2.IT_LEVEL                           = "1" ;

/* Master Port pmpi3 */
vmaster.Mmaster->pmpi3.C_DATA_TYPE                        = "VOID";
vmaster.Mmaster->pmpi3.SoftPortType                       = "PacketsOperations";
vmaster.Mmaster->pmpi3.R_FIFOTagAddress                   = " 0xB0100000";//size 9x32b
vmaster.Mmaster->pmpi3.R_AddressHeadFIFOTagAddress        = " 0xB0100024";
vmaster.Mmaster->pmpi3.R_AddressTailFIFOTagAddress        = " 0xB0100028";
vmaster.Mmaster->pmpi3.R_NbElementFIFOTagAddress          = " 0xB010002C";
vmaster.Mmaster->pmpi3.R_NbElementMaxFIFOTag              = "9";
vmaster.Mmaster->pmpi3.R_FIFODataAddress                  = " 0xB0100034";//size 9107x32b
vmaster.Mmaster->pmpi3.R_AddressHeadFIFODataAddress       = " 0xB0111E34";
vmaster.Mmaster->pmpi3.R_AddressTailFIFODataAddress       = " 0xB0111E38";
vmaster.Mmaster->pmpi3.R_NbElementFIFODataAddress         = " 0xB0111E3C";
vmaster.Mmaster->pmpi3.R_NbElementMaxFIFOData             = "18304";
vmaster.Mmaster->pmpi3.R_FIFOTagAddressSem                = " 0xB0111E40";
vmaster.Mmaster->pmpi3.R_FIFODataAddressSem               = " 0xB0111E48";

vmaster.Mmaster->pmpi3.S_FIFOTagAddress                   = " 0x80124000";//size 9x32b
vmaster.Mmaster->pmpi3.S_AddressHeadFIFOTagAddress        = " 0x80124024";
```

```
vmaster.Mmaster->pmpi3.S_AddressTailFIFOTagAddress        = " 0x80124028";
vmaster.Mmaster->pmpi3.S_NbElementFIFOTagAddress          = " 0x8012402C";
vmaster.Mmaster->pmpi3.S_NbElementMaxFIFOTag              = "9";
vmaster.Mmaster->pmpi3.S_FIFODataAddress                  = " 0x80124034";//size 9107x32b
vmaster.Mmaster->pmpi3.S_AddressHeadFIFODataAddress       = " 0x80135E34";
vmaster.Mmaster->pmpi3.S_AddressTailFIFODataAddress       = " 0x80135E38";
vmaster.Mmaster->pmpi3.S_NbElementFIFODataAddress         = " 0x80135E3C";
vmaster.Mmaster->pmpi3.S_NbElementMaxFIFOData             = "18304";
vmaster.Mmaster->pmpi3.S_FIFOTagAddressSem                = " 0x80135E40";
vmaster.Mmaster->pmpi3.S_FIFODataAddressSem               = " 0x80135E48";

vmaster.Mmaster->pmpi3.ProcessorId                        = "3";//Slave3 id
vmaster.Mmaster->pmpi3.RegExtItNumberAddress              = " 0xB0116000";
vmaster.Mmaster->pmpi3.IT_NUMBER                          = "2" ;
vmaster.Mmaster->pmpi3.IT_LEVEL                           = "1" ;
vmaster.Mmaster->MSAP2.SoftPortType                       = "GuardedRegister";

/* @ Slave1 Module */
vslave1.Mslave1->pmpi1.C_DATA_TYPE                        = "VOID";
vslave1.Mslave1->pmpi1.SoftPortType                       = "PacketsOperations";
vslave1.Mslave1->pmpi1.R_FIFOTagAddress                   = " 0x80100000";//size 9x32b
vslave1.Mslave1->pmpi1.R_AddressHeadFIFOTagAddress        = " 0x80100024";
vslave1.Mslave1->pmpi1.R_AddressTailFIFOTagAddress        = " 0x80100028";
vslave1.Mslave1->pmpi1.R_NbElementFIFOTagAddress          = " 0x8010002C";
vslave1.Mslave1->pmpi1.R_NbElementMaxFIFOTag              = "9";
vslave1.Mslave1->pmpi1.R_FIFODataAddress                  = " 0x80100034";//size 9107x32b
vslave1.Mslave1->pmpi1.R_AddressHeadFIFODataAddress       = " 0x80111E34";
vslave1.Mslave1->pmpi1.R_AddressTailFIFODataAddress       = " 0x80111E38";
vslave1.Mslave1->pmpi1.R_NbElementFIFODataAddress         = " 0x80111E3C";
vslave1.Mslave1->pmpi1.R_NbElementMaxFIFOData             = "18304";
vslave1.Mslave1->pmpi1.R_FIFOTagAddressSem                = " 0x80111E40";
vslave1.Mslave1->pmpi1.R_FIFODataAddressSem               = " 0x80111E48";

vslave1.Mslave1->pmpi1.S_FIFOTagAddress                   = " 0x90100000";//size 9x32b
vslave1.Mslave1->pmpi1.S_AddressHeadFIFOTagAddress        = " 0x90100024";
vslave1.Mslave1->pmpi1.S_AddressTailFIFOTagAddress        = " 0x90100028";
vslave1.Mslave1->pmpi1.S_NbElementFIFOTagAddress          = " 0x9010002C";
vslave1.Mslave1->pmpi1.S_NbElementMaxFIFOTag              = "9";
vslave1.Mslave1->pmpi1.S_FIFODataAddress                  = " 0x90100034";//size 9107x32b
vslave1.Mslave1->pmpi1.S_AddressHeadFIFODataAddress       = " 0x90111E34";
vslave1.Mslave1->pmpi1.S_AddressTailFIFODataAddress       = " 0x90111E38";
vslave1.Mslave1->pmpi1.S_NbElementFIFODataAddress         = " 0x90111E3C";
vslave1.Mslave1->pmpi1.S_NbElementMaxFIFOData             = "18304";
vslave1.Mslave1->pmpi1.S_FIFOTagAddressSem                = " 0x90111E40";
vslave1.Mslave1->pmpi1.S_FIFODataAddressSem               = " 0x90111E48";

vslave1.Mslave1->pmpi1.ProcessorId                        = "0"; //Master id
vslave1.Mslave1->pmpi1.RegExtItNumberAddress              = " 0x80136004";
vslave1.Mslave1->pmpi1.IT_NUMBER                          = "0" ;
vslave1.Mslave1->pmpi1.IT_LEVEL                           = "1" ;
vslave1.Mslave1->MSAP2.SoftPortType                       = "GuardedRegister";

/* @ Slave2 Module */
vslave2.Mslave2->pmpi1.C_DATA_TYPE                        = "VOID";
vslave2.Mslave2->pmpi1.SoftPortType                       = "PacketsOperations";
vslave2.Mslave2->pmpi1.R_FIFOTagAddress                   = " 0x80112000";//size 9x32b
vslave2.Mslave2->pmpi1.R_AddressHeadFIFOTagAddress        = " 0x80112024";
vslave2.Mslave2->pmpi1.R_AddressTailFIFOTagAddress        = " 0x80112028";
vslave2.Mslave2->pmpi1.R_NbElementFIFOTagAddress          = " 0x8011202C";
```

```
vslave2.Mslave2->pmpi1.R_NbElementMaxFIFOTag              = "9";
vslave2.Mslave2->pmpi1.R_FIFODataAddress                  = " 0x80112034";//size 9107x32b
vslave2.Mslave2->pmpi1.R_AddressHeadFIFODataAddress       = " 0x80123E34";
vslave2.Mslave2->pmpi1.R_AddressTailFIFODataAddress       = " 0x80123E38";
vslave2.Mslave2->pmpi1.R_NbElementFIFODataAddress         = " 0x80123E3C";
vslave2.Mslave2->pmpi1.R_NbElementMaxFIFOData             = "18304";
vslave2.Mslave2->pmpi1.R_FIFOTagAddressSem                = " 0x80123E40";
vslave2.Mslave2->pmpi1.R_FIFODataAddressSem               = " 0x80123E48";

vslave2.Mslave2->pmpi1.S_FIFOTagAddress                   = " 0xA0100000";//size 9x32b
vslave2.Mslave2->pmpi1.S_AddressHeadFIFOTagAddress        = " 0xA0100024";
vslave2.Mslave2->pmpi1.S_AddressTailFIFOTagAddress        = " 0xA0100028";
vslave2.Mslave2->pmpi1.S_NbElementFIFOTagAddress          = " 0xA010002C";
vslave2.Mslave2->pmpi1.S_NbElementMaxFIFOTag              = "9";
vslave2.Mslave2->pmpi1.S_FIFODataAddress                  = " 0xA0100034";//size 9107x32b
vslave2.Mslave2->pmpi1.S_AddressHeadFIFODataAddress       = " 0xA0111E34";
vslave2.Mslave2->pmpi1.S_AddressTailFIFODataAddress       = " 0xA0111E38";
vslave2.Mslave2->pmpi1.S_NbElementFIFODataAddress         = " 0xA0111E3C";
vslave2.Mslave2->pmpi1.S_NbElementMaxFIFOData             = "18304";
vslave2.Mslave2->pmpi1.S_FIFOTagAddressSem                = " 0xA0111E40";
vslave2.Mslave2->pmpi1.S_FIFODataAddressSem               = " 0xA0111E48";
vslave2.Mslave2->pmpi1.ProcessorId                        = "0";//Master id
vslave2.Mslave2->pmpi1.RegExtItNumberAddress              = " 0x80136008";
vslave2.Mslave2->pmpi1.IT_NUMBER                          = "1" ;
vslave2.Mslave2->pmpi1.IT_LEVEL                           = "1" ;
vslave2.Mslave2->MSAP2.SoftPortType                       = "GuardedRegister";

