Evaluation des performances pour les systèmes embarqués hétérogènes, multiprocesseur monopuces

I. Bacivarov

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

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

T H E S E

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Micro et Nano Electronique
préparée au laboratoire **TIMA**
dans le cadre de l'Ecole Doctorale « ELECTRONIQUE, ELECTROTECHNIQUE, AUTOMATIQUE, TELECOMMUNICATIONS, SIGNAL » (EEATS)

présentée et soutenue publiquement

par

Iuliana Beatrice BACIVAROV

Titre :

Evaluation des performances pour les systèmes embarqués hétérogènes, multiprocesseur monopuces

Directeur de thèse :

Ahmed Amine JERRAYA

JURY

M. Guy Mazaré,	**Président**
M. Paolo Ienne,	**Rapporteur**
M. Stanislaw Piestrak,	**Rapporteur**
M. Ahmed Amine Jerraya,	**Directeur de thèse**
M. Marcello Coppola,	**Examinateur**
M. Paul Amblard,	**Examinateur**

A ma famille,
A Cosmin

Remerciements

Je remercie à mon directeur de thèse, **M. Ahmed Jerraya** pour son accueil dans le groupe SLS, son soutien et la liberté qu'il m'a laissé dans mon travail.

Je remercie au directeur du laboratoire, **M. Bernard Courtois**, pour son accueil au laboratoire TIMA.

Je remercie à **M. Guy Mazaré** d'avoir présidé mon jury de thèse, et aussi pour ses appréciations.

Je tiens à remercier à **M. Paolo Ienne** et **M. Stanislaw Piestrak** d'avoir accepté rapporter sur mon travail de thèse, pour leurs commentaires pertinents et pour les horizons qu'ils m'ont inspiré à partir de mes travaux de thèse.

Je remercie à **M. Marcello Coppola** pour sa participation à mon jury de thèse, mais aussi pour les discussions qu'on a eu durant cette thèse.

Je tiens à remercier en particulier à **M. Paul Amblard** pour son soutien et de m'avoir aidé à améliorer ce manuscrit de thèse.

Je remercie à tous ceux qui m'ont fait découvrir l'enseignement, et qui m'ont permis de m'intégrer activement dans une équipe d'enseignants. D'abord je tiens remercier aux deux générations de présidents du CIES, **M. Gerard Cognet** et **M. Didier Retour**. Je remercie **M. Marc Brunello**, mon tuteur d'enseignement. Je remercie en particulier à **M. Emmanuel Toutain**, avec qui j'ai partagé deux années successifs d'enseignement d'électronique, qui m'a beaucoup appris sur les aspects pratiques de l'électronique, mais aussi sur les aspects pédagogiques. Je remercie à **M. Alain Bolopion** et **M. Stéphane Ploix** avec qui j'ai collaboré pour l'enseignement des bases de données. Je remercie à **Mlle Lorena Anghel** de m'avoir donné l'opportunité d'enseigner un domaine qui est très proche de mes travaux de thèse : la conception des systèmes MPSoC. Et en particulier je tiens remercier à **M. Frederic Rousseau** et **M. Paul Amblard** de m'avoir offert l'opportunité d'enseigner sur la conception avancée des circuits intégrés, aux étudiants du Master Recherche.

Je veux remercier en spécial à **M. Gilbert Vincent** et à **M. Pierre Gentil** pour leurs bons conseils dans mon orientation professionnelle, pour leur sincérité et disponibilité.

Un grand merci à tous mes professeurs de l'Université Polytechnique de Bucarest qui m'ont enseigné les bases de l'électronique et de l'informatique, connaissances que j'ai fortement utilisées pendant ma thèse et pendant mes enseignements. Sans pouvoir être exhaustive et citer tous les noms à qui je pense, je remercie à mes professeurs : **M. Constantin Bucur, M. Corneliu Burileanu, M. Dan Claudius, M. Radu Dogaru, M. Neculai Dumitriu, M. Nicolae Enescu, Mme Felicia Ionescu, M. Vasile Lazarescu, Mme. Anca Manolescu, Mme. Adelaida Mateescu, M. Mihnea Moroianu, M. Dan Mihut, M. Radu Radescu, M. Constantin Radoi, M. Adrian Rusu, M. Stefan Stancescu, M. Lucian Stanciu, M. Ioan Tache.**

Je remercie à toutes les trois générations de l'équipe SLS pour leur collaboration et pour leur sympathie. Je tiens à remercier à tous qui m'ont aidé à commencer cette thèse **Gabriela**, **Damien**, **Lovic**, **Sungjoo**,

Wander, **Yannick** et **Ferid**. Je remercie à **Lorenzo** et **Florin** pour leur collaboration dans le cadre de ce travail et leur amitié. Je tiens à remercier en particulier à **Lobna**, **Wassim** et **Aimen**, mes collègues de bureau pour leur sympathie et leur amitié. J'exprime mon gratitude à tous mes amis du groupe SLS (en manque d'espace je vais les citer en ordre alphabétique, mais je porte pour chacun d'entre eux un sentiment d'amitié personnel) : **Adina**, **Alex**, **Claudia**, **Diana**, **Florin D.**, **Frederic H**., **Gabriela**, **Iulia**, **Kati**, **Lorena**, **Marius**, **Nacer**, **Patrice ;** et à tous ceux qui m'ont donné des conseils précieux et qui m'ont montré leur soutien **Sonja**, **Paul**, **Nacer**, **Frederic P.** et **Frederic R.** Un grand merci à tous les membres du laboratoire TIMA que je n'ai pas cité mais que je n'oublie pas.

Un grand merci à mes amis et partenaires d'entraînement de Boxe Thai qui sont venus m'encourager non seulement à mes combats, mais aussi au jour de la soutenance : **Nadir** et **Christophe**.

Une pensée toute particulière pour mon fiancé **Cosmin**. Nous avons suivi le même chemin à partir de la première année de l'Université Polytechnique de Bucarest où on s'est rencontré la première fois et jusqu'au jour de la soutenance. Il soutient sa thèse de doctorat à une semaine différence, dans un domaine fascinant et très difficile : les nanotechnologies. Je tiens à lui exprimer tout mon amour et ma gratitude pour son soutient, pour son esprit pragmatique, pour le support qu'il m'a montré dans les moments difficiles et pour avoir partagé les moments heureux. Je tiens aussi à lui remercier pour l'exemple qu'il m'a donné dans sa persévérance au travail, pour son courage de prendre les défis et d'explorer des zones de la science, pas encore étudiés. Je lui souhaite une carrière brillante, et que tous ses rêves se matérialisent.

Je finis par remercier à **ma mère** et à **mon père** pour leur constant soutient intellectuel et affectif, et de m'avoir indiqué le chemin de la science encore de mes premiers ans de ma vie et de m'avoir guidé dans mes décisions professionnels. Je remercie à **ma sœur** qui est en train de finir en ce moment son stage chez Philips au Pays Bas. Merci, Ioana, pour tes encouragements, pour la confiance que tu me montres, pour tes conseils pragmatiques, pour ta puissance de décision et je suis fière que tu écoutes bien mes conseils. Et aussi je tiens à te remercier pour tous les vacances qu'on a passé ensemble ces derniers années.

Je pense aussi avec une grande affection à **ma grande-mère**. Je tiens à lui remercier pour tout son amour, son soutient, pour les soucis qu'elle se fait pour moi et pour son confiance qu'elle me montre.

Sommaire

Table des Figures

Liste de Tables

Introduction

1 Evaluation des performances des systèmes MPSoC

Ce travail s'inscrit dans le contexte de la conception des systèmes embarqués multiprocesseur monopuces, que la littérature référence sous le terme de MPSoC[1]. En bref, ces systèmes sont constitués de plusieurs sous-systèmes de nature différente (comme sous-systèmes matériel, logiciel et interconnexion), ce qui en fait un système hétérogène. Dans cette thèse, on s'intéresse aux MPSoCs numériques, avec une partie logicielle tournant sur un ou plusieurs processeurs, et/ou une fonction directement « câblée » en matériel.

Les tendances actuelles dans l'industrie des semi-conducteurs montrent que 90% des nouveaux ASICs[2] incluent déjà un CPU[3] embarqué, réalisé dans la technologie de 130nm [Itrs] [Jon 03]. Les plateformes multimédia comme par exemple Cell [Cell], Nomadik [Noma] et Nexperia [Nex] sont déjà des systèmes MPSoC qui contiennent des processeurs variés, incluant des DSPs[4] et des microcontrôleurs [Jer 04]. A part les processeurs multiples exécutant en parallèle des programmes, ces puces contiennent des blocs matériels et des réseaux d'interconnexion avec des mécanismes de communication sophistiqués, nommés réseaux-sur-puce (NoC[5]).

Les systèmes hétérogènes sont de plus en plus exploités pour satisfaire des fortes contraintes de performance, coût et énergie consommée. Les applications destinées au marché de masse utilisent déjà des MPSoC ; quelques exemples d'applications sont

[1] Multi-Processor System-on-Chip
[2] Application Specific Integrated Circuit
[3] Central Processing Unit
[4] Digital Signal Processors
[5] Network-on-Chip

les terminaux mobiles, les processeurs de jeux, les processeurs vidéo et les processeurs de réseau. Il est crucial de maîtriser la conception de tels systèmes tout en respectant les objectifs de performance et les contraintes de mise sur le marché.

Par conséquence, la conception des SoC[6] sera orientée vers la sélection de la meilleure solution architecturale, composée de meilleurs processeurs et interconnexions. Le terme « performance » est un mot très souvent utilisé dans la conception. La performance est le critère de base pour la conception, la sélection et l'utilisation des systèmes intégrés. Sans doute, les objectifs importants de conception sont de fixer les demandes de performance, de pouvoir comparer différents alternatives et de choisir celui qui respecte le mieux ces demandes. Il est naturel d'associer une phase d'évaluation des performances avec chaque étape de conception pour choisir la réalisation optimale.

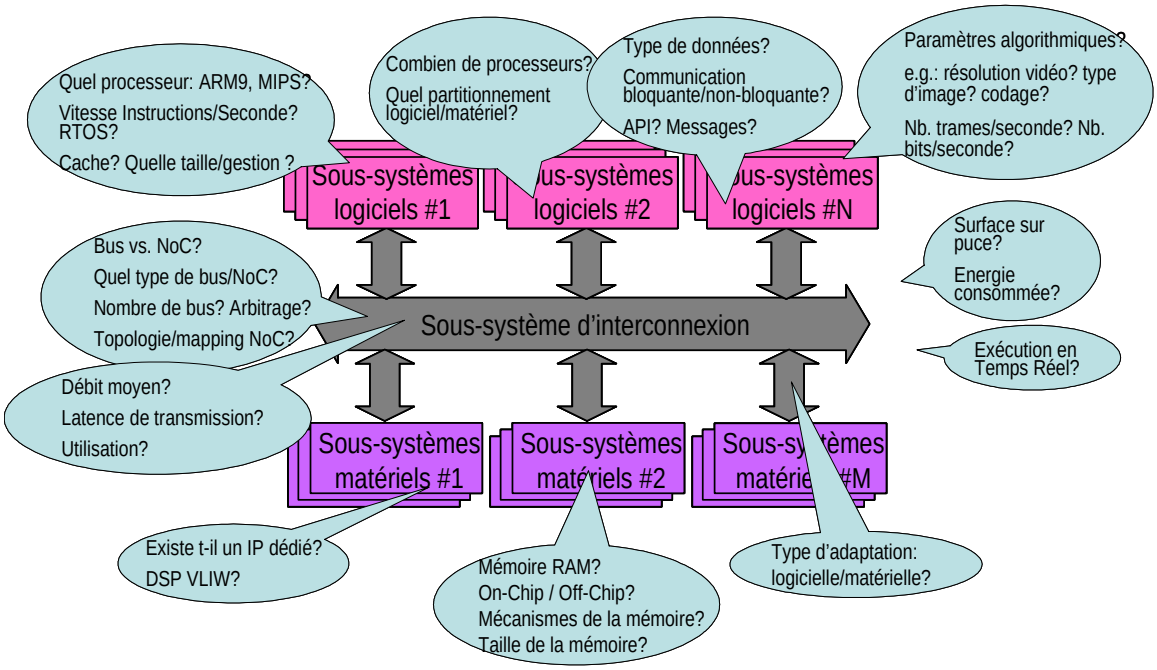


Figure 1. Un système MPSoC générique et ses demandes de conception

Par exemple, la Figure 1 montre un MPSoC dans une phase préliminaire de conception. Il est composé de plusieurs sous-systèmes logiciels et/ou matériels et d'un sous-système d'interconnexion. Tous les paramètres d'architecture ou les algorithmes qui vont s'exécuter sur cette architecture sont des inconnus au moment de la conception. La complexité des décisions à prendre est évidente.

[6] System-on-Chip

On va illustrer ce vaste espace des inconnues de conception par quelques exemples pour mieux expliquer pourquoi il est besoin de l'évaluation des performances et les différents niveaux où l'évaluation des performances intervient: (a) application, (b) architectural et (c) technologique.

(a) Les contraintes au niveau application. Le niveau « application » n'est pas un niveau utilisé pour mesurer de métriques de performance. On le nomme « niveau 0 », car c'est ici qu'on établit les intervalles où les différentes métriques sont censées varier. Ces intervalles sont imposés comme des contraintes au moment de la conception, incluses dans la description de l'application. Ils doivent être respectés à tous les niveaux d'abstraction pendant le raffinement d'architectures. Les compromis de conception en vue de l'optimisation se font par variation des métriques dans les limites imposées par ces intervalles. Par exemple augmenter le temps de traitement des données, tout en respectant des contraintes de consommation d'énergie pour une application en temps réel qui est en plus dédiée à être utilisée sur un téléphone portable.

Ensuite, toutes les décisions sur les paramètres, au niveau architectural ou technologique doivent satisfaire les contraintes définies précédemment au niveau application.

(b) Les contraintes au niveau architectural impliquent l'utilisation des métriques de performance pour optimiser la réalisation de l'application sur un plateforme : le partitionnement logiciel/matériel, les choix des processeurs, des mémoires etc. Pour la réalisation d'une application le traitement vidéo de haute définition, il est essentiel d'utiliser une plateforme hétérogène. Selon [Iwa 03], si l'on considère une approche entièrement logicielle pour implémenter cet encodeur video haute définition, il y aurait besoin de 32,000 processeurs RISC[7] tournant à 1GHz. Ce type de plateforme est à l'heure actuelle irréalisable en termes de surface sur puce et surtout de consommation d'énergie. Par contre, si on considère la solution mixte

[7] Reduced Instruction Set Computer

logicielle/matérielle du même encodeur MPEG2, cela pourrait être implémenté en utilisant seulement quatre processeurs.

(c) Les contraintes au niveau technologique, impliquent l'utilisation des métriques de performance pour optimiser la réalisation finale sur le matériel. Ils imposent la réalisation avec une technologie précise (par exemple 65nm ou 130 nm, etc.), ou des paramètres déterminés pour la fréquence d'horloge, tension d'alimentation, etc. Dans les réalisations actuelles, ces paramètres sont proposés à la conception, et ils sont choisis selon l'expérience du concepteur. Le concepteur propose une architecture convenable et définit les ressources du système. Ces ressources limitent les performances ; par exemple sur un processeur ARM9, on est limités à 220 MIPS[8] pour une fréquence de 200MHz. En plus, les nouvelles technologies rajoutent de la complexité au circuit résultant, et implicitement dans l'évaluation de leurs performances ; par exemple il y aurait besoin de la réalisation des circuits de correction, des concepts de la qualité des services, ou des nouveaux systèmes d'interconnexion qui remplacent les fils trop longs sur la puce.

Malgré tous les outils existants pour l'évaluation des performances, un outil générique, rapide et précis et qui prend en compte le système MPSoC global, n'existe toujours pas. Cela est dû principalement à la conjonction de plusieurs facteurs, décrits comme suit.

1. L'hétérogénéité des systèmes MPSoC. Le fait que les MPSoC sont composés de sous-systèmes différents a engendré une grande diversité d'outils pour l'évaluation des performances de chaque sous-système. Ces sous-systèmes ont des modèles différents en termes de notion de temps (par exemple temps discret et temps continu) et des interfaces de communication. En conséquence il est difficile de les évaluer sur une même plateforme.

Le problème est de trouver l'outil d'évaluation des performances le plus adapté à un sous-système particulier, et de considérer l'ensemble des interactions avec les autres sous-systèmes. Afin de pouvoir gérer la diversité, une étude systématique est nécessaire, concernant les principaux sous-systèmes, les domaines d'applicabilité pour chaque outil et les différents niveaux d'abstraction. Dans le cadre de la recherche

[8] Mega Instructions per Second

actuelle, il serait très utile de trouver des méthodes d'évaluation de performance pour les systèmes globaux MPSoC et aussi de définir des valeurs spécifiques pour les critères de performance.

2. L'incompatibilité des outils. Les incompatibilités des différents sous-systèmes et leurs demandes concurrentes pendant le processus d'évaluation ont fait que pour chaque type de sous-système il existe un outil d'évaluation particulier. Les spécificités de ces outils en termes de modèles de description et contraintes des performances rendent difficile l'utilisation des ces outils pour évaluer un système hétérogène, composé de plusieurs sous-systèmes différentes. De plus, il est impossible de mener une analyse globale si on évalue chaque sous-système individuellement. Cela nécessite un environnement capable d'exécuter concurremment tous les outils d'évaluation et d'analyser le système en ensemble. Une évaluation globale est impérative pour évaluer un système complexe comme MPSoC.