/* @ Slave3 Module */
vslave3.Mslave3->pmpi1.C_DATA_TYPE                        = "VOID";
vslave3.Mslave3->pmpi1.SoftPortType                       = "PacketsOperations";
vslave3.Mslave3->pmpi1.R_FIFOTagAddress                   = " 0x80124000";//size 9x32b
vslave3.Mslave3->pmpi1.R_AddressHeadFIFOTagAddress        = " 0x80124024";
vslave3.Mslave3->pmpi1.R_AddressTailFIFOTagAddress        = " 0x80124028";
vslave3.Mslave3->pmpi1.R_NbElementFIFOTagAddress          = " 0x8012402C";
vslave3.Mslave3->pmpi1.R_NbElementMaxFIFOTag              = "9";
vslave3.Mslave3->pmpi1.R_FIFODataAddress                  = " 0x80124034";//size 9107x32b
vslave3.Mslave3->pmpi1.R_AddressHeadFIFODataAddress       = " 0x80135E34";
vslave3.Mslave3->pmpi1.R_AddressTailFIFODataAddress       = " 0x80135E38";
vslave3.Mslave3->pmpi1.R_NbElementFIFODataAddress         = " 0x80135E3C";
vslave3.Mslave3->pmpi1.R_NbElementMaxFIFOData             = "18304";
vslave3.Mslave3->pmpi1.R_FIFOTagAddressSem                = " 0x80135E40";
vslave3.Mslave3->pmpi1.R_FIFODataAddressSem               = " 0x80135E48";

vslave3.Mslave3->pmpi1.S_FIFOTagAddress                   = "0xB0100000";//size 9x32b
vslave3.Mslave3->pmpi1.S_AddressHeadFIFOTagAddress        = " 0xB0100024";
vslave3.Mslave3->pmpi1.S_AddressTailFIFOTagAddress        = " 0xB0100028";
vslave3.Mslave3->pmpi1.S_NbElementFIFOTagAddress          = " 0xB010002C";
vslave3.Mslave3->pmpi1.S_NbElementMaxFIFOTag              = "9";
vslave3.Mslave3->pmpi1.S_FIFODataAddress                  = " 0xB0100034";//size 9107x32b
vslave3.Mslave3->pmpi1.S_AddressHeadFIFODataAddress       = " 0xB0111E34";
vslave3.Mslave3->pmpi1.S_AddressTailFIFODataAddress       = " 0xB0111E38";
vslave3.Mslave3->pmpi1.S_NbElementFIFODataAddress         = " 0xB0111E3C";
vslave3.Mslave3->pmpi1.S_NbElementMaxFIFOData             = "18304";
vslave3.Mslave3->pmpi1.S_FIFOTagAddressSem                = " 0xB0111E40";
vslave3.Mslave3->pmpi1.S_FIFODataAddressSem               = " 0xB0111E48";
vslave3.Mslave3->pmpi1.ProcessorId                        = "0"; //Master id
vslave3.Mslave3->pmpi1.RegExtItNumberAddress              = " 0x8013600C";
vslave3.Mslave3->pmpi1.IT_NUMBER                          = "2" ;
vslave3.Mslave3->pmpi1.IT_LEVEL                           = "1" ;
```

```
        vslave3.Mslave3->MSAP2.SoftPortType                         = "GuardedRegister";

        /* External Ports --------------------------------- */
        /* @ Master Module */
        vmaster.VPmpi1.POOL_SIZE = "32";
        vmaster.VPmpi1.HardPortType = "HNDSHK";
        vmaster.VPmpi2.POOL_SIZE = "32";
        vmaster.VPmpi2.HardPortType = "HNDSHK";
        vmaster.VPmpi3.POOL_SIZE = "32";
        vmaster.VPmpi3.HardPortType = "HNDSHK";

        /* @ Slave1 Module */
        vslave1.VPmpi1.POOL_SIZE = "32";
        vslave1.VPmpi1.HardPortType = "HNDSHK";

        /* @ Slave2 Module */
        vslave2.VPmpi1.POOL_SIZE = "32";
        vslave2.VPmpi1.HardPortType = "HNDSHK";

        /* @ Slave3 Module */
        vslave3.VPmpi1.POOL_SIZE = "32";
        vslave3.VPmpi1.HardPortType = "HNDSHK";

        /*--------------------------------- Parameters for Task Port --------------------*/
        /* @ Master Module */
        vmaster.Mmaster->Master->Pmpi1.SoftService= "HighIO/MPI/Pt2Pt/MPI_Std_Send;
HighIO/MPI/Pt2Pt/MPI_Std_Recv";
        vmaster.Mmaster->Master->Pmpi2.SoftService= "HighIO/MPI/Pt2Pt/MPI_Std_Send;
HighIO/MPI/Pt2Pt/MPI_Std_Recv";
        vmaster.Mmaster->Master->Pmpi3.SoftService= "HighIO/MPI/Pt2Pt/MPI_Std_Send;
HighIO/MPI/Pt2Pt/MPI_Std_Recv";

        vmaster.Mmaster->Master->Pmpi1.C_DATA_TYPE = "VOID";
        vmaster.Mmaster->Master->Pmpi2.C_DATA_TYPE = "VOID";
        vmaster.Mmaster->Master->Pmpi3.C_DATA_TYPE = "VOID";

        /* @ Slave1 Module */
        vslave1.Mslave1->Slave1->Pmpi1.SoftService= "HighIO/MPI/Pt2Pt/MPI_Std_Send;
HighIO/MPI/Pt2Pt/MPI_Std_Recv";
        vslave1.Mslave1->Slave1->Pmpi1.C_DATA_TYPE = "VOID";

        /* @ Slave2 Module */
        vslave2.Mslave2->Slave2->Pmpi1.SoftService= "HighIO/MPI/Pt2Pt/MPI_Std_Send;
HighIO/MPI/Pt2Pt/MPI_Std_Recv";
        vslave2.Mslave2->Slave2->Pmpi1.C_DATA_TYPE = "VOID";

        /* @ Slave3 Module */
        vslave3.Mslave3->Slave3->Pmpi1.SoftService= "HighIO/MPI/Pt2Pt/MPI_Std_Send;
HighIO/MPI/Pt2Pt/MPI_Std_Recv";
        vslave3.Mslave3->Slave3->Pmpi1.C_DATA_TYPE = "VOID";

        /*--------------------------------- Parameters for Tasks ----------------------*/
        /* @ Master Module */
        vmaster.Mmaster->Stby0->setColifParam("TaskPriority","0");
        vmaster.Mmaster->Stby0->setColifParam("Source", "SystemC", "Master/stby0.cpp",
"Master/stby0.h");
        vmaster.Mmaster->Stby0->setColifParam("module_level", "DL");
        vmaster.Mmaster->Stby0->setColifParam("module_type", "software");
```

```
        vmaster.Mmaster->Stby0->setColifParam("UserStackAddress"," 0x05D00000");
        vmaster.Mmaster->Stby0->setColifParam("SvcStackAddress", " 0x05E00000");

        vmaster.Mmaster->Master->setColifParam("TaskPriority","1");
        vmaster.Mmaster->Master->setColifParam("Source", "SystemC", "Master/task_master.cpp",
"Master/task_master.h");
        vmaster.Mmaster->Master->setColifParam("module_level", "DL");
        vmaster.Mmaster->Master->setColifParam("module_type", "software");

        vmaster.Mmaster->Master->setColifParam("UserStackAddress"," 0x05C00000");
        vmaster.Mmaster->Master->setColifParam("SvcStackAddress"," 0x05F00000");