3. Le compromis vitesse d'évaluation versus précision. Dû aux nouvelles contraintes imposées par l'intégration et la complexité croissante des MPSoCs, la littérature prouve qu'une grande partie du temps de conception est passé dans l'évaluation des performances. De plus, les temps de mise sur le marché deviennent plus courts et, par suite, les itérations dans la conception deviennent prohibitives [MPSoC 05].

En conclusion, pour minimiser le temps de conception il est nécessaire de : minimiser le temps de l'étape d'évaluation des performances et les retours arrière doivent être évités. Une étape d'évaluation de performances rapide et moins coûteuse peut être obtenue très tôt au cycle de conception [Itrs 05]. Mais, un modèle de haut niveau n'est pas précis (Chapitre II.1.4). La précision de l'évaluation de performances intervient pour minimiser les retours arrière. Si un niveau bas de conception est utilisé, on aura l'avantage de la précision ; mais la contrepartie est : une vitesse lente de simulation, une conception déjà détaillée et par conséquent des retours en arrière coûteuses, des surcoûts dus à la modélisation du système. Pour cela, il y a souvent le cas où les concepteurs utilisent des modèles de systèmes de haut niveau, annotés avec informations collectionnées des niveaux bas.

2 Objectif

L'objectif général encadrant cette thèse est de proposer une méthode globale pour l'évaluation des performances des systèmes hétérogènes MPSoC. Cette méthode se propose :

- de décrire un environnement global et flexible pour l'évaluation des performances des systèmes MPSoC ;
- de définir une méthode d'évaluation des performances, rapide et précise, pour le sous-système logiciel embarqué ;
- d'explorer l'espace de conception des sous-systèmes d'interconnexion embarqués.

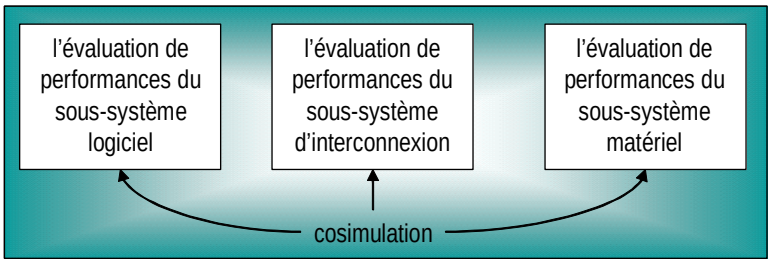


Figure 2. Schéma d'évaluation d'un système global par la cosimulation de différents outils

Le but est de construire un modèle global d'évaluation des performances, capable de réunir des sous-systèmes différents : logiciels, matériels et d'interconnexion. Chaque sous-système est associé à une méthode d'évaluation des performances spécifique. Le modèle global d'évaluation des performances doit offrir l'environnement pour adapter et synchroniser ces sous-systèmes différents et ces méthodes différentes en vue d'une analyse globale des performances. Ce modèle est schématiquement représenté en Figure 2, et il sera détaillé en Chapitre III.

Le moteur à la base de l'évaluation des performances est la mesure de différentes métriques, utilisées ensuite pour optimiser un certain aspect de la conception. Le point clef est de trouver une méthode et de créer l'environnement efficace pour mesurer les métriques. Même si on va le désigner avec le terme « d'évaluation des performances », notre méthode est générique. A part l'évaluation des performances, les métriques mesurées et collectionnées pourront être interprétées pour plusieurs objectifs dans l'analyse des systèmes MPSoC. Cette technique est fondamentalement

la même pour une gamme large des étapes de conception parmi lesquels la validation, la vérification, etc.

Cette thèse s'articulera autour de l'évaluation des performances pour déterminer une architecture optimale avec minimum de ressources : moins de processeurs, moins de mémoire, puissance de calcul réduite etc. Ainsi, dans un premier temps, on évaluera rapidement et précisément les performances du logiciel embarqué, où on peut explorer la réalisation optimale du système d'exploitation. Dans un deuxième temps, en utilisant des métriques mesurées, on est capable de prédire la meilleure réalisation pour le sous-système d'interconnexion sur puce.

On a vu les exemples présentés dans la section antérieure focalisant sur les critères de performance comme : temps, consommation et surface. Dans cette thèse on restreint les performances aux aspects liés aux temps de calcul. L'aspect temporel est analysé en vue de la vérification des contraintes pour les systèmes en temps réel et aussi en vue de l'optimisation du système d'exploitation et du sous-système d'interconnexion.

Dans le flot de conception considéré dans le cadre de cette thèse, les sous-systèmes matériels sont préconçus, et la plupart de temps décrits comme des boîtes noires prêtes à être intégrées dans le MPSoC. Pour ces sous-systèmes matériels, il y a déjà des valeurs des métriques des performances. Pour ces raisons, dans cette thèse, on va s'intéresser aux évaluations de performances pour le sous-système logiciel et d'interconnexion (en Chapitre III et Chapitre IV respectivement).

3 Contributions

Ce travail présente trois contributions majeures :

- (1) la proposition d'une méthode globale d'évaluation des performances par composition des outils individuels dans un environnement global de cosimulation ;
- (2) la proposition d'un modèle rapide et précis d'évaluation des performances pour le logiciel embarqué incluant le système d'exploitation ;

- (3) la proposition d'une stratégie efficace d'exploration (par la permutation parmi différents modèles de performances) des sous-systèmes d'interconnexion embarqués.

Aussi, partie des contributions de cette thèse est une étude systématique des méthodes d'évaluation des performances pour les différentes sous-systèmes composant les MPSoCs. Cette étude a aidé à la définition d'une méthode globale pour l'évaluation des performances des systèmes hétérogènes MPSoC. Chaque contribution sera présentée plus en détail par la suite.

3.1 Etude systématique de l'évaluation des performances

Cette étude couvre les différents sous-systèmes électroniques : logiciels, matériels, CPU, interconnexion et des systèmes MPSoC. Dans une phase préliminaire, une taxonomie couvrant les éléments essentiels de l'évaluation des performances est présenté. Ces éléments sont : les métriques, les paramètres de conception considérés le sous-système à analyser, son niveau d'abstraction, la méthode d'évaluation. Les méthodes typiques d'évaluation des performances considérées sont : statistiques et déterministes (approches par simulation et analytiques).

Le cœur de notre étude est l'analyse de chaque sous-système particulier, en conformité avec cette taxonomie. L'étude couvre aussi les solutions proposées actuellement pour l'évaluation des performances des MPSoCs, en saisissant leurs points forts et leurs points faibles. Cette étude, détaillée en Chapitre II, est à la base de notre proposition d'une méthode globale d'évaluation des performances pour les systèmes hétérogènes MPSoC (Chapitre IV).

3.2 Méthode globale pour l'évaluation des performances des MPSoC

Comme l'on a déjà mentionné, les outils d'évaluation des performances sont généralement adaptés pour évaluer des domaines restreints des sous-systèmes spécifiques. L'hétérogénéité des systèmes MPSoC rend difficile leur analyse par l'application des méthodes standard.

Une méthode pour l'évaluation globale des systèmes MPSoC est proposée et mise en œuvre dans cette thèse. Cette méthode est basée sur la définition d'un modèle de cosimulation où chaque composant est représenté par son modèle d'évaluation des

performances (par exemple un modèle de simulation enrichi avec des annotations temporelles et avec des « sondes de mesure ») connecté avec le reste du système par des interfaces d'adaptation. Les interfaces d'adaptation sont un point clé pour pouvoir simuler ensemble des sous-systèmes différents communiquent par des protocoles différents. Nous proposons donc cet environnement d'évaluation des performances où plusieurs modèles d'évaluation des performances de différents sous-systèmes peuvent être interconnectés grâce à des interfaces d'adaptation flexibles et génériques. Le terme « flexibles » désigne la possibilité de connecter sous-systèmes différents. Le terme « génériques » désigne la possibilité de générer plusieurs instances spécialisées à partir d'une même structure.

3.3 Modèles d'évaluation des performances pour les sous-systèmes logiciel et d'interconnexion

Dans les systèmes MPSoC, le sous-système d'interconnexion et le sous-système logiciel sont complexes. Les deux sous-systèmes ont un comportement fortement dynamique, difficilement prédictible, et de paramètres de conception multiples à tailler. On va présenter nos solutions pour l'évaluation des performances de ces deux types de sous-systèmes.

1. Evaluation de l'interconnexion embarquée. Pour l'évaluation des performances du système global, on considère la plateforme de simulation composée d'une application et d'un réseau d'interconnexion de la manière suivante. Les modèles de différents modules représentant l'application et l'interconnexion sont assemblés et communiquent via des fonctions d'interface (ou d'entrée/sortie) spécifiques. Chacun de ces modèles, de l'application et l'interconnexion a ses propres fonctions d'interfaçage. Le défi est de faire communiquer l'application avec l'interconnexion. Ainsi, des interfaces d'adaptation flexibles et précises ont été développées pour l'intégration de ces sous-systèmes communiquant différemment.

Dans notre approche, on considère l'application prédéfinie, et différents sous-systèmes d'interconnexion prêts à être choisis d'une bibliothèque (comme on verra en Chapitre IV). Les interfaces d'adaptation aident à l'exploration rapide des architectures d'interconnexion, en vue de la sélection d'une architecture optimale.

2. Evaluation de du logiciel embarqué. L'évaluation des sous-systèmes logiciels embarqués, tôt dans les étapes de conception n'est plus faisable en employant les approches traditionnelles de simulation, précises au niveau cycle. Celles-ci sont basées sur l'exécution du logiciel sur des simulateurs de processeurs (ISS[9]) multiples, et de plus, concurremment avec des simulations matérielles, au niveau transaction ou au niveau cycle [SeamCVE] [ConvSC] [SStu]. Le problème vient de la vitesse lente de simulation de l'ISS. Pour une simulation plus rapide, un modèle fonctionnel de haut niveau (appelé « matériel virtuel » en [Row 94]) peut être employé. Dans ce cas, l'application logicielle est compilée et exécutée sur un serveur de simulation hôte, entraînant d'habitude une vitesse élevée de simulation. Mais en ignorant l'architecture réelle du matériel, cette technique manque d'exactitude.

Le défi est alors, de trouver une méthode d'évaluation plus rapide que des méthodes traditionnelles de cosimulation, mais sans sacrifier la précision. Une évaluation rapide est nécessaire pour l'exploration d'un vaste espace de conception. La précision de l'évaluation certifiera l'exactitude de la conception.

Notre contribution consiste dans le développement d'un modèle de simulation rapide et précis, en exploitant l'exécution native et les annotations temporelles. L'exécution native est la simulation de l'application sur le processeur hôte (par exemple un processeur Sparc sur une station de travail Sun), au lieu de le simuler sur le processeur cible de la vraie architecture (par exemple un processeur ARM7). Dans notre approche, les annotations temporelles sont les temps d'exécution, déterminés pour le code réel de l'application et du système d'exploitation correspondant évalués pour la vraie réalisation (par exemple sur le processeur ARM7). Nous avons développé un outil d'annotation automatique, nommé ChronoSym. Aussi, nous avons implémenté une interface de cosimulation pour considérer les interactions du logiciel avec le matériel.

4 Plan du document

Le rôle du **Chapitre I** est de présenter l'environnement de conception décrit avec l'outil ROSES, développé au sein du groupe SLS[10]. Il présente tous les détails

[9] Instruction Set Simulator
[10] System Level Synthesis (group)

nécessaires à l'implémentation et à l'interfaçage de notre méthode d'évaluation des performances : le flot de conception des systèmes MPSoC, leur modèle de description, l'étape de cosimulation logicielle/matérielle que l'on va utiliser intensivement dans notre méthode d'évaluation des performances. On présente aussi les sous-systèmes que l'on considère critiques du point de vue de l'évaluation des performances : le sous-système logiciel et le sous-système d'interconnexion. Comme cette thèse focalise sur le sous-système logiciel et d'interconnexion, pour des raisons de brièveté, nous n'avons pas donné une section relative au sous-système matériel.

Le **Chapitre II** est une étude des outils d'évaluation des performances pour différents sous-systèmes composants des MPSoC. Tous les éléments impliqués dans l'évaluation des performances sont présentés en cette section. Ils sont structurés dans une taxonomie. Cette étude, vue par la taxonomie proposée nous permettra de fixer les problèmes et les manques des outils actuels, et de proposer une méthode d'évaluation des performances.

Le **Chapitre III** présente un modèle d'évaluation des temps d'exécution, basé sur la simulation, pour le sous-système logiciel embarqué. Dans le cadre de ce chapitre, on présente notre outil pour la génération automatique du modèle de performances du logiciel. Nous appliquons cet outil à l'exploration du sous-système d'exploitation et à l'application embarquée.

Les premiers chapitres servent comme base pour proposer, en **Chapitre III,** une méthode globale d'évaluation des systèmes hétérogènes MPSoC, basée sur l'assemblage des outils. Notre méthode est appliquée à l'exploration des architectures des sous-systèmes d'interconnexion embarqués. Cela est un exemple. Mais, comme c'est une méthode flexible, rapide et précise elle pourra être appliquée à n'importe quel sous-système, ou même à l'exploration de l'architecture MPSoC en totalité.

CHAPITRE I.

La conception des systèmes MPSoC

Introduction

Ce chapitre présente les étapes de conception des systèmes MPSoC à l'aide de l'outil ROSES, développé au sein du groupe SLS. Les principales étapes de ce flot sont analysées, en vue de leur utilisation dans l'évaluation des performances :

- (a) la description des MPSoC à base d'architecture virtuelle, fondement pour la plateforme d'évaluation des performances ;
- (b) la génération des interfaces de cosimulation, des interfaces logicielles et matérielles, utilisés et/ou adaptés pour l'évaluation des performances ;
- (c) la cosimulation logicielle/matérielle, qui servira comme technique de base pour l'évaluation des performances par composition.

Un autre point important de ce chapitre est la définition des sous-systèmes « critiques » dans l'évaluation des performances : le sous-système logiciel et d'interconnexion. Ces deux sous-systèmes sont complexes à cause de leur comportement dynamique et temps d'exécution non-prédictibles. On convient qu'il y a deux principales difficultés qui émergent en définissant une méthode globale pour les MPSoC hétérogènes :

- (1) la spécification du modèle de simulation pour le sous-système logiciel ;

- (2) la grande diversité des paramètres du sous-système d'interconnexion.

La structure des sections reflète les deux classifications précédentes. La première section sera consacrée à la problématique de la conception des MPSoC avec l'outil ROSES, en détaillant le flot de conception pour ces systèmes, leur modèle de description et l'étape de cosimulation. Les sections deux et trois présentent les détails de conception pour les sous-systèmes auxquels on s'intéresse pour l'évaluation des performances : le sous-système logiciel et d'interconnexion. Ces détails seront nécessaires pour bâtir leurs modèles d'évaluation des performances.

1 Conception des systèmes MPSoC à l'aide de l'outil ROSES

Cette section offre une vue d'ensemble sur la conception MPSoC, avec l'outil ROSES. On présentera dans la première sous-section le modèle de description des systèmes hétérogènes MPSoC, qui est composé des modules à réaliser en matériel, des modules à mapper sur des CPUs et de sous-systèmes variés d'interconnexions. Une telle description hétérogène est possible par l'utilisation des concepts de l'architecture virtuelle qui sera présentée en détail dans cette première sous-section. La deuxième sous-section présente la manière de cosimuler les systèmes hétérogènes ; les interfaces utilisées pour la cosimulation sont détaillées dans cette sous-section. Finalement, la troisième sous-section montre la vue d'ensemble du flot de conception MPSoC : à partir de la description, grâce aux outils de synthèse automatique des interfaces logicielles et matérielles, on génère l'architecture de niveau RTL[1]. Les interfaces de cosimulation sont générées automatiquement, aux différents niveaux d'abstraction, au long de ce flot.

1.1 Modèles de description des systèmes MPSoC

Cette section présentera le modèle de description d'un MPSoCs et son modèle de simulation. Le « modèle de description » est la représentation du circuit à réaliser dans un langage de haut niveau. Il a une partie comportementale, et une partie

[1] Register Transfer Level

architecturale. La partie comportementale est décrite par un ensemble des fonctions, alors que la partie architecturale implique une étape de partitionnement effectuée auparavant pour associer les fonctions à être exécutés sur un ensemble des processeurs et/ou blocs matériels.

Le « modèle de simulation » est l'application du modèle de description sur des simulateurs spécifiques. Il y a souvent le cas où la translation est directe, par exemple une description VHDL sera simulée par un simulateur VHDL, ou une description en langage C sera compilé et exécutée (par exemple sur un simulateur de processeur). Dans le cas où un mapping direct ne peut pas être envisagée, le modèle de simulation sera développé comme une description équivalente du circuit, du point de vue fonctionnel et temporel. Ce dernier cas peut correspondre au cas où le système est trop complexe pour être simulé en totalité. Par exemple, un circuit spécialisé de calcul de FFT (en N points et K bits) peut être remplacé par un programme ou une fonction déjà existante dans une bibliothèque mathématique (en C ou en Fortran) (réalisant toujours le calcul en N points et K bits donnés comme des paramètres pour cette fonction).

L'environnement ROSES permet une description hétérogène des systèmes MPSoC, englobant plusieurs modèles de description et aussi plusieurs modèles de simulation : à des niveaux d'abstraction différentes et/ou décrites en différents langages. Le cadre qui offre cette possibilité est l'utilisation du concept de l'architecture virtuelle.

On nomme **architecture virtuelle** l'organisation du système où les différents modules (logiciels ou matériels) n'ont pas encore des interfaces bien définies. On ne spécifie pas encore ces interfaces pour des raisons de flexibilité, pour pouvoir connecter les différents modules sans être obligés de concevoir leurs vraies interfaces, leurs vrais ports ou leurs vrais canaux de communication. L'architecture virtuelle a comme composants de base : (1) les modules virtuels, (2) les ports virtuels et (3) les canaux virtuels.

Un système représenté à l'aide de l'architecture virtuelle (Figure 3) est décrit comme un ensemble de modules (hiérarchiques) virtuels interconnectés, en utilisant des canaux virtuels point-à-point et/ou des réseaux de transmission reliés aux ports virtuels.

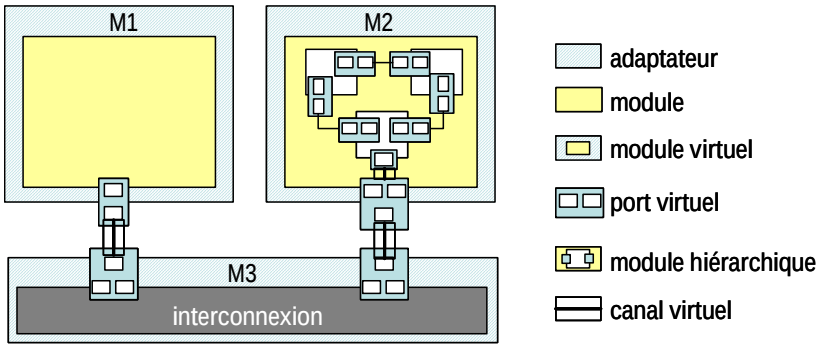


Figure 3. Le modèle de description basé sur l'architecture virtuelle

En cette thèse, le langage de description utilisé pour l'architecture virtuelle est VADeL[2] [Ces 02]. VADeL est un langage de haut niveau qui peut décrire les systèmes d'une manière hétérogène. Il a été créé comme bibliothèque au-dessus de SystemC [SystemC], rajoutant les modules virtuels, les ports virtuels et les canaux virtuels avec plusieurs niveaux de hiérarchie.

Un **module** est une entité décrite par une structure et un comportement. La structure d'un module peut être un ensemble d'autres modules interconnectés par leurs ports (comme par exemple pour module M2), ou une feuille dans la hiérarchie, représentant un comportement élémentaire (comme par exemple M1). Le comportement peut être décrit par des tâches logicielles ou des fonctions matérielles ; par exemple les modules à l'intérieur de M2 peuvent être des tâches qui s'exécuteront sur un processeur, ou des fonctions à l'intérieur d'un bloc matériel.

Dans les systèmes hétérogènes MPSoC, les modules communiquent de manière hétérogène. Pour adapter la communication du module à la voie de transmission, une couche d'adaptation y est associée. **Le module virtuel** est l'abstraction du module considéré, ensemble avec la couche d'adaptation à la communication. La fonction de cette couche d'adaptation est de lier les accès internes du module aux canaux externes pour la communication inter-modules. L'adaptation peut être nécessaire par rapport aux niveaux d'abstraction, protocoles de transmission et langages de spécifications. Par exemple, on veut connecter un module (logiciel) qui a la taille du port d'entrée d'un entier de 32 bits, avec un module (matériel) qui a la taille du port de sortie d'un octet, en mode « burst »/ou non. Il faut alors décrire les interfaces d'adaptation pour la

[2] Virtual Architecture Description Language

prise en compte du mode « burst »/ou non, la conversion de quatre octets vers un entier de 32 bits signé, et la détection d'éventuelles erreurs d'envois successives d'octets.

Pour la communication, chaque module possède un ensemble de ports, réunis sous des ports virtuels. **Le port virtuel** regroupe plusieurs ports internes, liés aux modules internes, et plusieurs ports externes, liés aux canaux d'interconnexion virtuels. Le mapping entre les ports internes et les ports externes peut être 1:1 ou n:m. Dans le dernier cas, une fonction de résolution pour le canal d'interconnexion doit être mise en oeuvre. Par exemple, en Figure 3, pour le port d'interface du module M2 : les ports internes correspondent 1:1 aux ports externes, cas où la donnée sera transmise sans modifications supplémentaires. Au contraire, pour le port d'interface du module M1, les ports internes sont en nombre de 3 (correspondant par exemple au protocole « handshake » – un port pour les données et deux pour le contrôle : demande et acquittement), tant qu'il y a un seul port externe (par exemple pour les données). Dans ce dernier cas de correspondance 3 :1, il est besoin d'une fonction de résolution qui indique quelles données mettre sur ce port (externe), en respectant les conditions du protocole établi par le module (par ses ports internes).

Les canaux virtuels groupent un certain nombre de canaux externes qui ont la même dépendance logique, par exemple le même protocole ou la même paire de source/destination. Ils abstraient également des détails de la transmission, en utilisant des primitives de communication de haut niveau comme par exemple le transfert des messages. Par exemple, ils cachent les protocoles (comme « handshake », transmissions série ou parallèle), les temps de transmission, etc.

Le modèle de l'architecture virtuelle est utile pour l'automatisation de la conception, puisqu'il permet l'interconnexion de divers modules virtuels définis dans de langages différents et/ou à différents niveaux d'abstraction. L'automatisation de la conception a des implications dans de raffinement des architectures et aussi dans la génération (automatique) des modèles de simulation. Le raffinement des architectures consiste en à partir d'une description de haut niveau, et suite à des raffinements successifs de générer le modèle de bas niveau de l'architecture MPSoC, prêt à être synthétisé. Par l'utilisation de l'architecture virtuelle, des modèles de simulation sont générés

automatiquement à partir des modèles de description, à différents instants du raffinement architectural.

1.2 Cosimulation logicielle/matérielle des systèmes MPSoC

La cosimulation est une technique qui permet la simulation parallèle et synchronisée avec plusieurs simulateurs différents : par exemple pour le logiciel et le matériel. Puisque ces simulateurs (du logiciel et du matériel) échangent des données, l'adaptation et la synchronisation est nécessaire. Ceci est habituellement exécuté à l'aide des interfaces de cosimulation et de l'environnement de cosimulation. Tous ces éléments seront illustrés pour l'outil ROSES dans la suite de cette section.

Les méthodes conventionnelles de cosimulation sont synthétisées en [Nic 01]. Les différents modèles actuels de cosimulation logicielle/matérielle, multi-niveau, multi-langage et multi-modèles sont présentés de manière détaillée en [Nic 03].

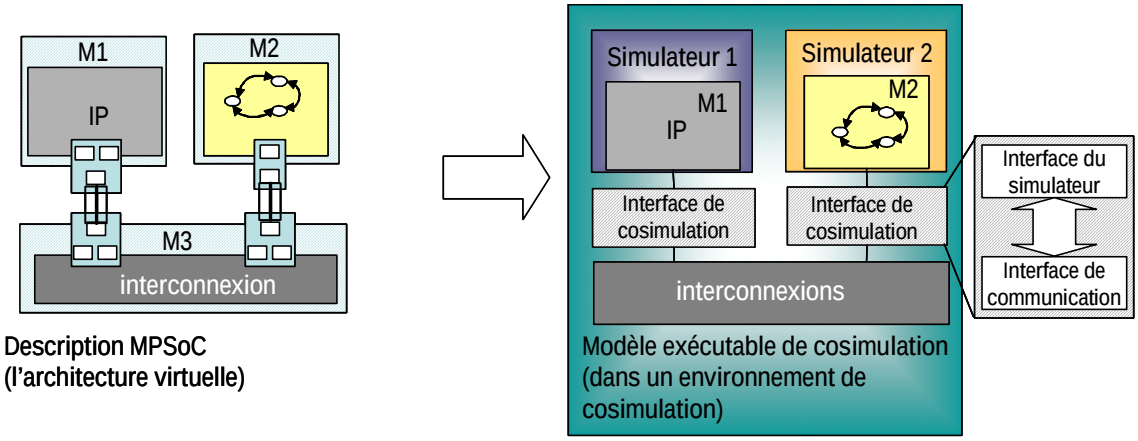


Figure 4. Spécification hétérogène et modèle de cosimulation correspondant

En ROSES, le modèle de simulation des systèmes embarqués hétérogènes est défini comme suit. Les différents simulateurs simulent les composants logiciels et matériels d'un système MPSoC (Figure 4). La partie logicielle d'un système, contenant les différentes tâches et éventuellement le système d'exploitation qui seront exécutés sur un CPU, est simulée à l'aide d'un ou plusieurs simulateurs de processeur (ISS[3]). La partie matérielle (les IPs[4] matériels) est simulée sur un ou plusieurs simulateurs de matériel (par exemple un simulateur de VHDL, SystemC etc.). Ces simulateurs

[3] Instruction Set Simulator
[4] Intellectual Property

différents communiquent via des interfaces de cosimulation avec une fonctionnalité complexe qui assure l'adaptation entre simulateurs et de la communication (plus exactement des niveaux d'abstraction ou des protocoles de communication).

Aujourd'hui la méthode conventionnelle pour l'interconnexion des modèles différents de simulation, surtout pour la validation de l'interfaçage logiciel-matériel, est constituée par le modèle fonctionnel de bus BFM[5] [Nic 03] [Sem 00]. Le BFM est l'interface de cosimulation ; il est donc spécifique à la cosimulation et n'apparaîtra pas dans la vraie architecture.

Le BFM permet l'échange de données entre deux types d'interfaces : l'interface du simulateur et l'interface de communication. Il est décrit par l'ensemble de signaux assurant le transfert des données ou appels de primitives de communications entre les différents éléments d'une instance de cosimulation. Comme une illustration, il permet la transformation des accès fonctionnels des pilotes du système d'exploitation en accès au niveau du cycle sur les signaux du bus du processeur.

En ROSES, les interfaces de cosimulation sont générées automatiquement à partir des modèles de description maintenus dans les bibliothèques des enveloppes pour les différents composants [Nic 03].

1.3 Flot de conception ROSES pour les systèmes MPSoC

L'outil ROSES [Ces 02] définit le cadre conceptuel de réalisation des MPSoCs, par assemblage de composants standard interconnectés les uns avec les autres. Les composants peuvent avoir des origines différentes, décrits à différents niveaux d'abstraction et utilisant protocoles de communication différents. Pour faciliter l'assemblage de ces composants hétérogènes, il faut utiliser les concepts d'architecture virtuelle (décrits dans la section précédente Chapitre I.1.1).

L'entrée du flot de conception (Figure 5) est la macro-architecture, c'est-à-dire la description non-détaillée, de haut niveau du système. Elle peut être écrite manuellement ou générée automatiquement en utilisant des outils de synthèse des systèmes, comme par exemple VCC[6] [VCC]. Nous décrivons la macro-architecture du

[5] Bus Functional Module
[6] Virtual Component Codesign

système par son architecture virtuelle : comme un ensemble des modules virtuels communicant par des canaux virtuels via des ports virtuels, en VADeL.

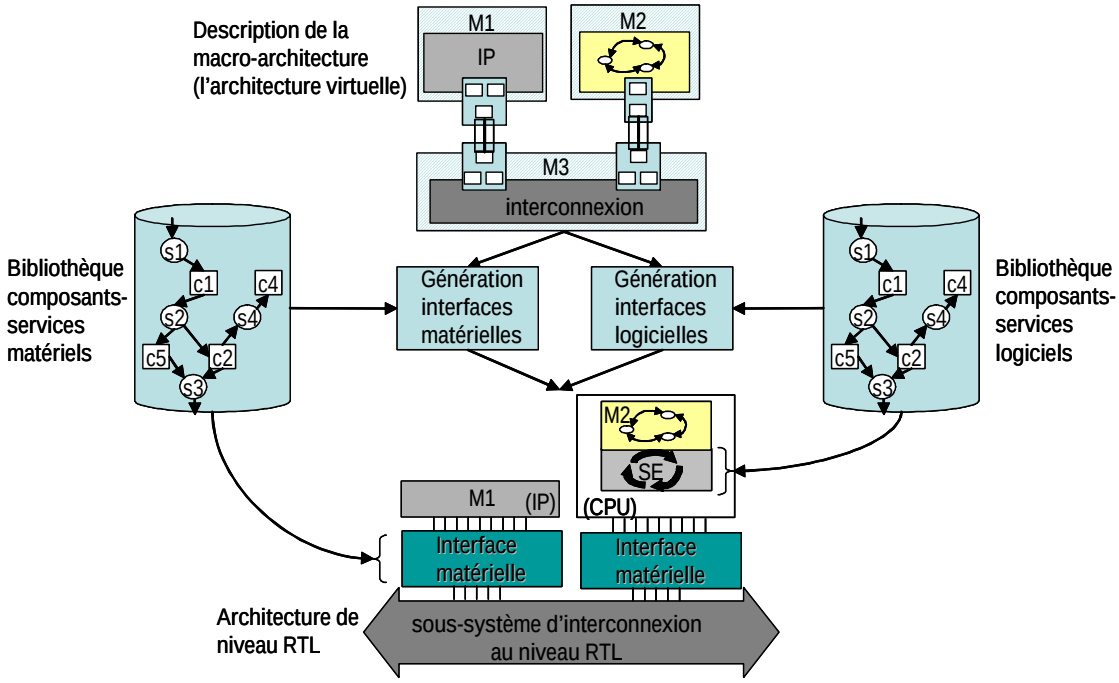


Figure 5. Flot de conception des systèmes embarqués logicielles/matérielles

Après que chaque module est associé avec un IP matériel ou un CPU et son logiciel, l'outil ROSES va générer automatiquement la microarchitecture du système en assemblant ces composants et par la génération automatique des interfaces logicielles/matérielles. Cela constitue la réalisation des interfaces d'adaptation dans l'architecture finale, entre modules décrits d'une manière hétérogène dans la l'architecture virtuelle. Ensuite, les canaux de communication virtuels sont transposés sur un réseau d'interconnexion spécifique, comme par exemple un bus ou un réseau-sur-puce.

ROSES comporte trois outils de génération automatique des interfaces : (1) un outil pour la génération des interfaces logicielles, (2) un outil pour la génération des interfaces matérielles et (3) un outil pour la génération des modèles de cosimulation.

- **l'outil de génération des interfaces logicielles,** décrit en [Gaut 01]. L'interface logicielle est le systèmes d'exploitation (SE) spécifique et pré-configuré pour chaque module logiciel s'exécutant sur un processeur. Cet outil sera cité aussi dans le

Chapitre III de cette thèse, qui illustrera son utilisation pour la génération des modèles de simulation pour les SEs.

- **l'outil de génération des interfaces matérielles,** décrit en [Gras 05]. L'interface matérielle est l'interface d'adaptation entre le CPU et le matériel, qui fait la translation de protocole et la résolution des canaux de communication.
- **l'outil de génération des interfaces de cosimulation**, décrit en [Nic 03] et [Sarm 05]. Le modèle de cosimulation réunit tous les modèles de simulation, et leur interfaces d'adaptation et synchronisation. L'outil de génération des modèles de cosimulation fournit ces interfaces. Par rapport aux deux premières interfaces citées (logicielles et matérielles) qui apparaîtront dans la vraie architecture du système, les interfaces de cosimulation ont un intérêt seulement pour la cosimulation du système hétérogène, qui s'exécute sur plusieurs simulateurs différents.

Tous ces trois outils de génération automatique reposent sur le modèle de description de l'architecture virtuelle. Les outils de génération automatique se servent des éléments prédéfinis des bibliothèques et des annotations du concepteur dans l'architecture virtuelle pour sélectionner les composants des bibliothèques, pour générer les interfaces logicielles, matérielles et de cosimulation.

Les bibliothèques contiennent toutes les parties de code génériques qui peuvent être spécialisés et assemblés pour obtenir l'interface complète.

- **La bibliothèque logicielle** est structurée en trois parties : (1) les primitives de haut niveau, (2) les services de communication/système, et (3) les paquetages pour la communication avec le matériel. Cette structure sera présentée en Chapitre I.2. Elle est décrite en détail en [Gaut 01].
- **La bibliothèque matérielle** est structurée en trois parties : (1) les fonctions d'adaptation au CPU, (2) les fonctions d'adaptation au réseau d'interconnexion et (3) les fonctions d'adaptation au

protocole de communication. Elle est décrite en détail en [Lyo 03] et [Gras 05].

- **La bibliothèque de cosimulation** est aussi structurée en trois parties : (1) une partie relative au simulateur, (2) la deuxième relative à l'environnement de cosimulation et (3) la troisième partie qui fait la translation et la synchronisation entre les deux premiers parties. Elle est décrite en détail en [Nic 03].

Ces bibliothèques sont basés sur trois concepts, qui donnent chacun une vue différente pour l'interface générée : (a) l'élément, (b) le service et (c) la réalisation. Ces trois concepts ne sont pas indépendants : chaque élément peut requérir des services et fournir des autres services. Les relations entre éléments et services sont représentées sous la forme de graphe orienté, ayant comme nœuds des éléments ou des services et comme arcs les relations de demande/offre de services.