        /* @ Slave1 Module */
        vslave1.Mslave1->Stby1->setColifParam("TaskPriority","0");
        vslave1.Mslave1->Stby1->setColifParam("Source", "SystemC", "Slave1/stby1.cpp",
"Slave1/stby1.h");
        vslave1.Mslave1->Stby1->setColifParam("module_level", "DL");
        vslave1.Mslave1->Stby1->setColifParam("module_type", "software");

        vslave1.Mslave1->Stby1->setColifParam("UserStackAddress"," 0x05D00000");
        vslave1.Mslave1->Stby1->setColifParam("SvcStackAddress", " 0x05E00000");

        vslave1.Mslave1->Slave1->setColifParam("TaskPriority","1");
        vslave1.Mslave1->Slave1->setColifParam("Source", "SystemC", "Slave1/task_slave1.cpp",
"Slave1/task_slave1.h");
        vslave1.Mslave1->Slave1->setColifParam("module_level", "DL");
        vslave1.Mslave1->Slave1->setColifParam("module_type", "software");

        vslave1.Mslave1->Slave1->setColifParam("UserStackAddress"," 0x05C00000");
        vslave1.Mslave1->Slave1->setColifParam("SvcStackAddress"," 0x05F00000");

        /* @ Slave2 Module */
        vslave2.Mslave2->Stby2->setColifParam("TaskPriority","0");
        vslave2.Mslave2->Stby2->setColifParam("Source", "SystemC", "Slave2/stby2.cpp",
"Slave2/stby2.h");
        vslave2.Mslave2->Stby2->setColifParam("module_level", "DL");
        vslave2.Mslave2->Stby2->setColifParam("module_type", "software");

        vslave2.Mslave2->Stby2->setColifParam("UserStackAddress"," 0x05D00000");
        vslave2.Mslave2->Stby2->setColifParam("SvcStackAddress", " 0x05E00000");

        vslave2.Mslave2->Slave2->setColifParam("TaskPriority","1");
        vslave2.Mslave2->Slave2->setColifParam("Source", "SystemC", "Slave2/task_slave2.cpp",
"Slave2/task_slave2.h");
        vslave2.Mslave2->Slave2->setColifParam("module_level", "DL");
        vslave2.Mslave2->Slave2->setColifParam("module_type", "software");

        vslave2.Mslave2->Slave2->setColifParam("UserStackAddress"," 0x05C00000");
        vslave2.Mslave2->Slave2->setColifParam("SvcStackAddress"," 0x05F00000");

        /* @ Slave3 Module */
        vslave3.Mslave3->Stby3->setColifParam("TaskPriority","0");
        vslave3.Mslave3->Stby3->setColifParam("Source", "SystemC", "Slave3/stby3.cpp",
"Slave3/stby3.h");
        vslave3.Mslave3->Stby3->setColifParam("module_level", "DL");
        vslave3.Mslave3->Stby3->setColifParam("module_type", "software");

        vslave3.Mslave3->Stby3->setColifParam("UserStackAddress"," 0x05D00000");
```

```
        vslave3.Mslave3->Stby3->setColifParam("SvcStackAddress", " 0x05E00000");

        vslave3.Mslave3->Slave3->setColifParam("TaskPriority","1");
        vslave3.Mslave3->Slave3->setColifParam("Source", "SystemC",
"Slave3/task_slave3.cpp","Slave3/task_slave3.h");
        vslave3.Mslave3->Slave3->setColifParam("module_level", "DL");
        vslave3.Mslave3->Slave3->setColifParam("module_type", "software");

        vslave3.Mslave3->Slave3->setColifParam("UserStackAddress"," 0x05C00000");
        vslave3.Mslave3->Slave3->setColifParam("SvcStackAddress"," 0x05F00000");

        /*************************************************************************/
        /*-------------------------------- Connecting Ports --------------------*/
        /*************************************************************************/
        /* Highest hierarchical level */
        (*vmaster.VP1)(VC1);
        (*vmaster.VP2)(VC2);
        (*vmaster.VP3)(VC3);

        (*vslave1.VP1)(VC1);
        (*vslave2.VP1)(VC2);
        (*vslave3.VP1)(VC3);

        /* Internal level */
        vmaster.VPmpi1(VC1.mpi_channel);
        vmaster.VPmpi2(VC2.mpi_channel);
        vmaster.VPmpi3(VC3.mpi_channel);

        vslave1.VPmpi1(VC1.mpi_channel);
        vslave2.VPmpi1(VC2.mpi_channel);
        vslave3.VPmpi1(VC3.mpi_channel);

        /*************************************************************************/
        /*-------------------------------- Translate on COLIF -------------------*/
        /*************************************************************************/
        sc_start(0);
        return EXIT_SUCCESS;
    };
```

VIII.4 Annexe C : détail des APIs CORBA utilisées pour la conception de l'application SDR

- **CORBA::ORB_ptr CORBA::ORB_init (int & argc, char ** argv)**

La fonction ORB_init est utilisée par le serveur et par le client pour se connecter à l'e*ORB et pour pouvoir effectuer les opérations de communication (les services requis et/ou fournis).

Selon l'implémentation d'e* ORB de « PrismTech », elle retourne au composant appelant un pointeur sur un objet ORB. L'objet ORB offre une interface regroupant certaines méthodes utiles pour lancer l'exécution et la terminer. Ces Méthodes sont « run » et « Shutdown ». La fonction « run » est utilisée par le serveur tandis que la fonction Shutdown est invoquée indirectement.

- **void CORBA::ORB::run (CORBA::Long timeout_msec)**

Comme c'est expliqué précédemment, cette fonction permet de lancer l'exécution de l'ORB. Lors de l'appel de cette fonction, il est impératif pour le serveur d'avoir enregistré les différents objets qui offrent les différents services. Le temps d'exécution de cette fonction peut être fixé en donnant en paramètre une variable temporelle « timeout_msec ». Le client ne peut s'exécuter correctement qu'après que le serveur ait exécuté cette primitive.

Cette fonction peut être arrêtée aussi si l'e*ORB exécute la fonction « Shutdown ». Elle sera expliquée ultérieurement.

- **CORBA::Object_ptr CORBA::ORB::resolve_initial_references**
  (const CORBA::ObjectId & id
  EORB_ENV_ARGN)

Cette fonction est exécutée par tous les composants. C'est un service offert par l'ORB. Elle fait partie des services de nommage. Elle permet de faire la résolution de référence. Elle n'est utilisée que si on utilise dans les paramètres d'exécution du serveur ou du client des adresses de localisation de ressources (URL). Dans e*ORB, elle permet de retourner un pointeur sur un objet à partir de son identifiant.

Une autre méthode est utilisée pour la recherche de cet objet si l'URL n'est pas utilisé. Cette méthode est :

- **CORBA::Object_ptr IORUtil::read**
  (const CORBA::ORB_ptr orb,
  char * name
  EORB_ENV_ARGN)

Elle retourne un pointeur sur un IOR (pour *interoperable object reference*). Ce pointeur représente une référence extraite d'un fichier qui contient des informations sur un objet déjà inscrit

dans l'e*ORB « orb ». La recherche de ce fichier s'effectue à partir du nom attribué dans la liste des paramètres de la fonction « name ».

Cette fonction est effectuée par le demandeur de service seulement.

- **void IORUtil::write**

```
( const CORBA::ORB_ptr orb,
  const CORBA::Object_ptr objref,
  char * name
  EORB_ENV_ARGN    )
```

Cette fonction est effectuée par les composants fournissant les services. Elle permet d'enregistrer une référence d'un objet « objref » dans l'e*ORB « orb » dans un fichier ayant le nom « name ».

- **PortableServer::ObjectId_ptr PortableServer::POA::activate_object**

```
( PortableServer::Servant p_servant
  EORB_ENV_ARGN    )
```

Cette primitive est effectuée au sein du serveur, elle permet d'associer un servant « p_servant » à l'objet offrant le service. Ce servant fera partie de la liste des servants du squelette et sera actif, c'est-à-dire, il est prêt à exécuter les requêtes demandées. Cette primitive retourne un identifiant de cet objet dans le POA.

- **PortableServer::POA_ptr PortableServer::POA::_narrow**

```
( CORBA::Object_ptr obj
  EORB_ENV_ARGN )
```

Cette primitive est utilisée au niveau du serveur. Elle permet de forcer le type : une surcharge de la classe de base (CORBA::Object) vers la classe dérivée (PortableServer::POA) lors de l'exécution. Cette fonction est nécessaire pour garantir la transparence d'exécution et de recherche des services.

Remarque :

Dans la modélisation donnée par « PrismTech » des macros sont définies. Elles sont utilisées dans les paramètres des primitives. Parmi ces macros on a : « EORB_ENV_ARGN » qui permet de définir l'environnement dans lequel on exécute le client et/ ou le serveur.

Certaines APIs sont utilisées au niveau des adaptateurs qui sont générés automatiquement. Ces APIs seront décrites dans la suite. Elles permettent de cacher les méthodes de communication et de recherche du service par rapport à l'utilisateur.

- **void CORBA::ORB::shutdown**