- **(a) L'élément** représente une partie structurelle de l'interface, de sorte que l'interface complète est modélisée par un ensemble des éléments interconnectés.
- **(b) Le service** représente une fonctionnalité de l'interface, de telle sorte que le comportement fonctionnel de l'interface peut être modélisé comme un ensemble de services.
- **(c) La réalisation** représente un assemblage particulier du comportement de l'interface. La réalisation peut être compatible avec certaines architectures matérielles, et incompatibles avec les autres. Les portions de code générique sont toujours associées aux réalisations particulières. La réalisation des interfaces peut générer trois classes différentes : (i) interfaces fonctionnelles, (ii) interfaces réalisées seulement en logiciel et (iii) interfaces réalisées seulement en matériel.

Cette manière de représentation des interfaces, à base de éléments et services, assure une représentation homogène pour les interfaces d'adaptation du système. Ce mécanisme est utilisé pour optimiser la taille des interfaces générées, en évitant la présence des éléments inutilisables. Il est possible de définir certaines réalisations,

uniquement si un service fournit par l'élément correspondant est requis par un autre élément. Cela permet que lorsque un élément fournit un service non-requis de ne pas mettre le code correspondant dans l'interface générée.

2 Le logiciel embarqué

Dans les MPSoC, le sous-système logiciel est de plus en plus important et sa performance devient dominante pour le système entier. Il y a souvent le cas, où le système contient des dizaines des processeurs en exécutant en parallèle plusieurs tâches complexes. La conception du SE est complexe puisqu'elle inclut des fonctionnalités sophistiquées telles que l'ordonnancement des tâches, la synchronisation, la gestion d'interruptions, la gestion de la mémoire et la gestion des entrées/sorties [Gau 01].

Dans le groupe SLS, le sous-système logiciel embarqué est composé principalement de l'application spécifiée par l'utilisateur et du système d'exploitation (SE). Le SE peut être généré en configurant un SE existant, configurable ou en développant un SE spécifique à l'application. En ROSES, le SE est conçu comme l'interface logicielle spécifique à l'application.

Dans les sous-sections suivantes nous allons présenter le système d'exploitation spécifique à l'application, en détaillant : son flot de conception, et son composition en parties spécifiques et non-spécifiques du processeur sur lequel il va s'exécuter. Ensuite, ses différentes couches hiérarchiques et ses niveaux d'abstraction sont expliqués.

2.1 Système d'Exploitation spécifique à l'application

Le SE est l'ensemble des programmes qui permettent à l'application logicielle de s'exécuter sur un matériel donné (un CPU et l'ensemble matériel d'entrées/sorties). La méthodologie que l'on utilise en ROSES pour la génération automatique du SE est décrite en [Gau 01] et plus en détail en [Gaut 01].

En ROSES, le SE est conçu pour être spécifique à l'application. C'est-à-dire ayant des fonctionnalités spécifiques et optimisées pour l'application pour quelle il est employé. On verra dans cette section que le caractère spécifique est fourni par l'assemblage des éléments de base, selon les besoins de l'application. L'assemblage des éléments se fait

automatiquement, à partir d'une bibliothèque contenant les éléments et services de base.

L'assemblage des éléments de base donne le caractère modulaire du SE. La modularité implique une réalisation indépendante pour les différents services, comme par exemple : différentes politiques d'ordonnancement, différents pilotes d'entrée/sortie, etc. La modularité permet une réalisation du SE par des incrémentations successives. Cela suppose une procédure de type évaluation-réévaluation pour obtenir un SE minimal et performant.

De plus, le SE doit être conçu (en partie) indépendamment du matériel sur quel il va s'exécuter. La principale raison est de permettre la portabilité du SE et de l'application logicielle sur plusieurs architectures pour faciliter l'exploration de différentes architectures matérielles avec le même code logiciel.

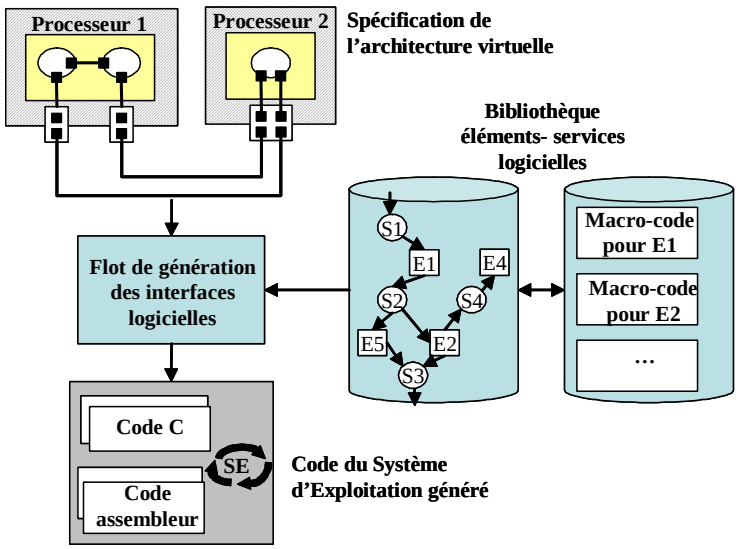


Figure 6. La génération du système d'exploitation dans le groupe SLS

Cette méthodologie utilise trois représentations en entrée :

- (1) la spécification d'entrée du module qui sera mappé sur un processeur, décrit par son **architecture virtuelle**. Cette spécification décrit la topologie de l'architecture cible.
- (2) **la bibliothèque du SE** qui permettra le raffinement du SE. Elle fournit qui, paramétrés, constitueront le code final du SE. Détails sur cette bibliothèque se trouvent en [Gaut 01].

- (3) la représentation utilisée pour le code généré dépend de **la représentation comportementale** choisie pour des éléments de bibliothèque : par exemple ils auront un comportement décrit en langage C ou en langage assembleur. La flexibilité du flot permet de prendre n'importe quel langage de programmation pour cette représentation et même d'utiliser plusieurs langages différents dans la même bibliothèque.

La sélection des services de la bibliothèque engendre un SE spécifique à l'application. Pour chaque service, plusieurs réalisations sont possibles. Les services composant le SE peuvent être (a) spécifiques ou (b) non-spécifiques au processeur.

- Pour **les services spécifiques au processeur**, la réalisation diffère d'un processeur à un autre. Par exemple, une grande partie des services spécifiques au processeur sont décrits en code assembleur.
- La réalisation d'un **service qui n'est pas spécifique à un processeur** reste la même pour tous les processeurs, sa spécification étant de règle en langage C.

Le SE utilisé dans notre approche, contient 90% du code non-spécifique au processeur, par rapport au code total du SE. Ce partitionnement en services spécifiques et non-spécifiques au processeur est choisi pendant la description du SE, par le concepteur des bibliothèques de SE. Le critère à base de ce choix est d'un côté l'aspect généricité du SE (qu'il soit portable, indépendant d'architecture), et d'un autre côté l'aspect optimalité (que la réalisation sur une architecture particulière soit optimale, et utilise bien les ressources de l'architecture).

2.2 Hiérarchie du sous-système logiciel embarqué

Le sous-système logiciel résultant est hiérarchisé en plusieurs couches d'abstraction. Ces couches sont organisés sous la forme d'une pile protocolaire, allant de

l'application logicielle, jusqu'au code dépendant du matériel (HAL[7]) [Gaut 01]. Chaque couche ne peut interagir qu'avec ses voisines.

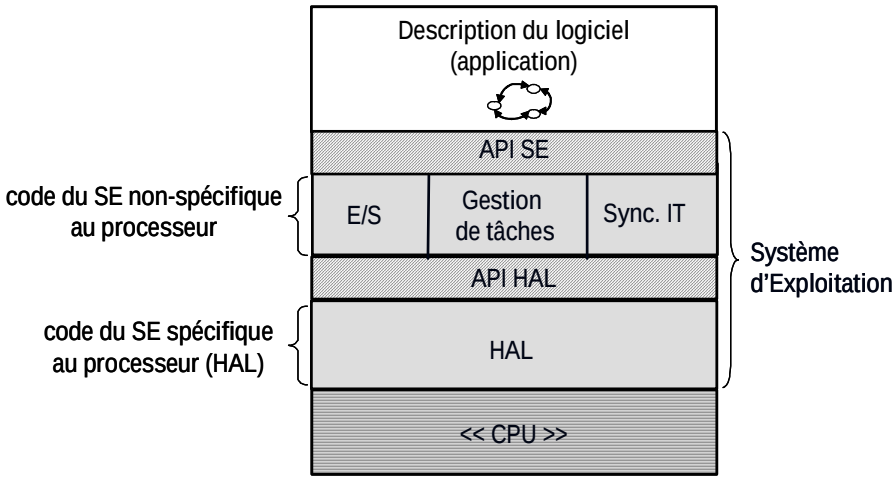


Figure 7. Organisation en couches du sous-système logiciel embarqué

La couche de l'application logicielle est au plus haut niveau d'abstraction, nommé « niveau application », ou niveau « HLL[8] » (car elle est programmé dans un langage de haut niveau – HLL). Par exemple, un programme développé pour l'application logicielle peut uniquement appeler une procédure de niveau moyen (niveau SE/HAL) avec des conventions fixés par son API[9] (l'API du SE). Mais ce programme application ne peut pas appeler directement les procédures du niveau bas (HAL). Et ensuite, chaque niveau est bâti sur l'API du niveau immédiatement inférieur. Aussi, dans ce cas on remarque une séparation évidente entre les couches du SE et du HAL.

Dans la Figure 7 est représentée une vue simplifiée du sous-système logiciel. Les différentes couches et niveaux d'abstractions sont indiqués. Les couches du logiciel sont détaillées ensuite :

- La première couche du sous-système logiciel embarqué est celui de **l'application logicielle.** L'application est conçue en utilisant les APIs du SE.
- Pour permettre la conception modulaire et incrémentale du SE, **le SE** est divisé à son tour dans un ensemble de couches et des APIs hiérarchiques que l'on va présenter en ordre : (1) ***l'API du***

[7] Hardware Abstraction Layer
[8] High Level Language
[9] Application Programming Interface

SE, (2) ***la couche du noyau du SE***, (3) ***l'API du HAL*** et (4) ***la couche d'abstraction du matériel HAL***.

(1) **L'API du SE** représente les appels système de haut niveau, invoqués par l'application (par exemple les APIs POSIX).

(2) **La couche du noyau (« Kernel ») du SE** offre les services de base pour faire tourner les applications utilisateur ou système, ainsi que pour gérer d'une manière efficace les ressources matérielles sous-jacentes. Parmi les services qu'elle contient, il y a des services liés à l'ordonnancement des tâches, des services liés aux interruptions ou aux appels système, des services liés à la gestion de la communication ou des périphériques.

(3) **L'API du HAL** représente les appels de bas niveau, invoqués par le SE pour accéder aux ressources matérielles. L'ensemble des APIs HAL sont fournies par la couche HAL au composant logiciel virtuel qui se trouve au-dessus. Pour la conception d'un système embarqué, on peut employer les APIs d'un HAL standard, par exemple HdS[10]-API in VSIA[11] [VSIA], ou des APIs de HAL pour un SE spécifique au fournisseur [ECos], ou un SE spécifique à l'architecture du système où il est intégré, comme dans notre cas.

(4) La couche la plus basse est **la couche d'abstraction du matériel (HAL)**. HAL est une abstraction de l'architecture matérielle sur quelle va s'exécuter le sous-système logiciel. Nous définissons le HAL comme tout le logiciel qui dépend directement du matériel. Cette couche fournit les pilotes et les contrôleurs pour la gestion de la communication, comme par exemple la gestion d'interruptions, les périphériques matériels comme entrées/sorties. Par ailleurs, la couche HAL contient l'information sur les adresses physiques réelles du système (par exemple Scatter Loading File [Pet 06]).

[10] Hardware-dependent Software
[11] Virtual Socket Interface Alliance

3 Le sous-système d'interconnexion

Cette section présente des notions préliminaires sur les sous-systèmes d'interconnexion embarqués. D'abord on définit un sous-système d'interconnexion embarquée en précisant ses différentes architectures qui peuvent être incorporés dans le système MPSoC dans l'outil ROSES. Après que l'on explique brièvement comment on peut sélectionner une structure particulière d'interconnexion, on présente la méthode de description d'un réseau d'interconnexion, à l'aide de la méthodologie OCCN[12][13].

Le sous-système d'interconnexion embarqué est tout fil de connexion ou circuit avec le rôle de transmission de l'information. Sur une puce, la transmission de l'information peut se faire : via des connexions point-à-point entre modules communicants, via des bus partagés avec des politiques de arbitrage et de gestion de ressources ou via des réseaux-sur-puce.

Les interconnexions point-à-point se matérialisent dans des canaux séparés de communication pour chaque paire de ports communicants. Ils sont les moins coûteux en effort de conception ; mais ils sont loin d'être optimaux en termes de surface occupée et effets parasites prononcés dûs aux fils globaux traversant des distances significatives.

Le moyen de transmission embarqué, le plus usuel est le bus partagé, où tous les sous-systèmes partagent le même media de transmission. Parmi ses avantages, il y a : la topologie simple, la surface occupée, son extensibilité et ses possibilités de hiérarchisation. Ses défauts, comptent parmi les possibilités de collision, aussi les effets parasites accentués par un bus trop long ou trop des sous-systèmes connectés.

Les réseaux-sur-puce (NoC) sont un nouveau paradigme de conception, qui utilise les modèles et techniques des réseaux informatiques, réévalués pour les fortes contraintes d'intégration des MPSoC. Dans ce contexte, la surface occupée et l'énergie doivent être minimisées d'une part et d'une autre part la fiabilité de la transmission doit être assurée [Wor 04] [Wor 05]. Ainsi, les NoCs bénéficient d'un parallélisme explicite, et

[12] On-Chip Communication Network

[13] Le projet OCCN fournit un environnement sous licence publique GNU-GPL, open source, pour la description, modélisation, simulation et l'exploration des architectures des réseaux-sur-puce. Il est basé sur SystemC 2.0.

la modularité pour réduire au minimum l'utilisation des fils globaux. Le principe de la localité est utilisé pour la minimisation d'énergie consommée [Ben 02] [Cop 03] [Hem 00] [Rij 03] [Wor 05].

Les NoCs sont utilisés pour intégrer plus efficacement des sous-systèmes logiciels ou matériels sur une seule puce. Le découplage entre ces sous-systèmes et l'interconnexion est explicite. L'abstraction des ressources et des structures de données contribuant à la transmission, aussi que l'organisation hiérarchique des protocoles facilite la spécialisation.

Structurellement, les NoCs sont constitués par un ensemble des interfaces réseau (NIs[14]) et routeurs connectés par une topologie d'interconnexion, et leurs paramètres correspondants (par exemple la taille des buffers, la taille des données, les nombres des canaux virtuels. Les différentes approches utilisent diverses stratégies de routage ou topologies de réseaux. La communication est réalisée par échange de paquets [Hor 04].

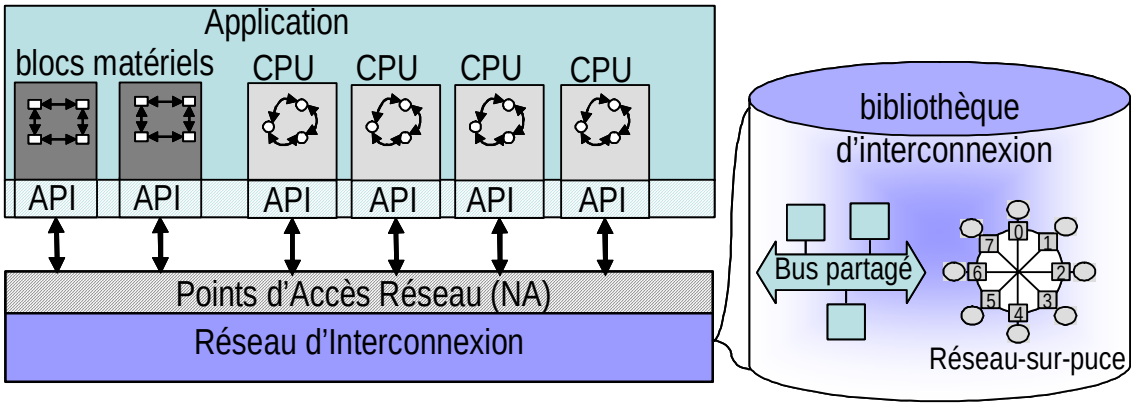


Figure 8. Schéma générique de l'exploration des différents sous-systèmes d'interconnexion

En ROSES, nous disposons des bibliothèques de modèles de descriptions pour les interconnexions (Figure 8) : par exemple bus et réseaux-sur-puce. Ces interconnexions sont décrits comme des modules distincts, et interconnectés à côté des modules représentant l'application. L'interface sous-système d'interconnexion est

[14] Network Interfaces

constituée par ses points d'accès au réseau (NA[15]) qui fournissent des services à l'application.

Les sous-systèmes d'interconnexion considérés sont décrits en utilisant la méthodologie OCCN. La méthodologie OCCN propose un cadre efficace et flexible spécialement conçu pour la description et la simulation des sous-systèmes d'interconnexion embarqués. OCCN définit (a) une bibliothèque orientée-objet en C++, construite au-dessus du SystemC et (b) une l'interface de programmation (API) générique [Cop 03] [Copp 03].