```
( CORBA::Boolean wait_for_completion
  EORB_ENV_ARGN    )
```

C'est la fonction qui va permettre d'arrêter le fonctionnement global de l'e*ORB. Elle s'effectuera si l'événement « wait_for_completion » est activé.

- **EORB::Codec::RequestId CORBA::Object::invoke_request**

```
( CORBA::String opname,
  CORBA::ULong oplen,
  CORBA::OperationMode mode,
  EORB::Codec::Param * putParams,
  CORBA::ULong putElems,
  EORB::Codec::Param * getParams,
  CORBA::ULong getElems
  EORB_ENV_ARGN )
```

Lors de l'appel d'un service fourni par l'interface décrite en IDL, le composant requérant ce service va faire un appel indirect à cette primitive à travers le conteneur. Elle permet d'invoquer un service de l'e*ORB. Elle comprend certains paramètres. Ces paramètres sont comme suit : « opname » identifie le nom du service requis, les pointeurs « putParams » et « getParams » identifient les variables d'entrées et de sortie pour ce service. A partir de ces paramètres, l'ORB génère une requête qui sera acheminée jusqu'au serveur.

- **void EORB::Codec::Request::get_args**

```
( EORB::Codec::Param * params,
  CORBA::ULong elems
  EORB_ENV_ARGN )
```

Lors de l'invocation d'un objet pour exécuter un service donnée, cet objet a besoin d'avoir les paramètres effectifs sur lesquels il doit travailler. Ces paramètres ont été envoyés par le composant requérant le service grâce à la primitive « invoke_request » décrite précédemment. La primitive « get_args » permet d'avoir les paramètres d'entrées du service à fournir à partir de l'e*ORB.

- **void EORB::Codec::Request::put_args**

```
( EORB::Codec::Param * params,
  CORBA::ULong elems
  EORB_ENV_ARGN )
```

Pour mettre à jour les paramètres de sorties d'un service réalisé, l'objet concerné doit envoyer à l'e*ORB les nouvelles valeurs de ces paramètres. Ceci est effectué par la primitive « put_args ».

- **CORBA::Boolean CORBA::Object::_is_a**

```
( const char * logical_is_type
  EORB_ENV_ARGN )
```

Cette primitive est utilisé par le squelette et la souche pour tester si le type sur lequel on travaille est le même que «`logical_is_type`». Ceci est nécessaire pour pouvoir effectuer certaines opérations sur cet objet.

- **`CORBA::Object_ptr CORBA::Object::_duplicate ( CORBA::Object_ptr obj )`**

Cette primitive est nécessaire pour dupliquer un objet en paramètres et retourner un pointeur sur ce nouvel élément dupliqué.

Références

A	
[a386]	a386, Bibilothèque C qui abstrait des processeurs. http://a386.nocrew.org/
[AMBA]	Bus AMBA, developé par ARM. http://www.arm.com/products/solutions/AMBA_Spec.html
[Actors]	G. Agha. « Actors : A model of concurent compution in distributed system ». *MIT Press,* 1986.
[Akgul 01]	B. S. Akgul, J. Lee and V. Mooney, « System-on-a-Chip processor synchronization hardware unit with task preemption support », dans les actes de *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '01), pp.149-157*, Novembre 2001.
[Alpha]	F. de Dinechin, P.Quiton, et T. Risset. « Structuration of the Alpha language ». Dans *Massively parallel programming models, Berlin. IEEE Computer Society Press*, Octobre 1995.
[Argus]	B. Liskov. « Implementation of Argus », Dans les actes de 11th symposium on operating systems principles, pp. 111-122. ACM Press, 1987.
[ARM Integrator AP]	Plateforme ARM Integrator, « Integrator/AP User Guide », ARM Limited, 1999. http://www.arm.com
B	
[BCM1480]	Boardcom, BCM1480 Quad-Core 64-bit MIPS http://www.broadcom.com/products/Enterprise-Small-Office/Communications-Processors/BCM1480
[Beidas 03]	R. Beidas, J. Zhu, « Performance Efficiency of Context-Flow System-on-Chip Platform», Dans les actes de *IEEE/ACM International Conference on Computer Aided Design,* Novembre 2003.
[Benini 03]	L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, M. Poncino. "SystemC cosimulation and emulation of multiprocessor SoC designs". Dans *IEEE Computer, 36/4, pp53-59*, Avril 2003.
[Beynon 02]	M.D. Beeynon, H. Andrade, J. Saltz, « Low-Cost Non-Intrusive Debugging Strategies for Distributed Parallel Programs », dans les actes de *IEEE International Conference on Cluster Computing (CLUSTER'02) pp. 439,* 2002.
[Bonaciu 06 – A]	M. Bonaciu, A. Bouchhima, W. Youssef, X. Chen, W. Cesario, A.A. Jerraya, « High-Level Architecture Exploration for MPEG4 Encoder with Custom Parameters ». Dans les actes de *11th Asia and South Pacific Design Automation Conference ASP-DAC Yokohama City, Japan.* Janvier 2006.
[Bonaciu 06 – B]	M. Bonaciu, « Conception d'Algorithme et d'Architecture Flexibles pour la Compression Vidèo » *Thèse de doctorat, INPG, Spécialité Microélectronique, laboratoire TIMA,* 2006.
[Bouchhima 04]	A. Bouchhima, S. Yoo, A.A. Jerraya, « Fast and Accurate Timed Execution of High Level Embedded Software Using HW/SW Interface Simulation Model ». Dans les actes *de ASP-DAC 2004, Yokohama, Japan,* Janvier 2004.
[Bouchhima 05]	A. Bouchhima, I. Bacivarov, W. Youssef, M. Bonaciu, A.A. Jerraya, « Using Abstract CPU Subsystem Simulation Model for High Level HW/SW Architecture Exploration », dans les actes de *ASP-DAC 2005, pp18-21, Shanghai, China,* Janvier 2005.
[Bouchhima 06]	A. Bouchhima « Modèlisation et Validation du Logiciel Embaqué à Différents Niveaux d'Abstraction en vue de la Synthèse et de la Validation des Systèmes sur Puce » , Thèse de doctorat, INPG, Spécialité Microélectronique, laboratoire TIMA, 2006.
[Brisolara 04]	L.B.Brisolara, L.B.Becker, L.Carro, F.R.Wagner, C.E.Pereira. « Evaluating High-Level Models for Real-Time Embedded Systems Design ». Dans les annales de *DIPES'2004 - IFIP Working Conference on Distributed and Parallel Embedded Systems. Toulouse, France,* Aout 2004.
[Brunel 00]	J.-Y. Brunel, et al., « COSY Communication IP's ». Dans les actes de Design Automation Conference, pp 406-409, Los Angeles, California, USA, 2000.

[BSP]	W. F. McColl Bulk « Synchronous Parallel Computing ». Dans les actes de *second workshop on abstract models for parallel computation*. Oxford University Press, 1994.
C	
[C EXECUTIVE]	Système d'exploitation C EXECUTIVE, produit par Software Systems. http://www.jmi.com/cexec.html
[CAN]	R. Bosch. « CAN Specification V2.0 », 1991. http://www.algonet.se/~staffann/developer/can2spec.pdf
[Casseau 02]	E. Casseau, « SoC Design Using Behavioral Level Virtual Components ». Dans les actes de *ICECS 02, IEEE International Conference on Electronics, Circuits, and Systems, Dubrovnik, Croatie,* Septembre 2002.
[Cell]	Dac C. Pham,.« The Design and Implementation of a First-Generation CELL Processor: A Multi-Core SuperComputer SoC » Dans les actes de *5th international forum on application specific multi-processor SoC, MPSoC'05, Relais de Margaux, France, Juillet 2005.*
[Cesario 01]	W. Cesario, G. Nicolescu, L. Gauthier, D. Lyonnard, « Colif: a Multi Design Representation for Application-Specific Multiprocessor System-on-Chip Design». Dans les actes de *12th Rapid System Prototyping (RSP'01), California,* 2001.
[Cesario 02]	W. Cesário, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, « Component-Based Design Approach for Multicore SoCs ». Dans les actes de *Design Automation Conference (DAC'02), New Orleans*, 2002.
[Chorus OS]	M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser, « Overview of the Chorus Operating System ». Dans les actes de *the USENIX Workshop on Micro-Kernels and Other Kernel Architectures, pp. 39–69,* Avril 1992.
[Clouard 02]	A. Clouard, « Functional and Timed Transactional-Level SoC Models in SystemC 2.0 », dans les actes de *5th European SystemC Users Group Meeting*, Mars 2002.
[COM/DCOM][DCOM]	« DCOM Technical Overview », par *Microsoft.* http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomtec.asp.
[Compositional C++]	K. M. Chandy, C. Kesselman, « Compositional C++: Compositional Parallel Programming ». Dans les actes de *5th International Workshop on Languages and Compilers for Parallel Computing*, pp 124 – 144, 1992.
[Concurent C]	H.H. Gehani and W.D. Roome « Concurrent C ». *Software-practice and experience*, ISSN:0038-0644, v.16 n.9: pp821-844. 1986.
[Concurent Prolog]	E. Shapiro. « Concurrent Prolog: A progress report ». Paru dans *IEEE Computer, 19: pp44-58*, Août 1986.
[ConvergenSC]	ConvergenSC / Incisive Design Flow. http://www.cadence.com/whitepapers/CDNCoWare_WPfnl.pdf
[CORBA]	Object Management Group, « The Common Object Request Broker: Architecture and Specification », Février 1998. http://www.omg.org
D	
[Dataparallel C*]	P.J. Hatcher and M.J. Quinn. « Data-Parallel programming on MIMD computers ». *MIT Press* 1991.
[Daveau 01]	J.M. Daveau, G.F. Marchioro, C. Valderrama, A.A. Jerraya, « VHDL Generation from SDL Specification ». Publié dans *readings in hardware/software co-design, Morgan Kaufmann Publishers, pp 125-134,* 2001.
[DCE]	« Distributed Computing Environment », de Open Software Foundation, « DCE Portal», The Open Group. http://www.osf.org/dce/
[De Lange 04]	M. De Lange, ACE Associated Compiler Experts, « Will the Software Dinosaurs Step Aside or Step on MPSoC? », au MPSoC'04, Saint-Maximin la Sainte Baume, France, Juillet 2004.
[Dick 00]	R. Dick, G. Lakshiminarayana, A. Raghunathan, and N. Jha, « Power Analysis of Embedded Operating Systems » pp.312-315. Dans les actes de *Design Automation Conference*, Juin 2000.
[Dietician]	Outil d'optimisation d'images du système d'exploitation QNX, http://www.qnx.com/developers/docs/momentics621_docs/ide_en/user_guide/builder.html#DIET
[DivX]	Format de compression vidéo. http://www.divx.com/

[Dorseuil 91]	A. Dorseuil and P. Pillot, « Le Temps Réel en Milieu Industriel », chapitres 2 et 3, pp 46 - 74. Bordas, 1991.
[DSP Consulting 99]	Liste de systèmes d'exploitation pour systèmes embarqués. http://www.dspconsulting.com/rtos.html
	E
[eCos]	Système d'exploitation eCos de Red Hat. http://sources.redhat.com/ecos/
[e*ORB]	Implémentation de la spécification [minimumCORBA], par Prismetech. http://www.prismtechnologies.com/section-item.asp?sid4=77&sid3=190& sid2=10&sid=18&id=404
[Enea Data]	http://www.enea.com/
[epsilon]	Système d'exploitation OSE Epsilon de Enea data. http://www.enea.com/templates/Page____1305.aspx
[EYRX]	Eyring Corporation. EYRX. http://www.jmi.com/eyrx.com
	F
[FORK]	C.W. Kessler and H. Seidl. « Integrating Synchronous and Asynchrounos Paradigms : The Fork95 Parallel Programming Language ». dans les actes de *massively parallel programming models, Berlin,1995. IEEE Computer Society Press,* Octobre1995.
	G
[GAUT]	Générateur d'Unité de Traitement Automatique, Produit par le Laboratoire d'Electronique des Systèmes TEmps Réel (LESTER). http://web.univ-ubs.fr/gaut/