(a) La bibliothèque OCCN construite au-dessus du SystemC utilise les concepts de base comme les ports et les canaux, comme suit :

- La couche de communication du sous-système d'interconnexion modélisé avec OCCN est dérivée de la classe de base « **`sc_channel`** » de SystemC. C'est cette couche de communication qui établit le transfert des messages, selon le protocole spécifique de chaque réseau.
- L'interface de communication est implémentée comme une spécialisation de l'objet de SystemC « **`sc_port`** ». En ce cas, un port SystemC est vu comme un point d'accès aux services (SAP[16]), et c'est l'API OCCN qui définit son service. Cette interface de communication abstrait la communication et la synchronisation inter-modules, fournissant les buffers nécessaires à la communication, et fournit des paradigmes de passage de message (ou même la mémoire partagée) pour être utilisée de tout sous-système d'interconnexion.

(b) L'API OCCN est basé sur le paradigme de transfert des messages. Ce paradigme améliore la portabilité et la réutilisation de tous les modèles décrits avec OCCN et utilisant ce type d'API. Ainsi, elle fournit l'ensemble de méthodes pour l'échange de données entre sous-systèmes, mais aussi pour leur synchronisation.

[15] Network Access (Point)
[16] Service Access Point

Dans notre réalisation, le sous-système d'interconnexion fournit cette API par l'intermédiaire de son interface d'accès au réseau (NA). L'API abstrait les différentes couches qui implémentent les pilotes de communication pour le sous-système d'interconnexion, les détails architecturaux liés à un protocole particulier de communication et à la topologie de l'interconnexion. Par exemple on ne décrit pas les détails de réalisation des composants, mais seulement les services fournis à une éventuelle application qui utilise cette interconnexion.

La modélisation en OCCN repose sur les **`PDUs`**[17] qui sont les unités de base pour la transmission. Ces PDUs, ailleurs nommés paquets ou messages, sont échangées au lieu des signaux individuels. En OCCN, les PDUs échangées sont constitués de deux parties :

- **un en-tête contenant l'information de contrôle (PCI**[18]). Cet en-tête a une signification surtout pour le sous-système d'interconnexion. Selon les informations de contrôle, obtenus suite au décodage du PCI, l'interconnexion prend en charge la transmission des paquets, conformément à son propre protocole.
- **le payload, nommé SDU**[19], contient les données effectives, sans une signification particulière pour le réseau. Ce corps du message sera récupéré à la réception et décodé par le module récepteur.

Un exemple de PDU est défini ensuite :

```
Pdu<HeaderType, BodyUnitType, length> pk ;
```

Le PDU « **`pk`** » contient un en-tête de type « **`HeaderType`** » et un corps de message de longueur « **`length`** », constitué des éléments de type « **`BodyUnitType`** ».

Les PDUs sont transférés en utilisant une approche de transfert de messages, comme par exemple MPI[20], basé sur l'API standard **`send()/receive()`**. La composition

[17] Protocol Data Units
[18] Protocol Control Information
[19] Service Data Unit
[20] Message Passing Interface

des PDUs, leur information de contrôle et la longueur de message, diffère d'un réseau à l'autre et d'un protocole à l'autre.

L'API OCCN est :

```
void send(Pdu<…>* p, sc_time& time_out=-1, bool& sent);

void asend(Pdu<…>* p, sc_time& time_out=-1, bool& dispatched);

Pdu<…>* receive(sc_time& time_out=-1, bool& received);

void reply(uint delay=0) or void reply(sc_time delay);
```

La première fonction (**send()**) réalise l'envoi synchrone. Le module émetteur va délivrer le PDU « **p** » seulement si : le récepteur peut recevoir ce message, le canal de communication est disponible et la valeur du compteur « **time_out** » (défini par l'utilisateur) n'est pas arrivée à l'échéance. Dans le cas où l'émetteur ne peut pas envoyer le message, il sera bloqué dans un état d'attente active (où il essayera d'envoyer ce message). Le flag nommé « **sent** » indique si le message a été envoyé ou pas (la valeur « **FALSE** » indique que le compteur a expiré avant que le module a réussi d'envoyer son message).

La fonction **asend()** est la version asynchrone de la fonction **send()**. Elle est aussi bloquante, mais par rapport à la fonction **send()**, la fonction **asend()**perd les paquets dans le cas de l'échec à l'envoi.

La paire des fonctions **receive()** et **reply()** réalise la réception synchrone. La fonction **receive()** est bloquante, jusqu'au quand le récepteur reçoit un PDU ou que la variable « **time_out** » arrive à l'échéance. Dans le dernier cas, le flag « **received** » devient « **FALSE** » – signifiant qu'aucun PDU n'a été réceptionné. La fonction **reply()** réalise la synchronisation entre le module émetteur et le récepteur. Cela est réalisé après le temps de la transaction du récepteur (stocké dans la variable « **delay** », ou un temps absolu de type « **sc_time** »).

Conclusions

ROSES est une méthode au niveau système pour la conception des MPSoC, développée au sein du groupe SLS. Son point de départ est la description de

l’architecture virtuelle du système. L’architecture finale est obtenue par des raffinements successifs de la communication et la génération des interfaces entre le logiciel et le matériel.

L’objectif de ce chapitre est de présenter le cadre de ce travail, et tous les notions de base impliqués. Premièrement, les étapes du flot de conception ROSES sont présentées. Le modèle d’architecture virtuelle est décrit, car il sera à la base de la plateforme proposée d’évaluation des performances. Un rôle majeur sera joué par la cosimulation logicielle/matérielle, car elle servira de modèle pour l’évaluation des performances par composition. Ce chapitre décrit aussi les sous-systèmes que l’on a trouvés « critiques » pour l’évaluation des performances : le sous-système logiciel et le sous-système d'interconnexion. Leur modèle de description et leur méthode de génération dans le flot ROSES sont brièvement introduits.

CHAPITRE II.

Etude systématique sur l'évaluation des performances des systèmes MPSoC

Introduction

L'objectif de ce chapitre est d'étudier les différents outils de l'évaluation des performances existants pour différents sous-systèmes. Cela va nous permettre l'analyse des points forts et points faibles pour les outils existants. A la fin, cette étude nous permettra de proposer une solution basée sur l'assemblage des outils individuels en vue d'une évaluation globale des systèmes MPSoC.

Les trois contributions majeures du travail présenté dans ce chapitre sont :

- (1) l'étude des éléments fondamentaux utilisés dans l'évaluation des performances à travers toutes les étapes d'un flot complet de conception MPSoC ;
- (2) la définition d'une taxonomie permettant l'étude structurée des sous-systèmes différents dans un environnement homogène ;
- (3) l'étude et l'analyse des différentes approches proposées actuellement pour l'évaluation des performances des différents sous-systèmes composantes du MPSoC.

La première section servira à établir une taxonomie qui nous facilitera l'étude de différents environnements d'évaluation de performances. Premièrement on détaille les métriques considérés, et les paramètres de conception d'un sous-système. Ensuite, cette taxonomie inclut à part le type du sous-système à analyser, ses niveaux d'abstraction, les méthodes que l'on peut utiliser pour évaluer ses performances :

statistiques, empiriques et analytiques. Ensuite, dans la deuxième section, tous ces éléments seront réunis dans un flot de conception. La section trois utilise la taxonomie définie précédemment pour présenter différents types de sous-systèmes. Elle est consacrée à l'étude explicite des sous-systèmes typiques considérés dans l'évaluation des performances : sous-systèmes matériels, CPU, logiciels, d'interconnexion et systèmes MPSoC. Après la définition, leurs niveaux d'abstraction, leurs métriques et paramètres de conception spécifiques seront détaillés. Dans la cinquième section nous analyserons l'état de l'art pour l'évaluation de performances des MPSoC.

1 Taxonomie pour l'étude des différents environnements d'évaluation des performances

Cette section a comme rôle la définition de tous les concepts de base, et de fournir une analyse pour les aspects impliqués dans l'évaluation des performances. Ces concepts sont réunis dans une taxonomie générique et constitueront les vecteurs d'orientations pour étudier des sous-systèmes variés (dans la suite de ce chapitre). Cette section commence par présenter les différentes métriques de performance. Ensuite, elle présente l'influence des paramètres de conception sur l'évaluation des performances. Après la définition des sous-systèmes à analyser et leurs niveaux d'abstraction, les méthodes les plus rencontrées dans l'évaluation des performances sont présentées.

1.1 Métriques

Par définition, **une métrique** est une fonction définie sur un certain domaine de définition, et qui prend des valeurs dans l'ensemble des nombres réels. Le domaine de définition est un ensemble abstrait. Dans notre cas, il contient l'ensemble de toutes les configurations dans l'espace architectural, plus l'ensemble de paramètres qui peuvent varier dans une architecture. Par exemple, en Chapitre III on étudiera l'influence de différents réseaux d'interconnexion sur les performances d'un MPSoC, en utilisant des métriques comme le débit, la latence, le temps d'exécution. Le domaine des valeurs, qui est l'ensemble des nombres réels est un ensemble totalement ordonné, qui permet des relations et comparaisons.

Les valeurs extrêmes qui délimitent l'intervalle de validité d'une métrique, sont nommés **contraintes de performances**. Des exemples sont la latence maximale acceptée dans un système, la consommation maximale d'énergie, etc. Dans le cas où

les contraintes de performance ne sont pas respectées, l'architecture doit être re-conçue.

Tant qu'une métrique ne dépasse pas ces valeurs extrêmes et elle se trouve à l'intérieur de l'intervalle accepté, elle constitue un **indicateur de performances**. Les indicateurs de performances servent à optimiser l'architecture conçue en rapport avec un ou plusieurs de ces indicateurs (le dernier cas correspond à l'optimisation multicritère).

Cette thèse considère principalement des indicateurs de performance temporels, qui sont : (a) ***le temps total d'exécution***, (b) ***la latence des primitives de transfert*** et (c) ***le débit*** de chaque module. On verra au cours des expérimentations que l'utilisation d'un seul indicateur n'est pas suffisante. En plus de ces indicateurs de base, il est aussi nécessaire de combiner leurs effets pour un module donné ou entre des modules différents. Par exemple, il est nécessaire d'inspecter le temps d'exécution d'une certaine primitive de communication ou portion de code, et il peut être aussi nécessaire d'inspecter les latences de transmission dans tout le système. Mais, pour évaluer l'architecture en totalité, il est impératif de connaître l'effet joint de ces deux indicateurs donnés en exemple.

1.1.1 Le temps d'exécution

Le temps d'exécution représente le temps nécessaire pour une application d'exécuter son code, éventuellement sur plusieurs processeurs parallèles. Cela inclut les temps nécessaires pour le transfert des données, pour l'exécution d'une primitive de communication et pour le transfert par le réseau. On mesure le temps d'exécution en collectant le temps du système avant et après l'exécution de la section à analyser.

Le temps d'exécution servira comme indicateur pour la comparaison entre plusieurs composants de l'architecture, comme par exemple le sous-système d'interconnexion, un CPU particulier, un composant matériel. Un temps plus court correspond à une réalisation plus efficace.

Néanmoins, le rapport entre le temps d'exécution du système en entier et temps dépensé pour l'exécution du composant sous analyse est un facteur important. Si ce

rapport est grand, il est difficile de déceler entre différentes réalisations pour le composant seulement en examinant cet indicateur.

1.1.2 Le débit

Le débit associé à un module est défini comme la moyenne du nombre de bytes transmis par l'unité de temps du module respectif et routés par le réseau d'interconnexion. La limite supérieure est le largueur de bande du réseau.

Un débit plus élevé reflète une communication plus efficace, et améliore ainsi des temps d'exécution. Aussi, dans la plupart des applications il n'est pas avantageux de dépasser le débit des ressources de calcul. Pour cela, pendant l'évaluation des performances on tiendra compte de ressources connectées aux réseaux d'interconnexion.

Un autre indicateur utile pour l'optimisation est **le ratio d'utilisation de l'interconnexion**. Il est évalué comme rapport moyen entre le débit moyen et la largeur de bande du réseau.

$$Utilisation = \frac{\langle Through\_NoC_y(M_x) \rangle}{\langle Through\_NoC_y \rangle} * 100\%$$

,

où $\mathbf{<Through\_NoC_y(M_x)>}$ est le débit moyen évalué pour le module $\mathbf{M_x}$, sur le réseau $\mathbf{NoC_y}$, alors que le $\mathbf{<Through\_NoC_y(M_x)>}$ est la largeur de bande du réseau $\mathbf{NoC_y}$.

1.1.3 La latence

Dans l'étude des réseaux**, la latence** est définie comme la quantité de temps nécessaire à un message pour traverser de la source à la destination.

Par l'analyse de la latence, il est possible de déduire ponctuellement si l'interconnexion répond aux contraintes imposées par l'application. Dans le cas où les latences ne sont pas incluses dans l'intervalle du temps réel, les conditions de temps réel imposées pour l'application ne sont pas satisfaites.

L'intervalle du temps réel (RT[1]) est par définition le temps passé entre deux événements successifs, où le système doit accomplir un certaine tâche. Par exemple, pour une application de codage d'image, on définit de manière générale, l'intervalle de temps réel comme le temps entre la réception d'une trame, et le temps de réponse du système avec l'image codée, ou plus simple comme l'intervalle de temps entre deux réceptions de trames consécutives. Ceci signifie que tous les calculs et transfert de données devraient être réalisés dans cet intervalle.

1.2 Les paramètres de conception

On a défini dans la section précédente une métrique comme une fonction. Les arguments de cette fonction sont **les paramètres de conception**. Ils peuvent être variés en vue de l'amélioration de l'architecture conçue, ou pour respecter les contraintes de performance. Ces paramètres de conception incluent des paramètres architecturaux (par exemple on choisit un partitionnement logiciel/matériel), des paramètres technologiques qui sont affectés à un processus donné (par exemple l'implémentation sur FPGA ou ASIC d'une fonction) ou des paramètres de dimensionnement (par exemple les paramètres intrinsèques d'un réseau-sur-puce : le nombre des nœuds, la topologie des routeurs, la largeur de bande etc.)

Selon le niveau d'abstraction, on s'appuie sur des paramètres de conception différents :

- A un haut niveau d'abstraction, les paramètres sont les caractéristiques architecturales abstraites comme par exemple le parallélisme d'une application ou la topologie d'un réseau.
- L'évaluation des performances à un bas niveau d'abstraction, est fondée sur le domaine de la technologie électrique ou des matériaux. Par exemple, ici, la tension d'alimentation ou la technologie de silicium sont des facteurs essentiels.

Une fois que l'on a établi les notions de « métrique » et « paramètres de conception », on peut commencer le processus d'évaluation de performances. Le processus

[1] Real Time

d'évaluation des performances implique l'existence de trois principaux éléments: le sous-système à analyser, son niveau d'abstraction et la méthode utilisée pour l'évaluation des performances. Le diagramme présenté dans la Figure 9 détaille sur trois axes ces trois éléments. Chacun de ces axes sera analysé individuellement dans les sections suivantes.

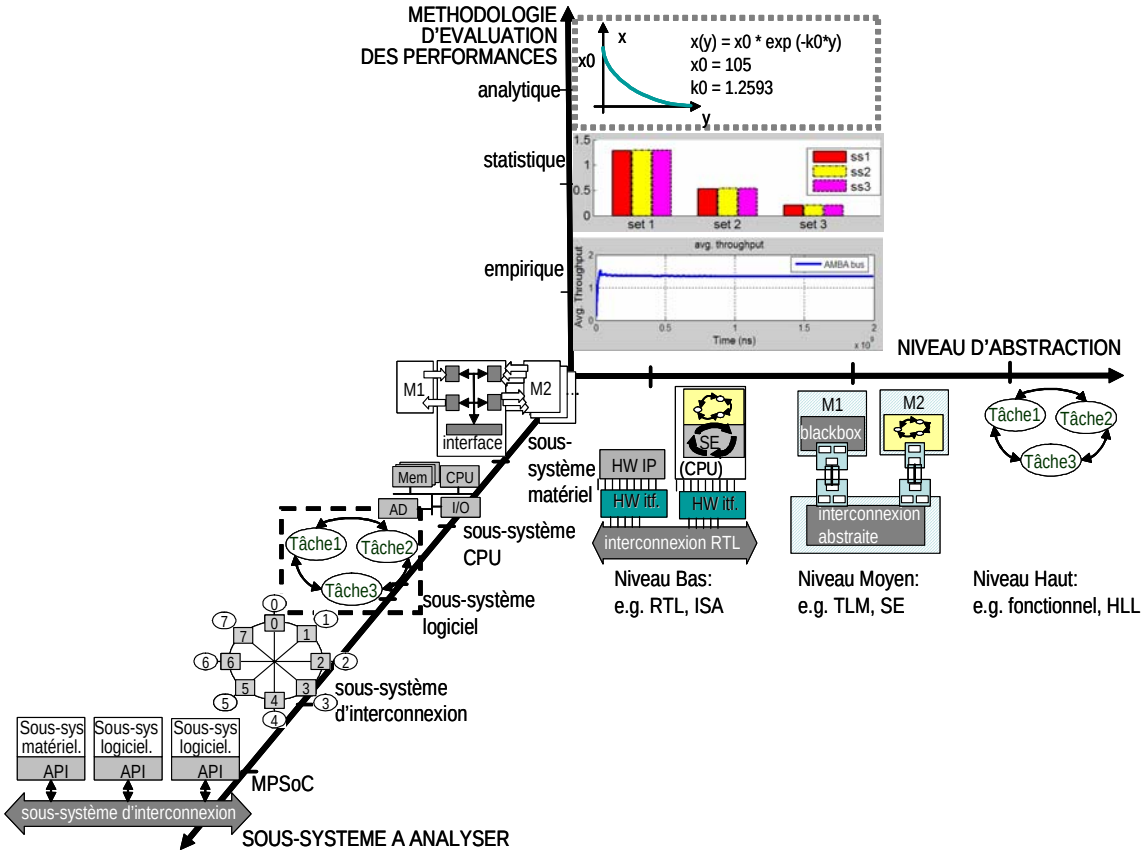


Figure 9. Les trois axes principales, caractéristiques pour les environnements d'évaluation des performances

1.3 Sous-systèmes à analyser

Le sous-système à analyser est défini par deux composantes : (1) son type, appartenant à une des classes sous-système logiciel, sous-système matériel, sous-système CPU, sous-système d'interconnexion ou MPSoC, et (2) son niveau d'abstraction, appartenant par exemple à une des classes niveau haut, intermédiaire et bas, selon le sous-système considéré. Les différents sous-systèmes et leurs caractéristiques seront détaillés dans la suite de ce chapitre.