[Gauthier 01]	L. Gauthier, « Génération de Système d'Exploitation pour le Ciblage de Logiciel Multitâche sur des Architectures Multiprocesseurs Hétérogènes dans le Cadre des Systèmes Embarqués spécifiques», *Thèse de Doctorat INPG, Spécialité Microélectronique, laboratoire TIMA,* 2001.
[Geib 97]	J.M. Geib, C. Gransart et P. Merle. « CORBA : des Concepts à la Pratique », *Collection InterEditions, Editions MASSON, Paris, France,* 1997.
[Gharsalli 03]	F. Gharsalli, « Conception des Interface Logiciel- Matériel pour l'Intégration des Mémoires Globales dans les Systems Monopoces*», Thèse de doctorat, INPG, Spécialité Informatique, laboratoire TIMA,* 2003.
[Grandpierre 00]	T. Grandpierre. « Modélisation d'architectures parallèles hétérogènes pour la génération automatique d'exécutifs distribués temps réel optimisés. Partie II : Génération automatique d'exécutif distribué ». *Thèse de doctorat, Université de Paris XI, UFR de Sciences*, 2000.
[Grasset 04]	A. Grasset,F. Rousseau,A.A. Jerraya. « Network Interface Generation for MPSOC: from Communication Service Requirement ti RTL Implementation», dans les actes *de Rapid system prototyping (RSP),* 2004.
[Grasset 06]	A. Grasset, « Synthèse des Interfaces de Communication dans la Conception des Systèmes Monopuces : de la Spécification à la Génération Automatique *», Thèse de doctorat, INPG, Spécialité Microélectronique, laboratoire TIMA,* 2006.
[Gropp 95]	W. Gropp and E. Lusk, « MPICH Working note : The Second Generation ADI for MPICH Implementation of MPI ». Dans *Mathematics and computer science division, Argonne national laboratory,* 1995*.*
[Gupta 95]	R. K. Gupta, « Co-synthesis of Hardware and Software for Digital Embedded Systems», *Kluwer Academic Publishers, Boston, MA,* 1995.
	H
[Han 04]	S.-I. HAN, A. BAGHDADI, M. BONACIU, S.-I. CHAE, A.A. Jerraya, « An Efficient Scalable and Flexible Data Transfer Architecture for Multiprocessor SoC with Massive Distributed Memory ». Dans les actes de *Design Automation Conference, DAC'04, San Diego, USA,* Juin 2004.
[Haskell]	P. Hudak and J. Fasel. « A Gentle Introduction to Haskell ». *ACM SIGPLAN Notices, 27, NO.5* Mai 1992.
[HP-MPI]	Librairie HP Message Passing Interface, http://h21007.www2.hp.com/dspp/tech/tech_TechDocumentDetailPage_IDX/1,1701,1238,00.html
	I
[ISETL-Linda]	A. Douglas, A. Rowstron, and A. Wood. « ISTEL-LindaParallel Programming with Bags ». Dans *Technical report YCS 257, Departement of computer science, University of York, U.K.*, Septembre 1995.

[ITRS]	International technology roadmap for semiconductors. http://public.itrs.net/
	J
[Java]	D. Lea. « Concurrent Programming in Java : Design Principles and Patterns ». *Addison-Wesley*, 1996.
[JaZZ]	Version « JaZZ » du système d'exploitation [ChorusOS]
[JPEG]	http://www.jpeg.org
[JTRS]	Joint Tactical Radio System, http://jtrs.army.mil/sections/technicalinformation/fset_technical_sca.html
[Jerraya 05],	A.A. Jerraya, « Long Term Trends for Embedded System Design », CEPA 2 Workshop – Digital Platforms for Defence, Bruxel , Belgique, Mars 15-16, 2005
	K
[Keeth 00]	B. Keeth and R. J. Baker. « Dram Circuit Design : A Tutorial ». *Wiley-IEEE Press*, novembre 2000.
[KeykOS]	A. Bomberger, N. Hardy, A. Frantz, C. Landau, W. Frantz, J. Shapiro, and A. Hardy. « The KeykOS Nanokernel Architecture ». Dans les actes de *USENIX Workshop on MicroKernels and Other Kernel* Architectures, pp 95–112, Avril 1992.
[Kriaa 02]	L. Kriaa, M.W. Youssef, G. Nicolescu, S. Martinez, S. Levitan, J. Martinez, T. Kurzweg, A.A. Jerraya, B. Courtois, "SystemC-Based Cosimulation for Global Validation of MOEMS", Dans les actes de *DTIP2002, Cannes-Mandelieu, France,* Mai 2002.
[Kriaa 05]	L. Kriaa, « Modélisation et Validation des Systèmes Hétérogènes : Définition d'un Modèle d'Exécution Global », *Thèse de doctorat, UJF, Spécialité Informatique, laboratoire TIMA,* 2005.
	L
[LAM/MPI]	Implémentation LAM/MPI de MPI. http://www.lam-mpi.org/about/overview/
[Lee 02]	J. Lee, K.K. Ryu and V.J. Mooney. « A Framework for Automatic Generation of Configuration Files for a Custom Hardware/Software RTOS». Dans les actes de *the international conference on engenerring of reconfigurable systems and algorithms (ERSA'02) pp. 31-37 Atlanta, Georgia, USA,* 2002.
[Lieverse 01]	P. Lieverse, *et al.*, « A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems», Dans *Journal of VLSI Signal Processing, vol. 29, pp. 197–206, Kluwer Academic Publishers,* 2001.
[Lindh 98]	L. Lindh, J. Furunäs, J. Adomat and M. El Shobaki; « Hardware accelerator for single and multiprocessor real-time operating systems». *Mälardalens högskola*, Sweeden, 1998.
[Lindsley 02]	B.L. Lindsley, U. Dayan, M. Tarrab, « Implementing a Real Time Task Scheduling Accelerator». *Brevet, Motorola, Inc. Schaumbrug, USA.* Mars 2002.
[Linux]	Systeme d'exploitation GNU, http://www.linux.org/info/index.html
[LogP]	D. Culler, R. Karp, D. Patterson, A. sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. « LogP : Toward a Realistic Model of Parallel Computation». Dans les actes de *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Mai 1993.
[Lorch 96]	J.R. Lorch, A.J. Smith. « Reducing Power Consumption by Improving Processor Time Management in a Single User Operating System ». Dans les actes de *2nd ACM international conference on mobile computing and networking*, 1996.
[LynxOS]	Système d'exploitation LynxOS, produit par LynuxWorks. http://www.lynuxworks.com.
[Lyonard 03]	Damien Lyonnard, « Approche d'Assemblage Systématique d'Elément d'Interface pour la Generation d'Architectures Multiprocesseurs », Thèse doctorat, INPG, Spécialité Microélectronique, laboratoire TIMA, 2003.
	M
[Mach]	D. Black, D. Golub, D. Julin, R. Rashid, R. Draves, R. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. « Microkernel Operating System Architectures and Mach ». Dans les actes de *the USENIX Workshop on Micro-Kernels and Other Kernel Architectures, pp 11–30,* Avril 1992.
[macro m4]	m4, implémentation GNU du processeur macro Unix, http://www.gnu.org/software/m4/

[Martin 93]	E. Martin, O. Stentieys, H. Dubois, J.L. Philippe, « GAUT : An Architectural Synthesis Tool for Dedicated Signal Processors », Dans les actes de *EURO-DAC 93, Hambourg, Germany,* septembre 1993.
[McMahon 96] [eMPICH]	T.P. McMahon**,** A. Skjellum**,** « eMPI/eMPICH: Embedding MPI ». Dans les actes de *Second MPI Developers Conference pp. 0180,* 1996.
[Micheli et al 96]	G. D. Micheli and M. Sami, editors, « Hardware/Software Co-Design », *Kluwer Academic Publishers, Norwell, MA*, 1996.
[micro]	Version « micro » du système d'exploitation [ChorusOS]
[MicroC/OS-II]	Micriμm. MicroC/OS-II. http://www.ucos-ii.com
[Microsoft C#]	T. Archer, « Formation à Microsoft C# » *Microsoft Press*, 2001.
[minimumCORBA]	minimumCORBA specification http://www.omg.org/docs/formal/02-08-01.pdf
[Modula3]	E.A. Heinz. « Modula-3* : An Efficiently Compilable Extension of Modula-3 for Problem-Oriented Explicitly Parallel programming ». Dans les actes de *Joint Symposium on Parallel Processing, pp 269-276, Waseda University, Tokyo*, Mai 1993.
[Moore 03]	L.J. Moore, A.R. Moya, « Non-Intrusive Debug Technique for Embedded Programming ». Dans les actes de *14th International Symposium on Software Reliability Engineering*, pp 375-380 ISSRE Novembre 2003.
[Moraes 05]	M.Moraes, E.Cota, F.R.Wagner, L.Carro, M.Lubaszewski. « An Application-Oriented Method for the Selection of Self-test Routines in Real-Time Embedded Systems ». Dans les actes de *LATW'05 - Latin-American Test Workshop. Salvador, Brazil,* Mars 2005.
[MPI]	J. Dongarra, S. W. Otto, M. Snir, and D. Walker. « An Introduction to the MPI Standard ». *Technical report CS-95-274, University of Tennessee*, Janvier 1995. http://www.netlib.org/tennessee/ut-cs-95-274.ps.
[MPICH]	Implémentation portable de MPI, http://www-unix.mcs.anl.gov/mpi/mpich/
[MPSoC'04]	4ieme séminaire international sur les systèmes sur puce multiprocesseurs spécifiques à l'application, Saint-Maximin la Sainte Baume, France, Juillet 2004.
N	
[Neutrino]	Micro noyau du système d'exploitation [QNX]. http://support7.qnx.com/download/download/8829/QNX_NeutrinoRTOS_Kernel_Benchmark_Methodology.pdf
[Nexperia]	Plateforme MPSoC, Philips. http://www.semiconductors.philips.com/products/nexperia/home/index.html
[Nexperia 8550]	Description du kit SDK du processeur Nexperia PNX8550. http://www.semiconductors.philips.com/acrobat_download/literature/9397/75013306.pdf
[Niagara]	P. Kongetira, K. Aingaran, and K. Olukotun. « Niagara: A 32-way Multithreaded Sparc Processor ». Dans *IEEE Micro, pp 21.29, Mars/Avril* 2005. http://www.sun.com/processors/UltraSPARC-T1/details.xml