Les concepteurs doivent évaluer des performances pour tous types de sous-systèmes, à partir des dispositifs simples et jusqu'à des modules sophistiqués. Sans perdre la généralité, ce travail considérera cinq types de sous-systèmes à analyser : matériel, logiciel, CPU et interconnexion, qui sont tous des parties constituantes du système multiprocesseur monopuce (MPSoC). Couramment, ces sous-systèmes sont étudiés par cinq communautés de recherche différentes :

- Classiquement, la communauté de conception des ***sous-systèmes matériels***, les définit dans des langages de description du matériel (HDL[2]), comme décrit en [Mic 01]. Ces sous-systèmes sont conçus avec des outils pour l'automatisation de la conception des systèmes électroniques – EDA[3]. Généralement ces outils incluent des évaluations de performances spécifiques, comme par exemple il est décrit en [Aga 00] [Bra 04] [Dey 97] [Yos 04].
- La communauté de l'architecture des ordinateurs étudie les sous-systèmes construits à base de l'unité centrale de traitement – des ***sous-systèmes CPUs*** [Pat 98]. Ils les considèrent comme des microarchitectures complexes ; par exemple en [Bis 06] le sous-système CPU contient à part l'unité centrale, des coprocesseurs matériels sont spécialement conçus pour augmenter les performances de l'application. En conséquence, des méthodologies spécialisées de conception et évaluation de leurs performances ont été développées, par exemple [Ofe 00].
- La communauté de conception des ***sous-systèmes logiciels*** considère ces sous-systèmes comme des programmes exécutant des tâches éventuellement parallèles [Sel 93] [Sel 98]. Elles sont conçues avec des outils assistés par l'ordinateur (CASE[4]) et évaluées avec des méthodes spécifiques, comme par exemple ceux référencés dans [Agr 90] [Bal 01] [Esk 02] [Kin 00] [Laj 99] [Liu 02] [Lu 00] [Mut 04] [Sch 04] [Spi 98] [Suz 96] [Wal 98] [Wey 02].

[2] Hardware Description Language
[3] Electronic Design Automation
[4] Computer-Aided Software Engineering

- La communauté de conception des ***sous-systèmes d'interconnexion***, considère ces sous-systèmes comme la manière de joindre et de faire communiquer divers composants logiciels ou matériels [Ben 02] [Bol 03] [Hin 97] [Lah 99] [Laj 98] [Pes 04] [Pol 03] [Row 97]. Les performances des sous-systèmes d'interconnexion déterminent l'efficacité du système global, et par conséquent ce domaine est intensivement exploré.
- La communauté ***MPSoC*** construit ces systèmes par l'assemblage de plusieurs sous-systèmes différents comme matériel, logiciel et interconnexion. Les approches spécifiques sont présentées dans Chapitre II.4.

1.4 Niveaux d'abstraction

La séparation en « niveaux d'abstraction » est un concept courant dans la théorie des systèmes. Chaque niveau représente une vue différente du même système, exprimé de manière différente, relatif à un domaine d'applicabilité particulier. Par définition, l'abstraction est le processus de réduction de la complexité d'un système, afin de maintenir l'information significative pour un cas de conception particulier. Par la suite, les niveaux d'abstraction simplifient tous les éléments qui ne sont pas relevants pour un niveau donné.

Le modèle de description des systèmes MPSoC peut se situer aux différents niveaux d'abstraction. Un grand nombre des travaux de recherche sont dédies à l'abstraction et à la standardisation des niveaux d'abstraction [Bou 06] [Gaj 03].

Chaque niveau d'une abstraction « élevée » est basé sur un niveau « plus bas », qui fournissent des représentations de plus en plus « granulaire ». Des exemples de dépendances de niveaux élevés basés sur des niveaux plus bas peuvent apparaître dans les circuits électroniques (par exemple, les portes logiques sont construits avec des dispositifs électroniques), aussi, pour les langages (par exemple, le langage de programmation est construit sur le langage machine, et les applications sont construits sur les primitives des langages de programmation).

Chacune des cinq communautés présentées dans la section précédente (la communauté de conception des sous-systèmes matériels, logiciels, CPUs, de

l'interconnexion et des systèmes MPSoC), utilise des différents niveaux d'abstraction pour représenter leurs sous-systèmes. Sans perte de généralité, dans cette thèse on considérera toujours trois niveaux d'abstraction : le niveau bas, le niveau haut et un niveau intermédiaire. Cette abstraction est valable pour les modèles de description des systèmes, et aussi pour les modèles de simulation. Les informations présentes à chaque niveau sont évidemment différentes selon le type du sous-système : logiciel, matériel, etc. ; des exemples pour chaque sous-système seront donnés dans le Chapitre II.3.

- **Le niveau bas** est le niveau le plus près de la réalisation finale, mais il est plus difficile à décrire et le plus lent à simuler. On connaît à ce niveau les schémas en portes ou en transistors des blocs matériels et les programmes en langage machine figé dans les mémoires ROM[5] des processeurs.
- **Le niveau haut** est utile pour la conception indépendante de la plateforme et pour minimiser les coûts de conception, par abstraction des détails. L'abstraction conduit à de temps de modélisation et simulation réduits et des retours en arrière admissibles. On ne connaît à ce niveau que le comportement d'un ensemble des fonctions (tâches) et les traitements qu'elles réalisent.
- Généralement, il peut y avoir plusieurs **niveaux intermédiaires**, selon les besoins d'exprimer le système, et les compromis de conception entre le temps et la précision. Un niveau intermédiaire introduit des abstractions supplémentaires par rapport au niveau le plus bas considéré.

Comme ils présentés en cette section, ces niveaux d'abstraction ne sont pas explicites. Mais ils seront particularisés pour chaque type de sous-système dans Chapitre II.3.

1.5 Méthodes d'évaluation des performances

La conception des systèmes embarqués dispose d'une gamme très riche de méthodes d'évaluation des performances. En étudiant ces méthodes, on observe que la

[5] Read Only Memory

littérature rapporte deux classes principales : les approches statistiques et déterministes. Chacune de ces dernières approches sera détaillée dans les sections suivantes.

1.5.1 Les approches statistiques

Pour des approches statistiques, la performance est une variable aléatoire caractérisée par plusieurs paramètres tels qu'une fonction de distribution de probabilité, une moyenne, un écart type et d'autres propriétés statistiques [Esk 02] [Wey 02].

Les approches statistiques procèdent en deux phases :

- La première phase fixe le modèle qui exprime le mieux l'exécution du système. Habituellement ses paramètres sont calibrés en exécutant des benchmarks aléatoires.
- La deuxième phase se sert du modèle statistique précédemment trouvé pour prévoir l'exécution de nouvelles applications. Dans la plupart des cas, cette deuxième phase fournit un feed-back pour mettre à jour le modèle initial.

1.5.2 Les approches déterministes

Les approches déterministes sont divisées en approches empiriques (par exemple [Aga 00] [Che 01] [Hin 97] [Laj 98] [Laj 99] [Liu 02] [Pes 04] [Pol 03] [Row 97]) et analytiques (par exemple [Dey 97] [Her 00] [Kin 00] [Lu 00] [Sch 04] [Spi 98] [Wal 98]). Dans ce cas-ci, la fonction de coût de la performance est définie comme variable déterministe, fonction des paramètres critiques.

Les approches empiriques

Les approches empiriques sont réalisées soit par la mesure du système, soit par simulation.

- La mesure est habituellement effectuée sur un prototype du système, ou sur le système déjà implémenté. Elle fournit des résultats extrêmement précis. Puisque cette approche peut être appliquée seulement tard dans le cycle de conception, quand le

prototype est déjà disponible, nous ne l'incluons pas dans cette thèse.

- Les approches basées sur la simulation sont fondées sur l'exécution du système complet, en utilisant des scénarios d'entrée ou des benchmarks représentatifs. Le grand avantage de ces approches est la bonne précision qu'elles peuvent fournir. Leur inconvénient est la vitesse réduite d'évaluation. La précision et la vitesse sont des facteurs dépendants du niveau d'abstraction du système simulé.

Les approches analytiques

Les approches analytiques étudient formellement les performances du système. Le sous-système à analyser est généralement décrit à un haut niveau d'abstraction, à l'aide d'équations algébriques.

Les théories mathématiques appliquées à l'évaluation des performances rendent possible une analyse complète de l'exécution du système, même à une étape préliminaire de conception. D'ailleurs, de telles approches évaluent très rapidement les performances du système parce qu'elles remplacent les longues étapes de compilation et d'exécution du système, présentes dans les approches basées sur la simulation.

Néanmoins, la construction d'un modèle analytique peut être une étape très complexe. Par exemple, il est difficile de modéliser un comportement dynamique comme le contexte de programme, ou les temps d'attente dus aux conflits ou aux collisions sur les interconnexions. Aussi, il est difficile de saisir les étapes d'optimisation comme par exemple les optimisations du compilateur.

Approches analytiques vs. simulation

Puisque des simulations sont généralement effectuées pour fournir une évaluation précise de l'exécution, l'analyse est typiquement utilisée aux premières étapes de conception pour fournir des évaluations préliminaires. L'analyse fournit des estimations approximatives de l'exécution, en pouvant indiquer l'influence des différents facteurs sur la performance globale, sans demander un grand effort de conception. Ainsi l'analyse permet l'évaluation des familles entières d'architectures

avec des paramètres et configurations variés, en dérivant l'ensemble d'équations paramétriques qui prédit leur exécution.

Pourtant, l'analyse implique un certain nombre d'approximations qui peuvent affecter l'exactitude des résultats. En outre les méthodes analytiques sont limitées une fois appliquées pour des systèmes trop complexes.

Les avantages et les inconvénients présentés mènent à la conclusion que bien qu'il soit raisonnable de travailler avec les modèles simples aux premières parties de la conception, des modèles plus détaillés deviennent nécessaires pour évaluer exactement la performance des systèmes.

2 Flot générique de l'évaluation des performances

Cette section relie tous les éléments présentés, et appartenant à l'évaluation des performances, dans un flot de conception. Ainsi, on identifiera avec les étapes du flot : les sous-systèmes et leur différents modèles au long du flot de conception, les niveaux d'abstraction impliqués, la localisation de l'étape d'évaluation des performances et les métriques et paramètres de conception impliqués.

Un flot de conception générique démarre toujours par dissocier les spécifications initiales du **sous-système à analyser** (Figure 10). Elles sont séparées dans une partie fonctionnelle et une partie non-fonctionnelle, qui définissent le modèle de performance.

Le modèle de performance est la représentation du sous-système à analyser pendant l'étape d'évaluation des performances. Le modèle de performances est un modèle de simulation (un programme exécutable) ou un modèle analytique (un ensemble d'équations) étendu, dans lequel on a incorporé des annotations qui tiennent compte des métriques à mesurer et des paramètres de conceptions disponibles.

La partie fonctionnelle du modèle de performance contient la description du sous-système à analyser. Dans notre cas de conception, la partie fonctionnelle est l'application embarquée. Les modules logiciels ou matériels représentant l'application embarquée communiquent en utilisant des primitives génériques de haut niveau, basés sur le transfert de message, par exemple en utilisant MPI. Le réseau d'interconnexion

communique via des points d'accès au réseau (NA), en utilisant sa propre interface de communication.

La partie non-fonctionnelle du modèle de performance est constituée de l'ensemble de contraintes ou métriques choisis pour évaluer les performances des spécifications initiales. Comme une règle, les caractéristiques critiques du sous-système à analyser sont définies comme des paramètres dans le modèle de performance : comme des métriques à analyser ou des paramètres de conception donnés en entrée. Par exemple, des **métriques de performance** souvent employées sont le temps d'exécution, le délai de calcul, la latence de communication, le débit de bits transférés, la surface occupée par différents composants.

Les paramètres de conception, utilisés comme des paramètres dans la fonction de performance, peuvent être implantés dans l'outil d'évaluation, ou donnés comme des bibliothèques externes. En ROSES, on fournit les configurations à partir des bibliothèques de composants et des annotations avec des paramètres.

En cette thèse, l'évaluation des performances est orientée vers l'évaluation rapide et précise des temps dans le système tel qu'il sera réalisé.

- En Chapitre III, les aspects temporels concernent le temps d'exécution de l'application et du système d'exploitation afférent, en tenant compte de l'exécution conjointe du matériel, et des interruptions matérielles. Les temps réduits d'évaluation sont atteints par la modélisation native du logiciel et la précision introduite par l'annotation automatique avec les temps réels d'exécution et la prise en compte du matériel.
- En Chapitre IV, les aspects temporels concernent le comportement en temps réel de l'application, par l'évaluation du temps d'exécution, des latences de communication et du débit par le réseau d'interconnexion. Le temps réduit d'évaluation est dû au modèle de simulation de haut niveau, tandis que la précision est due aux annotations présentes dans le code et aux modèles de description réelle des composants. Aussi, la génération

automatique des interfaces d'adaptation fournit un temps réduit d'exploration des solutions architecturales.

Les parties fonctionnelle et non-fonctionnelle de la description du système à analyser constituent l'entrée de **l'outil d'évaluation des performances**. Cet outil acquiert les performances du système en s'appuyant sur une méthodologie particulière d'évaluation.

La méthodologie d'évaluation de performances est l'ensemble de stratégies pour la mesure du sous-système sous analyse et aussi la stratégie d'analyse des résultats de ces mesures. Sa précision est dépendante du niveau d'abstraction du sous-système à analyser. Les différents types de méthodes ont été présentés en Chapitre II.1.5. Dans notre cas on utilisera la cosimulation.

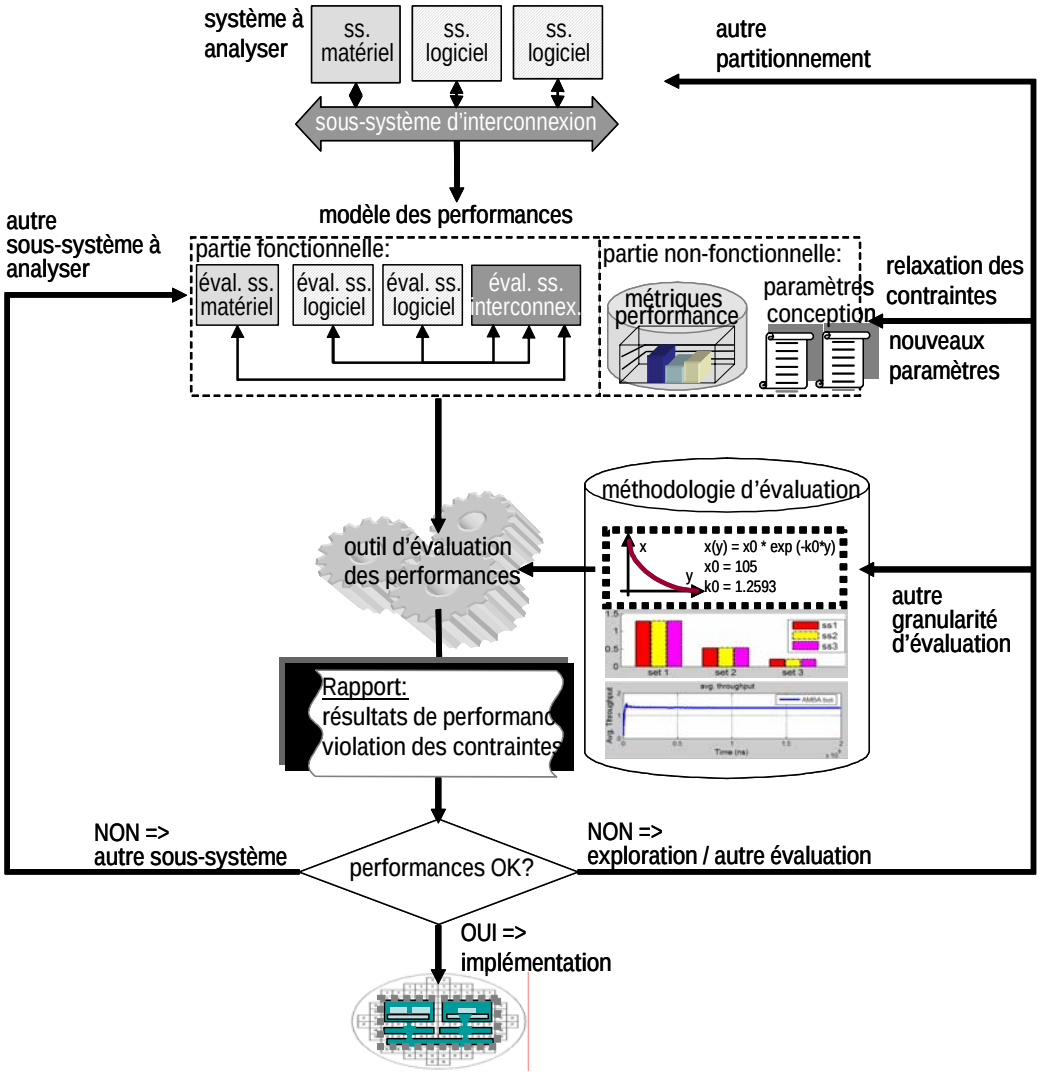


Figure 10. Flot générique de l'évaluation des performances et optimisation des MPSoC

Le processus de conception s'étend souvent sur plusieurs itérations où le système doit être raffiné ou même partiellement reconçu. Les diverses récurrences de conception peuvent modifier le modèle de performance, les paramètres du sous-système à analyser, les contraintes de conception, etc. Chaque itération suppose un nouveau calcul pour la réévaluation des métriques de performance. Parmi les nombreuses possibilités qui peuvent intervenir on peut énumérer : l'augmentation de la surface de la puce, afin que l'application satisfasse les contraintes de temps réel ; le changement de la technologie de réalisation ; ou la modification de la fréquence de calcul d'un ou plusieurs sous-systèmes.