[Nicolescu 02-A]	G. Nicolescu, S. Martinez, L. Kriaa, M.W. Youssef, S. Yoo, B. Charlot, A.A. Jerraya, "Application of Multi-domain and Multi-language Cosimulation to an Optical MEM Switch Design ", Dans les actes de *ASP-DAC 2002, Bangalore, India,* Janvier 2002.
[Nicolescu 02-B]	E.G.N. Nicolescu, « Spécification et Validation des Systèmes Hétérogènes Embarqués » , Thèse de doctorat, INPG, Spécialité Microélectronique, laboratoire TIMA, 2002.
[Nomadik]	Plateforme MPSoC, STMicroelectronics. http://www.st.com/stonline/products/promlit/pdf/flnomadik0205.pdf
[NucleusPLUS]	Système d'exploitation NucleusPLUS, produit par Accelerated technology http://www.acceleratedtechnology.com
O	
[Occam]	G. Jones and M. Goldsmith. « Programming in Occam2 ». *Prentice-Hall*, 1988.
[OMAP]	Plateforme MPSoC OMAP http://focus.ti.com/general/docs/wtbu/wtbugencontent.tsp?templateId=6123&navigationId=11988&path=templatedata/cm/ general/data/wtbovrvw/omap
[OMG]	Object Management Group, http://www.omg.org/

[OMG CORBA SPEC]	Les différentes spécifications du standard CORBA http://www.omg.org/technology/documents/corba_spec_catalog.htm
[OnCore OS]	Système d'exploitation OnCore OS, Produit par Oncore Systems. http://www.oncoresystems.com/
[OPERA]	J. Briat, M. Favre, C. Geyer, and J. Chassin de Kergommeaux. « Scheduling of OR-parallel Prolog on a Scalable Reconfigurable Distiributed Memory Multiprocessor. Dans les actes de *PARLE 91, Springer Lecture Notes in Computer Science 506, pp 385-402. Springer-Verlag,* 1991.
[OpenMP]	Modèle de programmation parallèle OpenMP http://www.openmp.org/drupal/
[Opus]	P. Mehrotra and M. Haines. « An Overview of the Opus Language and Runtime System » . Dans *Technical Report » pp 94-39, NASA ICASE,* Mai 1994.
[OS/360]	G. Mealy, B. Witt, and W. Clark. « The functional structure of OS/360 ». *IBM SystemsJournal, 5(1),* January 1966.
[OSE gateway]	OSE Gateway, produit par Enea Data. http://www.enea.com/epibrowser/Literature%20(pdf)/Pdf/Datasheets/OSE%20Gateway.pdf
[OSE]	Système d'exploitation , par Enea Data http://www.enea.com/templates/Extension____1301.aspx
[OSEck]	Verison compacte du noyau de [OSE], par Enea Data http://www.enea.com/templates/Page____1304.aspx
P	
[P3L]	M. Danelutto, R. di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. « A Methodology for the Development and the Support of Massively Parallel Programs ». Dans *Future generation computer system.*1992. Apparu aussi comme « The P3L language: An introduction ». Dans *Hewlett-Packard Report, HPL-PSC-91-29,* Decembre 1991.
[PALM]	M. Cannataro, G. Spezzano, and D. Talia « A Parallel Logic System on a Multicomputer Architecture ». Dans Future genration Computer System 6, pp 317–331. 1991.
[Patrallel sets]	M.F. Kilian. « Parallel Sets : An Object-Oriented methodology for massively parallel programming. » Thése de doctorat, Harvard university, 1992
[Paul 03]	JoAnn M. Paul « Programmers' views of SoCs ». Dans les annles de *CODESS+ISSS 03, California,* Octobre 2003.
[Paulin 02]	P. G. Paulin, C. Pilkington, E. Bensoudane, « StepNP: A System-Level Exploration Platform for Network Processors ». Dans *JEEE Design & Test of Computers, vol. 19, no.6,* Novembre 2002.
[Paulin 04 - A]	P.G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, G. Nicolescu. « Parallel programming models for multiprocessor SoC Platform applied to high speed traffic management ». Dans les actes de *CODES+ISSS'04, pp 48-53*, 2004.
[Paulin 04 - B]	P.G. Paulin et al, « Application of a Multi-Processor SoC Platform to High-speed Packet Forwarding ». Dans les actes de *DATE (Designer Forum), Paris*, Fevrier 2004.
[Paulin 05]	P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, K. Szabo, D. Lyonnad, and G. Nicolescu. « Multi-Processor SoC Platform and Tools for Communications Applications ». Chapitre dans *Embedded Systems Handbook, CRC Press*, 2005.
[Paviot 04]	Y. Paviot, « Implémentations Mixtes Logicielles/Matérielles des Services de Communication pour l'exploration du Partitionnement Logiciel/Matériel », Thèse de doctorat, INPG, Spécialité Microélectronique, laboratoire TIMA, 2004
[Pernet 02]	N. Pernet, Y. Sorel, « A Design Method for Implementing Specifications Including Control in Distributed Embedded Systems ». Dans les actes de *ETFA 2005 IEEE International Conference on Emerging Technologies and Factory Automation,* September 2005.
[Platform Express]	Platform Express, http://www.mentor.com
[Posadas 04]	H. Posadas, F. Herrera, P. Sánchez, E. Villar F. Blasco. "System-level performance analysis in SystemC". Dans les actes de *the Design, Automation and Test Conference, Paris, France*, Fevrier 200.
[POOL-T]	P. America. « POOL-T : A Parallel Object-Oriented Language ». Dans A. Yonezawa et al., editor, object-oriented concurent programming, pp 199-220, MIT Press, 1987.
[PPP]	B.S. Fagin and A.M. Despain. « The Performance of Parallel Prolog Programs ». Dans *IEEE transactions on computers, C-39, No.12 :pp1434-1445,* 1990.
[primitives MPI]	Spécification MPI http://www.mpi-forum.org/index.html

[PrimsTech]	http://www.prismtechnologies.com/
[pRISM+]	Environement de dévéloppement pour [pSOS]
[Probert 91]	D. Probert, et al. « SPACE: A New Approach to Operating System Abstraction ». Dans les actes de *International Workshop on Object Orientation in Operating Systems,* pp. 133–137, Octobre 1991.
[Projet Mayo]	Project Mayo, http://www.projectmayo.com, 2001
[pSOS]	Système d'exploition pSOS http://www.realtime-info.be/Magazine/99q3/pSOS.pdf
[pSOS+m]	Version multiprocesseur du noyau de [pSOS]
[PVM]	A. Beguelin, J. Dongarra, A. Geist, R. Manchek, S. Otto, and J. Walpole. « PVM : Experiences, Current Status and Future Direction ». Dans *Technical report CS/E 94-015 oregon graduate institute CS,* 1994.
Q	
[Qnet]	[QNX] networking, http://www.symmetry.com.au/Brochures%20and%20whitepapers/QSS/QNX_Networking_Qnet.pdf
[QNX]	D. Hildebrand. « An Architectural Overview of QNX ». Dans les actes de *USENIXWorkshop on Micro-Kernels and Other Kernel Architectures, pp 113–126,* Avril 1992.
R	
[RT Linux]	Version temps réel du système d'exploitation Linux
[RTXC]	Système d'exploitation RTXC par Embedded Power Corporation. http://www.embeddedpower.com
S	
[Sarmento 05]	A. Sarmento, L. Kriaa, A. Grasset, M.W. Youssef, A. Bouchhima, F. Rousseau, W. Cesario, A.A. Jerraya, « Service dependency graph, an efficient model for hardware/software interface modeling and generation for SoC design ». Dans les actes de *CODES+ISSS,* Septembre 2005.
[Schirrmeister 02]	Partie « Abstraction – A brief history ». Dans Seminar on virtual component codesign , proposé par Cadence Inc. 2002.
[Sckillicorn 98]	D.B. Skillicorn, D. Talia. « Models and Languages for Parallel Computation ». Publié dans *ACM Computing Surveys, Vol. 30, No. 2,* Juin 1998.
[Shah 03]	N. Shah, W. Plishker, K. Keutzer, « NP-Click : A Programming Model for the Intel IXP1200 ». Dans les actes de *9th international symposium on High Performance Computer Architecture,* 2003.
[Shalan et al 00]	M. Shalan and V. Mooney, « A Dynamic Memory Management Unit for Embedded Real-Time System-on-a-Chip». Dans les actes de *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '00), pp. 180-186.* Novembre 2000.
[Shiu et al 01]	P. H. Shiu, Y. Tan and V. Mooney, « A Novel Parallel Deadlock Detection Algorithm and Architecture». Dans les actes de *9th International Workshop on Hardware/Software Co-Design (CODES '01), pp.30-36,* Avril 2001.
[Sisal]	J.R. Mcgraw. « Parallel functional programming in Sisal: Fictions, facts, and future ». Dans Advanced Workshop, Programming Tools for Parallel Machines. Aussi disponible comme Laurence Livermore National Laboratories Technical Report. UCRL-JC-114360. Juin 1993.
[Stallman 91]	R. Stallman. « The C Preprocessor ». 1991.
[Sun]	http://fr.sun.com/
[System builder tool]	Outil de la suite QNX Monumentics http://www.qnx.com/products/development/
[System Studio]	CoCentric System Studio, http://www.synopsys.com/
[Systemc 00]	G. Arnout, « SystemC standard ». Dans *IEEE pp 573-577.* 2000. http://www.Systemc.org
T	
[Tanenbaum 94]	A. Tanenbaum. « Systèmes d'exploitation ». *Prentice Hall*, 1994.
[TCP/IP]	W. R. Stevens. « TCP/IP Illustrated, Volume 1: The Protocols ». *Addison-Wesley,* 1994.
[TI C5510]	Processeur de traitement de signal (DSP TMS320C5510) de Texas Instruments (TI), http://focus.ti.com/lit/ds/symlink/tms320vc510.pdf

[TLM Transfert] [TLM Transaction]	F.Ghenassia and A.Clouard « Chapter 1: TLM: An Overview and Brief History ». Dans Transaction-Level Modeling with SystemC, TLM Concepts and Applications for Embedded Systems, edited by Ghenassia, Frank 2005, XV . ISBN: 0-387-26232-6. 2005.
[Tornado II]	Windriver Tornado. http://www.windriver.com.