3 Modèles des sous-systèmes composants des MPSoC

Dans cette section, on va considérer cinq types principaux de sous-systèmes étudiés actuellement dans le monde de la conception: sous-systèmes matériels, sous-systèmes CPUs, sous-systèmes logiciels, sous-systèmes d'interconnexion, et les systèmes multiprocesseur monopuces (MPSoC).

Ce qu'on dénote par « modèle d'un sous-système » réunit son modèle de description, les niveaux d'abstractions auxquels peut se trouver ce modèle de description, aussi les métriques et paramètres de conception qui translatent le modèle de description en modèle de performance.

Le cadre de cette étude a été établi en s'appuyant sur les catégories définies dans Chapitre II.1 : les métriques d'évaluation des performances, les paramètres de conception, les sous-systèmes à analyser, les niveaux d'abstraction et les méthodes d'évaluation des performances. Pour toutes les classes de sous-systèmes, on particularisera ces éléments, définis précédemment. Cet ensemble de éléments permet de décrire et analyser chaque sous-système d'une manière homogène.

La structure générale de chaque sous-section qui suit, suit le schéma défini par ces catégories : c.-à-d. pour chaque sous-système on présentera ses métriques, paramètres de conception, niveaux d'abstraction et méthodes d'évaluation des performances.

3.1 Modèles des sous-systèmes matériels

Un sous-système matériel est composé d'un ensemble des unités fonctionnelles, (partiellement) statiques et faiblement programmables, comme par exemple les

FPGAs[6] ou les ASICs. En cette thèse, le concept de sous-système matériel exclut tous les modules qui sont des sous-systèmes CPUs ou d'interconnexion. Les CPUs exécuteront des programmes ; la transmission de données des sous-systèmes d'interconnexion se fait en deux couches : logicielle et matérielle. En plus, nous consacrons l'étude aux sous-systèmes matériels numériques.

La spécification du sous-système matériel peut être décrite à l'aide des machines à états finis (FSMs[7]), fonctions logiques, ou ensembles conjugués de parties contrôle et opérative.

Le plus bas niveau d'abstraction considéré dans le cadre de cette thèse est ***le niveau RTL***. Les données sont décrites sous la forme logique de vecteurs de bits. Le simple branchement des sorties d'un bloc aux entrées d'un autre assure la transmission de l'information. Le temps est celui de la réalisation finale. Au niveau RTL, le calcul et la communication du système sont détaillés au niveau de chaque cycle d'horloge.

Le plus haut niveau d'abstraction est ***le niveau fonctionnel***. A ce niveau de description, on décrit abstraitement le média de communication. Différentes entités (tâches, modules, etc.) échangent de l'information ainsi : un émetteur envoie (par un acte d'émission) une valeur ; de l'autre côté de cette chaîne virtuelle de communication y a un récepteur qui reçoit la valeur (par un acte de réception). L'émetteur, le récepteur, les actes d'émission/réception, le type de la valeur émise/réceptionnée sont nommés « virtuelles » (ou « abstraites »), car leur réalisation n'est pas encore définie à ce niveau. Le média virtuel n'a pas généralement de relation avec la topologie du réseau de communication final qui va servir pour l'implémentation réelle. Le temps n'est pas relevant au niveau fonctionnel. Le traitement réalisé est décrit par des combinaisons d'opérations arithmétiques et de mémorisation.

Les abstractions du niveau intermédiaire, peuvent être relatives aux données et/ou signaux de contrôle et horloge. Une donnée abstraite est par exemple un entier naturel, perdant des informations concernant les signaux réels, qui pouvaient être représentés dans le cadre des vecteurs de bits avec des états de haute impédance et

[6] Field-Programmable Gate Arrays
[7] Finite State Machines

indéfini (par exemple avec les symboles 'Z' et 'X'). L'abstraction relative aux protocoles et à l'axe du temps, peut ignorer toutes les étapes intermédiaires qui aboutissent à la transmission de données et ne garder que la donnée effective transmise ainsi que les instants de début et fin de la transaction (niveau TLM[8]).

Table 1. Niveaux d'abstraction du sous-système matériel

		FOURNIT :	
		DATA	**CONTROLE**
NIVEAU D'ABSTRACTION	**HAUT : FONCTIONNEL**	STRUCTURES ABSTRAITES (MESSAGES, STRUCT, ETC.)	MEDIA VIRTUEL DE COMMUNICATION, SANS NOTION DE TEMPS
	INTERMEDIAIRE : TRANSACTIONNEL	ENTIERS, PAQUETS, ETC.	ABSTRACTION DU CONTROLE, ORDRE DES EVENEMENTS, DUREE DES TRANSACTIONS
	BAS : RTL	VECTEURS DE BITS, SIGNAUX REELS	TOUS LES SIGNAUX DE CONTROLE, HORLOGE

Les métriques typiques dans l'évaluation des performances pour les sous-systèmes matériels sont la puissance consommée, le nombre de niveaux de portes (portant sur le temps d'exécution) ou le nombre des portes, des blocs FPGA, des blocs mémoire (portant sur la surface occupée). Ils sont généralement extraits avec précision pendant des évaluations de bas niveau et ensuite employés dans des modèles plus abstraits. Ce procédé est nommé « back-annotation » et il est utilisé dans tous les outils de synthèse.

Les paramètres de conception permettent de dissocier les détails d'exécution de la vraie réalisation matérielle, selon le niveau d'abstraction considéré, comme suit :

- ***A un haut niveau d'abstraction***, alors que les signaux et le comportement sont abstraits, les paramètres de conception dénotent le partitionnement de haut niveau des processus, avec la granularité des fonctions (par exemple fonctions C). Les évaluations des performances concernent la quantité de transactions échangées entre ces processus.

[8] Transaction Level Model

- ***Au niveau bas***, le modèle du sous-système matériel est complet. Il exige des paramètres désignant les propriétés structurales et temporelles (par exemple la taille de la mémoire, type de mémoire utilisée, etc.) et les détails de réalisation comme par exemple le mapping sur un FPGA ou ASIC.
- ***Au niveau intermédiaire***, les paramètres de conception sont liés aux formats de données, comme par exemple taille, codage etc., ou au traitement de données, comme par exemple nombre des bytes transférés, utilisation des ressources et latence.

Plusieurs **outils pour l'évaluation des performances** des sous-systèmes matériels existent. Ils s'appuient sur des différentes approches d'évaluation des performances : simulation [Aga 00], approches analytiques [Dey 97], approches mixtes analytiques et statistiques [Agr 90], approches mixtes basées sur simulation et statistiques [Bra 04], approches mixtes basées sur simulation et analyse [Bju 02], approches mixtes basées sur simulation, analyse et statistiques [Yos 04].

Par exemple, l'approche présentée en [Bra 04], est utile pour l'estimation paramétrique de la surface occupée par le système conçu, sur FPGA. **La métrique** examinée est donc la surface occupée, exprimée en nombre de Look-Up-Tables (LUTs) et de Flip-Flops (FFs).

Les paramètres de conception sont : le nombre des pas de contrôle et le nombre des entrées/sorties de contrôle pour les FSMs ; le nombre et la taille des registres de données (de sortie et internes) ; le nombre, le type et la taille des opérateurs ; le nombre et la taille des multiplexeurs. Ces paramètres ont été établis comme des paramètres relevants pour la métrique de performances, à partir d'un ensemble de benchmarks.

L'abstraction du système à analyser est sa description de haut niveau, en SystemC, qui sera transformée dans la description RTL-VHDL, directement synthétisable vers une netlist FPGA. L'architecture cible considérée est la famille FPGA Xilinx VirtexII-Pro.

La méthode d'évaluation est analytique ; elle est basée sur une fonction d'estimation qui est algorithmique et/ou algébrique pour les hauts niveaux d'abstraction, et purement algébrique pour les bas niveaux d'abstraction. La méthode nécessite de trouver cette fonction et d'établir ses paramètres. Ensuite, elle peut être utilisée pour estimer la surface des modèles de conception, à partir de haut niveau.

Dans la méthode d'évaluation des performances, les paramètres sont obtenus par l'inspection du code source, mais aussi des ports et des signaux de tous les modules de la description de haut niveau du système (en SystemC). Des méthodes statistiques et de pire cas interviennent aussi, pour l'estimation des paramètres difficile à mesurer. Ensuite, le modèle de bas niveau (niveau de transfert de registres) combine tous les paramètres de haut niveau pour obtenir le nombre des

LUTs et FFs nécessaires pour la réalisation finale du système.

En effet, cette approche présente une méthode d'exploration rapide et précise pour les différentes implémentations algorithmiques d'une application. Les auteurs proposent (en cours de développement) un outil académique pour l'évaluation de la surface, mais aussi incluant aussi des autres métriques comme le temps et l'énergie consommée.

3.2 Modèles des sous-systèmes CPU

Le sous-système CPU est un sous-système matériel exécutant un ensemble d'instructions spécifiques. Il est défini par l'architecture de l'ensemble des instructions (ISA[9]). Aussi, la réalisation et l'interconnexion des diverses unités fonctionnelles, l'utilisation de registres et l'adressage de la mémoire sont détaillées.

Le plus bas niveau d'abstraction est ***le niveau microarchitectural*** (ou RTL). Il offre la vue la plus détaillée du CPU. Il fournit la description détaillée complète de chaque module, tenant compte de la hiérarchie interne pour les structures de données, contrôle et mémoire. Chaque instruction est détaillée au niveau de chaque cycle ; le pipeline éventuel et les niveaux de cache sont indiqués de manière explicite. Les actions « atomiques » sont les transferts de registres (qui peuvent être parallèles).

Le niveau le plus élevé, pour décrire le CPU est ***le niveau ISA***. A ce niveau, les programmes s'exécutent sur une représentation virtuelle de l'unité centrale de traitement, avec des interconnexions symboliques et des paramètres abstraits ; un exemple éloquent est un simulateur d'ensemble d'instruction (ISS). Le modèle du processeur peut alors être un programme (par exemple décrit en C), manipulant un ensemble de variables entières représentant les registres (le compteur de programme, le registre d'instructions, etc.), un tableau d'entiers représentant la mémoire, etc. Ce modèle sera détaillé en Chapitre III.1.1.1.

Un niveau intermédiaire est ***le niveau ISA précis au niveau cycle***. Il exploite le vrai modèle de l'ensemble d'instructions et des ressources internes (registres) comme le niveau microarchitectural. Pourtant il utilise une représentation abstraite de l'unité centrale de traitement, comme le niveau ISA. La différence est que l'exécution des

[9] Instruction Set Architecture

instructions est détaillée sur ce CPU abstrait avec la précision du signal d'horloge. Les actions « atomiques » sont les instructions non-parallèles (sauf si le processeur est super-scalaire).

Table 2. Niveaux d'abstraction du sous-système CPU

		ABSTRACTION :	
		ARCHITECTURE	INSTRUCTIONS
NIVEAU D'ABSTRACTION	HAUT : ISA	CPU VIRTUEL	SEMANTIQUE DU LANGAGE MACHINE
	INTERMEDIAIRE : ISA – PRECISION DES CYCLES D'HORLOGE	–	INSTRUCTIONS PRECISES AU NIVEAU CYCLE
	BAS : RTL	DESCRIPTION DETAILLEE COMPLETE DU CPU : PARTIE OPERATIVE, PARTIE CONTROLE ; HIERARCHIE, MEMOIRE, PIPELINE	INSTRUCTIONS PRECISES AU NIVEAU CYCLE, INCLUANT LES EFFETS DE PIPELINE ET CACHES

Les principales métriques de l'évaluation des performances pour les sous-systèmes CPU sont liées au comportement temporel [Hen 03]. Les autres classes de métriques d'évaluation des performances sont la consommation d'énergie et la capacité de la mémoire utilisée par l'application.

Dans cette thèse on étudie spécialement le comportement temporel, qui inclut entre autres :

- ***le débit*** qui exprime le nombre d'instructions exécutées par le CPU par l'unité de temps ;
- ***l'utilisation*** qui représente le rapport entre le temps dépensé sur l'exécution des tâches, par rapport au temps total d'exécution (incluant aussi les communications, la synchronisation, et les opérations du système d'exploitation) ;
- ***le temps d'exécution***, qui est le temps consacré à l'exécution d'un programme sur un CPU particulier.

Les paramètres de conception abstraient les détails de réalisation de l'unité centrale de traitement, comme suit :

- ***Au niveau le plus bas***, microarchitectural, la technologie de réalisation du CPU est abstraite, et pour certains cas aussi les techniques particulières de réalisation des mécanismes (par exemple mécanismes de prédiction de branchement, etc.).
- ***Le niveau le plus haut***, ISA, abstrait la microarchitecture, comme par exemple le pipeline ou la mémoire cache, en fournissant seulement l'ensemble d'instructions pour la programmer.
- ***Les niveaux intermédiaires*** abstraient la réalisation de la partie opérative et de la partie contrôle, mais l'exécution est toujours détaillée avec la précision d'un cycle d'horloge. Cela est réalisé en utilisant l'ensemble réel d'instructions (`load/store`, instructions de calcul en virgule flottante ou instructions pour la gestion de la mémoire), l'ensemble effectif des registres et les ressources internes.

Plusieurs **outils de l'évaluation des performances** pour des sous-systèmes CPU existent. Ils sont basées sur la simulation [Che 01] [Tiw 94], sur des approches analytiques [Her 00] [Wal 98], approches statistiques [Esk 02], approches mixtes analytiques et statistiques [McK 99], approches mixtes basées sur simulation, analyse et statistiques [Ofe 00], ou des mesures sur les plateformes matérielles réelles [Bis 06].

Par exemple, l'outil nommé GROMIT, présenté en [Her 00], est un outil pour l'évaluation des temps d'exécution d'un programme s'exécuter sur un CPU. **La métrique** examinée est le temps d'exécution (pour le pire cas - WCET[10]), déterminée pour la voie d'exécution (dans le pire cas). **Les paramètres de conception** sont le programme à analyser (en assembleur), la description de la voie de données/contrôle du programme et la description des propriétés du processeur (modèle simplifié de ce processeur, en incluant la mémoire cache et le pipeline).

L'abstraction du sous-système à analyser inclut : l'abstraction du programme, l'abstraction de l'architecture du processeur et la composition de ces deux éléments. ***L'abstraction du programme***

[10] Worst-Case Execution Time

est décrite à l'aide d'une représentation interne sous la forme d'un graphe de flot de contrôle hiérarchique. Les restrictions imposés sont : le programme ne doit pas contenir des boucles infinies, des fonctions récursives ou des références dynamiques. Dans notre approche, ces restrictions n'apparaissent pas ; la différence est que nous évaluons les performances de manière dynamique (au temps d'exécution du programme), en temps que en [Her 00] l'évaluation est statique (analytique).

Les blocs de base de ce graphe de flot de contrôle hiérarchique seront analysés pour dériver le temps d'exécution. Les temps d'exécution de chaque bloc de base du programme seront calculés en conformité avec le modèle de processeur.

Le modèle du processeur inclut les effets de l'architecture cache et du pipe-line. Son abstraction (du point de vue structural et fonctionnel) est dérivée en conformité avec des meta-modèles prévues pour ce but. La modélisation du processeur focalise sur différents types de mémoire (mémoire interne/externe, de données/instructions, mémoire asynchrone/synchrone, de type DRAM, SRAM ou ROM) et le comportement du bus. Aussi, toutes les étapes du pipe-line sont modélisées. L'évaluation est réalisée sur deux processeurs différents (deux PowerPC : PPC403 et MPC750).

La méthode d'évaluation est basée sur la modélisation du processeur et l'analyse des temps d'exécution. La fonction de coût pour le temps d'exécution est une description ILP[11], incluant aussi les contraintes structurales et fonctionnelles pour les éléments de l'architecture et les éléments de programme, respectivement. Ensuite un outil de calcul (nommé « lp solve ») solutionnera la description ILP.