[Tron 00]	TRON Association Survey, late1999-early2000, Japan. 2000.
[TTL]	P. Van der Wolf, E.D. Kock, T. Henriksson, W. Kruijtzer, G. Essink. « Design and programming of Embedded Multiprocessors: an Interface-Centric Approach ». Dans les actes de *CODES+ISSS, pp 206-217*, 2004.
U	
[Unix]	S. Bourne. « Le Système UNIX ». 1985.
V	
[Van der Wolf 04]	P. Van der Wolf, E.D. Kock, T. Henriksson, W. Kruijtzer, G. Essink. « Design and programming of Embedded Multiprocessors: an Interface-Centric Approach ». Dans les actes de *CODES+ISSS, pp 206-217*, 2004.
[Van Der Wolf 05]	P. Van der Wolf. « Parallel Programming Models for Heterogenous MPSoCs ». Dans 5th international forum on application specific multi-processor SoC, MPSoC'05, Relais de Margaux, France, Juillet 2005. http://tima.imag.fr/mpsoc
[Verfaillie 03]	J. Verfaillie, « Realising High Level of Observability for SoC Debug by Combining Scan and Real Time Tracing », Dans *MEDEA+ Design Automation Conference*, Novembre 2003.
[Virtex-II Pro] [Virtex-II Pro Dev system]	Virtex-II Pro FPGAs, http://www.xilinx.com
[Virtuoso]	Système d'exploitation Virtuoso, produit par Eonic. http://www.eonic.com
[VisualityNQ]	VisualityNQ, produit par [Visuality Systems] http://www.visualitynq.com
[Visuality Systems]	Visuality Systems Ltd http://www.cifs.com/
[VMS]	H.M. Levy and R.H. Eckhouse. « Computer Programming and Architecture ». *Digital Press*, 1989.
[VSPWorks]	VSPWorks, produit par Windriver. http://www.windriver.com/products/device_technologies/os/vspworks/vspworks_tech_brief.pdf
[VxMP]	Extension VxMP du système d'exploitation VxWorks http://www.windriver.com/products/device_technologies/middleware/vxmp/vxmp.pdf
[VxVMI]	Extension VxVMI du système d'exploitation VxWorks http://www.windriver.com/products/device_technologies/middleware/vxvmi/vxvmi.pdf
[VxWorks] [VxWorks AE] [VxWorks 5.0] [VxWorks 6.0]	Différentes versions du système d'exploitation VxWorks. Par Wind River Systems. http://www.windriver.com
W	
[Wang 03]	S. Wang, S. Malik, and R. A. Bergamaschi, « Modeling and Integration of Peripheral Devices in Embedded Systems ». Dans les actes de *Design Automation and Test in Europe (DATE'03)*, Mars 2003.
[Wind]	Noyau du système d'exploitation [VxWorks]
[WindRiver System] [Windriver]	WindRiver Systems, Inc. http://www.windriver.com
[Windows Ce]	Système d'exploitation, produit par Microsoft http://download.microsoft.com/download/7/2/f/72fef3b0-9545-46a4-8886-a94f265df9c4/EVA-2.9-OS-CE-01-I01.pdf
Y	
[Yoo 03-A]	S. Yoo, G. Nicolescu, I. Bacivarov, M.W. Youssef, A. Bouchhima, A.A. Jerraya, "Multi-Level Software Validation for NoC", *Chapitre10 dans Networks on Chip", Kluwer Academic Publishers,* 2003.

[Yoo 03-B]	S. Yoo, I. Bacivarov, A. Bouchhima, Y. Paviot, A.A. Jerraya. « Building fast and accurate SW simulation models based on hardware abstraction layer and simulation environment abstraction layer ». Dans les actes de *Design, Automation and Test Conference, Munich, Germany*, Mars 2003
[Yoo 04]	S. Yoo, M.W. Youssef, A. Bouchhima, A.A. Jerraya, M. Diaz-Nava, « Multi-Processor SoC Design Methodology Using a Concept of Two-Layer Hardware-Dependent Software », Dans les actes de *Design Automation and Test in Europe, DATE'04, Paris, France,* Février 2004.
[Youssef 04]	M.W. Youssef, S. Yoo, A. Sasongko, Y. Paviot, A.A. Jerraya, "Debugging HW/SW Interface for MPSoC: Video Encoder System Design Case Study", Design Automation Conference, DAC'04, San Diego, USA, June 2004.
Z	
[Zuberi 99]	K. M. Zuberi and K. G. Shin. « Emeralds: A Small-Memory Real-Time Microkernel. » Dans *SOSP99*, 1999.
[Zwiers 93]	J. Zwiers and W. Janssen. « Partial Order Based Design of Concurrency Systems ». Dans Springer Verlag, editor, REX School/Symposium, pp 622–684. In a Decade of Concurrency, reflexions and perspectives, Juin 1993.

Publications

M. BONACIU, A. BOUCHHIMA, **M.W. YOUSSEF**, X. CHEN, W. CESARIO, A.A. JERRAYA, "High-Level Architecture Exploration for MPEG4 Encoder with Custom Parameters" 11th Asia and South Pacific Design Automation Conference ASP-DAC 2006, Jan. 24-27 2006, Yokohama City, Japan.

A. SARMENTTO, L. KRIAA, A. GRASSET, **M.W. YOUSSEF**, A. BOUCHHIMA, F. ROUSSEAU, W. CESARIO, A.A. JERRAYA, « Service dependency graph, an efficient model for hardware/software interface modeling and generation for SoC design » CODES+ISSS, September 2005.

A. BOUCHHIMA, I. BACIVAROV, **M. W. YOUSSEF**, M. BONACIU, A.A. JERRAYA, "Using Abstract CPU Subsystem Simulation Model for High Level HW/SW Architecture Exploration", ASP-DAC 2005 proceedings, Shanghai, China, January 2005.

M.W. YOUSSEF, S. YOO, A. SASONGKO, Y. PAVIOT, A.A. JERRAYA, "Debugging HW/SW Interface for MPSoC: Video Encoder System Design Case Study", Design Automation Conference, DAC'04, San Diego, USA, June 2004.

S. YOO, **M.W. YOUSSEF**, A. BOUCHHIMA, A.A. JERRAYA, M. DIAZ-NAVA, "Multi-Processor SoC Design Methodology Using a Concept of Two-Layer Hardware-Dependent Software". Design Automation and Test in Europe, DATE'04, Paris, France, February 2004.

S. YOO, G. NICOLESCU, I. BACIVAROV, **M.W. YOUSSEF**, A. BOUCHHIMA, A.A. JERRAYA, "Multi-Level Software Validation for NoC", Chapter 10 in "Networks on Chip", Kluwer Academic Publishers, 2003.

L. KRIAA, **M.W. YOUSSEF**, G. NICOLESCU, S. MARTINEZ, S. LEVITAN, J. MARTINEZ, T. KURZWEG, A.A. JERRAYA, B. COURTOIS, "SystemC-Based Cosimulation for Global Validation of MOEMS", DTIP2002, Cannes-Mandelieu, France, May 2002.

G. NICOLESCU, S. MARTINEZ, L. KRIAA, **M.W. YOUSSEF**, S. YOO, B. CHARLOT, A.A. JERRAYA, "Application of Multi-domain and Multi-language Cosimulation to an Optical MEM Switch Design ", ASP-DAC 2002, Bangalore, India, January 2002.

Résumé : Les systèmes mono-puce sont composés d'une partie logicielle et d'une partie matérielle. L'exécution de la partie logicielle sur les ressources de la partie matérielle est assuré a travers l'utilisation d'une interface logicielle/matérielle. Cette interface a une structure complexe, sa conception nécessite des compétences issues des domaines du logiciel et du matériel. Pour maîtriser cette complexité, des approches de conception de haut niveau sont requises. Dans cette optique, un flot de conception des systèmes MPSoC est proposé. Il est basé sur l'utilisation des API des modèles de programmation parallèle en vue de l'abstraction des interfaces logicielles/matérielles lors de la conception de la partie logicielle, puis de leur génération automatique en raffinant l'API utilisée sur l'architecture cible. Pour arriver à ce but, (1) une étude de l'architecture des interfaces logicielles/matérielles a été réalisé. Puis, (2) une étude des modèles de programmation parallèle et une classification en fonction de leur niveau d'abstraction a été effectué. Ensuite, le flot proposé a été utilisé pour la conception de deux applications : (1) un encodeur vidéo OpenDivX en utilisant le modèle de programmation parallèle MPI et la plateforme ARM IntegratorAP comme architecture matérielle cible, (2) une radio définie par logiciel en utilisant le modèle de programmation CORBA et une architecture matérielle spécifique comme architecture cible.

Mots clés : Systèmes MPSoC, interfaces logicielles/matérielles, modèle de programmation parallèle, logiciel dépendant du matériel.

Title: Hardware/Software interfaces study in the scope of multiprocessor system-on-chip and parallel programming models

Abstract: Today's systems-on-chip are multiprocessor. They are characterized by an increasing complexity and a reduced time to market. To tackle this complexity, the use of high level programming models seems to be a promising approach. In this work, we propose an MPSoC design flow, based on the use of high level parallel programming models API to design embedded software. An automated refinement of these API on target architecture is used. For that purpose, (1) MPSoC hardware/software interfaces were studied; then (2) parallel programming models and their classification in terms of provided abstraction were presented. The proposed flow has been used in two design experiments: (1) an MPEG video encoder, namely OpenDivX, using the MPI parallel programming model and targeting the ARM Integrator prototyping platform, (2) a software defined radio using the CORBA parallel programming model and targeting specific hardware architecture.

Key words: System-on-chip, MPSoC, hardware software interface, parallel programming model, hardware dependent software.

INTITULE ET ADRESSE DU LABORATOIRE

Laboratoire TIMA, 46 avenue Félix Viallet, 38031 Grenoble Cedex, France.

ISBN : (version brochée) ISBN: 2-84813-085-7

ISBNE : (version électronique) ISBN: 2-84813-085-7