Cette approche a été incluse dans la section « Modèles des sous-systèmes CPU », car elle est orientée vers la comparaison de deux CPUs, et pas vers l'évaluation d'une application logicielle avec plusieurs réalisations (comme il sera le cas pour les approches incluses dans la section « Modèles des sous-systèmes logiciels »). On remarque que ces deux sous-sections peuvent avoir des parties recouvrantes, à cause de l'interdépendance de l'application logicielle et le CPU sur lequel elle s'exécute.

3.3 Modèles des sous-systèmes logiciels

Le logiciel embarqué est défini par l'ensemble de programmes à exécuter sur un ou plusieurs CPUs. On peut distinguer des sous-systèmes logiciels avec :

- différentes représentations : procédurale, fonctionnelle, orientée objet, etc. ;

[11] Integer Linear Programming description

- différents modèles d'exécution : avec un seul fil d'exécution[12] ou des fils multiples d'exécution[13] ;
- différents degrés de réponse temporelle : en temps réel ou sans contraintes de temps réel ;
- différents langages de programmation : à partir du langage de haut niveau vers l'ensemble d'instructions assembleur ;
- différentes utilisations (niveaux d'abstraction) : application, couche « haute » du système d'exploitation, couche « basse » du système d'exploitation (HAL). Ces différentes couches sont les différents niveaux qui interviennent dans la conception du logiciel. Néanmoins, pendant l'exécution toutes les instructions sont unifiées (des différences peuvent apparaître quand elles sont utilisées en mode « utilisateur » et « superviseur »).

Le plus bas niveau est le niveau ISA (Instruction Set Architecture). Ce niveau abstrait la microarchitecture du processeur, en ne gardant que les parties fonctionnelles de celle-ci, comme les parties de calcul et de stockage : opérations élémentaires de l'unité d'exécution, interruptions et le banc des registres. L'interface de programmation de ce niveau est le langage assembleur du processeur utilisé. Au niveau ISA, le concepteur connaît avec une granularité fine les détails du matériel qui est au-dessous. Par suite, il peut décrire les réalisations spécifiques pour les procédures qui sont derrière les interfaces de programmation de la partie de niveau bas (HAL) ou de niveau haut du SE (Chapitre I.2.2).

Le niveau le plus abstrait, est celui fournit par les langages de haut niveau HLL[14], comme Java ou C/C++. Cette abstraction tend à uniformiser la machine en abstrayant tous les aspects liés à l'architecture comme les ressources de stockage ou de calcul ou le type de processeur. Certains langages peuvent décrire le parallélisme à un haut niveau d'abstraction. A ce niveau, l'application se compose de l'ensemble de tâches (parallèles) communiquant par les primitives fournis par les langages de

[12] single-threaded
[13] multi-threaded
[14] High Level Language

programmation, par exemple des appels des procédures, création des processus/threads, etc.

Un niveau intermédiaire est contenu par le système d'exploitation, assurant des mécanismes de gestion de ressources : mémoire virtuelle ou réelle, allocation du processeur au processus, entrées/sorties, etc. On va nommer ce niveau SE (ou HAL, selon les détails implémentés). Ce niveau contient la gestion du parallélisme des tâches de l'application, le mécanisme d'interruption, le changement de contexte, etc. L'application logicielle est constituée par des programmes parallèles, mappés et exécutés sur des CPU génériques. La communication est fondée sur des APIs spécifiques au SE. Les APIs du SE sont déterminés (par exemple les APIs POSIX), mais leur réalisation spécifique n'est pas encore déterminée.

Table 3. Niveaux d'abstraction du sous-système logiciel

		CARACTERISTIQUES		
		LANGAGE	FAIT ABSTRACTION DE :	FOURNIT :
NIVEAUX D'ABSTRACTION	HAUT : HLL	C/C++ ; UTILISE PAR : LANGAGES DE PROGRAMMATION	MACHINE D'EXECUTION	CODE APPLICATION DE HAUT NIVEAU
	INTERM. : SE/HAL	C/ASM ; UTILISE PAR L'APIS DU SE	CPU	GESTION INTERRUPTIONS, CHANGEMENT CONTEXTE, SCHEDULING
	BAS : ISA	ASM ; UTILISE PAR L'APIS DU HAL	MICRO-ARCHITECTURE	LECTURE/ECRITURE DES CONTROLEURS, DE LA MEMOIRE, EXECUTION DES INSTRUCTIONS

Les métriques les plus utilisées pour l'évaluation des performances sont comme pour tous les sous-systèmes, le temps d'exécution, la consommation d'énergie et la mémoire utilisée, particularisés pour le logiciel ; par exemple le temps d'exécution d'un programme et l'énergie dépensée dépendent de nombre de cycles exécutés par instruction, et du type d'instruction (avec la mémoire, de calcul, etc.), et la taille

nécessaire pour la mémoire de programme (ROM/RAM[15] du processeur). En plus, l'exécution du logiciel peut considérer entre autres, son degré de concurrence, en fonction de l'hétérogénéité et de l'abstraction à différents niveaux [Net 04]. Ces métriques sont calculées en variant les paramètres liés à l'application logicielle et à l'architecture du CPU.

Pour l'évaluation des performances des sous-systèmes logiciels, **les paramètres de conception** abstraient la plateforme d'exécution, et définissent l'application logicielle.

- ***A un haut niveau***, les paramètres de conception se rapportent la plupart du temps aux caractéristiques comportementales de l'application, abstrayant les détails de communication et réalisation. Par exemple, le parallélisme des tâches, leurs priorités et la politique d'ordonnancement sont abstraites ; aussi, le temps d'exécution est une notion abstraite : chaque instruction s'exécute instantanément ; l'utilisation mémoire aussi est abstraite – les programmes utilisent des variables, des pointeurs, sans considérer les mécanisme de mémoire partagée, mémoire virtuelle etc.
- ***Au niveau intermédiaire*** (SE/HAL), les paramètres de conception incluent des particularités du système d'exploitation comme par exemple interruptions, ordonnancement, changement de contexte, mais leur exécution demeure abstraite. Dans l'exécution des programmes, ils négligent la technique ou des ressources utilisées pour la synchronisation. Ils imposent d'employer les APIs spécifiques au SE (par exemple mécanismes basés sur des sémaphores, mutex, etc.)
- ***Au niveau bas***, les paramètres de conception abstraient seulement la technologie de réalisation, alors que tous les autres détails, comme le schéma de transfert de données, le mapping dans la mémoire ou le mode d'adressage sont explicitement mentionnés.

[15] Random Access Memory

Les différents **outils existants pour l'évaluation des performances** des sous-systèmes logiciels sont basées sur la simulation [Laj 99] [Suz 96], approches analytiques [Bal 01] [Spi 98], approches statistiques [Esk 02], et approches mixtes basées sur simulation, analyse et statistiques [Mut 04]. Des outils orientés vers l'évaluation des performances des modèles pour les applications logicielles, utilisent aussi des différentes approches: basées sur simulation [Liu 02], approches analytiques [Kin 00], approches statistiques [Wey 02].

Par exemple [Laj 99] présente un outil pour l'évaluation des temps d'exécution pour le logiciel embarqué. L'approche est incorporée dans l'environnement de conception, POLIS. POLIS utilise deux schémas différentes d'évaluation pour les temps d'exécution du logiciel : une de haut niveau basée sur la macro-modélisation des délais après l'analyse du code, et l'autre de plus bas niveau, basée sur les estimations de l'ISS. L'approche proposée en [Laj 99] se propose d'unifier ces deux techniques par le développement d'une approche intermédiaire qui prend les avantages des deux techniques.

La métrique examinée est le temps d'exécution pour le logiciel embarqué. Le temps est accumulé à partir des annotations introduites dans le code, au temps de la simulation (native[16]) du système, et évalué en nombre de cycles d'horloges. **Les paramètres de conception** sont le programme décrit dans un langage de haut niveau (en C) et le processeur cible. La différence par rapport aux travaux sur le sous-système CPU (présentés à la fin de la section précédente, en Chapitre II.3.2) est que pour les sous-systèmes CPU les performances sont mesurées en fonction de la variation des paramètres du processeur, ou même en fonction du processeur lui-même. Pour les sous-systèmes logiciels, les paramètres d'intérêt sont les temps d'exécution des différentes implémentations d'un programme, sur un CPU donné.

L'abstraction du sous-système à analyser fournie par l'environnement POLIS. Ainsi, le système est décrit à l'aide d'un modèle comportemental formel nommé CFSM[17]. Le code est compilé et pour l'estimation des performances il est utilisé l'ensemble des instructions assembleur sous la forme des expressions RTL (Register Transfer List). Ce format intermédiaire est habituel pour la majorité des algorithmes de compilation.

Le modèle de performance est composé de la description comportementale avec des annotations de temps spécifiques à un simulateur de processeur (ISS).

La méthode d'évaluation est une approche basée sur la simulation. La plateforme de simulation est flexible, rapide et précise, considérant l'effet de la compilation et les particularités du processeur cible.

L'approche ne présente pas une différence majeure par rapport à notre méthode d'annotation. La différence consiste en la prise en compte de ces annotations temporelles. Dans notre cas, le sous-système logiciel communique avec (et peut être interrompu par) le sous-système matériel. Nous

[16] simulation effectuée directement sur la machine hôte
[17] Codesign Finite State Machines

avons pris en compte ces interactions et on a développé l'interface de cosimulation avec le matériel. De plus, nous avons aussi considéré le système d'exploitation, qui peut contenir des parties du code spécifiques au processeur sur lequel il est exécuté ; et on a re-modélisé cette partie du logiciel.

3.4 Modèles des sous-systèmes d'interconnexion

On a défini (en Chapitre I.3) le sous-système d'interconnexion embarqué comme tout fil de connexion ou circuit avec le rôle de transmission de l'information. Le sous-système d'interconnexion fournit les médias et les protocoles nécessaires pour la communication. Le média d'interconnexion est la couche « matérielle » qui peut être définie à un niveau bas, comme le matériel avec le rôle d'interconnexion, ou à un niveau élevé, comme une topologie abstraite. Alors que le protocole est la couche « logicielle » qui indique comment employer des ressources du réseau pendant l'exécution du système. Ce protocole est parfois mis en oeuvre à l'aide d'automates matériels.

Le plus haut niveau d'abstraction est le ***niveau fonctionnel***. Au niveau fonctionnel, les différents modules communiquent en exigeant des services qui utilisent des protocoles abstraits, par l'intermédiaire des topologies abstraites de réseau. Ce niveau fournit à l'application qui utilisera le sous-système d'interconnexion, une interface de programmation (API) qui permet une communication basée sur l'échange des paquets. La plupart des communications impliquent le transfert de données, par l'envoi d'un simple paquet ou d'une séquence des paquets. Des fonctions typiques de l'API sont par exemple les appels **`read()/write()`** sur différents ports d'entrée/sortie, et ayant comme argument le message à transmettre.

Cette API soustrait tout le détail des signaux et de l'architecture. Il y a des modèles de haut niveau qui permettent une représentation des propriétés fondamentales, telles que la technique de commutation, la stratégie de routage, les schémas d'empilement ou contrôle de flot de données. A ce niveau d'abstraction, un bus est une topologie abstraite, vu comme noeud reliant plusieurs modules, maîtres et esclaves. Les modules communiquent par des primitives de haut niveau, ignorant les signaux du bus ou les protocoles détaillés d'envoi/acquittement.

Au plus bas niveau d'abstraction, ***le niveau RTL***, la communication est réalisée par des interconnexions explicites comme les fils physiques ou des bus, transportant des données décrites explicitement. A ce niveau, chaque paquet est segmenté en cellules de base, chaque cellule ayant la largeur appropriée pour pouvoir être routée sur l'architecture de l'interconnexion embarquée. La réalisation physique de ces cellules, ajoutant des paramètres supplémentaires de contrôle, comme des délimiteurs de paquets. Les circuits contrôleurs de réseau (comme les routeurs) sont décrits au niveau cycle près.

Le niveau intermédiaire est ***le niveau transactionnel*** (TLM). Le niveau TLM emploie toujours des protocoles de transmission abstraits, mais il nécessite une topologie fixe de communication. Généralement, cela sera la topologie effectivement réalisée. L'interface de programmation est générique, permettant la réutilisation des spécifications, pour des architectures variés de l'interconnexion: par exemple connexion point-à-point, bus, réseau embarqué. Les opérations de base sont envoi/réception. L'unité de communication de base est le paquet, variant selon la complexité du protocole, la quantité de données à transférer, et la largeur de bande disponible. Comme la topologie d'interconnexion est déjà fixée, chaque paquet porte un quanta fixe de données, directement dérivé de l'architecture d'interconnexion.

Table 4. Niveaux d'abstraction du sous-système d'interconnexion

		ABSTRACTION :	
		PROTOCOLE	MEDIA INTERCONNEXION
NIVEAU D'ABSTRACTION	HAUT : FONCTIONNEL	PROTOCOLE ABSTRAIT : ECHANGE DE DONNEES	MEDIA ABSTRAITE : NŒUDS D'INTERCONNEXION, ABSENCE DE SIGNAUX DE CONTROLE
	INTERMEDIAIRE : TRANSACTIONNEL	PROTOCOLE ABSTRAIT : PAQUETS PARAMETRABLES	MEDIA EXPLICITE : POINT-A-POINT, BUS, NOC
	BAS : RTL	PROTOCOLE EXPLICITE : CELLULES DE BASE (AVEC TAILLE, EN-TETES DE CONTROLE), SIGNAUX ENVOI/ACQUITTEMENT	MEDIA EXPLICITE : FILS PHYSIQUES OU DES ARCHITECTURES REELS DES BUS

Les principales métriques de l'évaluation des performances du sous-système d'interconnexion sont :

- **le trafic**, exprimant le rapport entre le nombre de paquets insérés dans le réseau et le nombre de paquets sortant du réseau, ou les paquets traversant un tel nœud, dans un tel intervalle de temps ;
- **la topologie de connexion**, décrite par exemple par topologie du réseau, le chemin de routage et le taux de perte de paquets dans des commutateurs ;
- **la technologie de connexion** décrite par exemple par la longueur totale des fils ou la quantité de logique présente dans le routeur de paquets ;
- **les demandes de l'application** qui sont par exemple le délai, débit et largeur de bande requièrent.

Une grande variété de **paramètres de conception** émerge aux différents niveaux d'abstraction, comme suit :

- ***A un haut niveau*** : des paramètres valides sont le débit ou la latence.
- ***Au niveau intermédiaire*** : des paramètres valides sont les temps des transactions et la stratégie d'arbitrage.
- ***Au niveau bas*** : les protocoles au niveau de fils et de pins permettent des mesures exactes pour les délais du système.

Les **outils d'évaluation des performances** sont détaillés ensuite. La simulation est la stratégie la plus utilisée dans l'évaluation des performances pour des sous-systèmes d'interconnexion à différents niveaux d'abstraction: niveau comportemental [Laj 98], niveau cycle [Row 97], niveau TLM [Pes 04]. Des modèles de simulation de communication peuvent être combinés avec la cosimulation logicielle/matérielle, à différents niveaux d'abstraction pour l'évaluation du système global [Hin 97] [Pol 03].

Par exemple [Pol 03] analyse l'impact sur la performance globale du système de différentes politiques d'arbitrage du bus. La politique d'arbitrage du bus a un rôle très important pour la performance du système, car elle indique quels processeurs ont l'accès aux ressources partagés et ainsi elle peut influencer le comportement en temps réel du système.

La métrique examinée est le temps d'exécution (en incluant la communication) de l'application communicant par un bus partagé. **Les paramètres de conception** sont les différentes politiques d'arbitrage, les paramètres des politiques d'arbitrage, les différents modèles de trafic, les différents mappings de l'application (en variant le nombre de processeurs) et les nécessités de communication de l'application (différents scénarios de trafic).

L'abstraction du sous-système à analyser est son modèle de description en SystemC, au niveau transactionnel. **La méthode d'évaluation des performances** est basée sur la simulation. La plateforme de simulation est nommée MPARM ; MPARM est un environnement de simulation globale pour les architectures SoC. MPARM permet la simulation en SystemC jointe avec des simulateurs de processeurs (ISS) ARM (à l'aide des adaptateurs vers SystemC). L'architecture de communication est basée sur le modèle du bus AMBA.

3.5 Modèles des systèmes MPSoC hétérogènes

Un système hétérogène embarqué multiprocesseur monopuce (MPSoC) est par définition un système constitué de plusieurs sous-systèmes différents comme matériel, logiciel et interconnexion. Dans cette étude nous considérerons le MPSoC comme l'interconnexion et la synchronisation entre les différents sous-systèmes logiciels, matériels et les logiciels s'exécutant sur des sous-systèmes CPU.

Le plus haut niveau d'abstraction est le niveau fonctionnel. Le système est représenté par l'application décrite d'une manière fonctionnelle. Le partitionnement logiciel/matériel n'est pas nécessairement établi à ce niveau. Il y a le cas où l'application est décrite d'une manière parallèle et s'exécute sur plusieurs fils d'exécution. La communication est réalisée par des primitives de haut niveau comme envoi/réception des messages. Les interfaces, les interconnexions et la synchronisation sont abstraites.

Un niveau intermédiaire est le niveau d'architecture virtuelle. L'application est toujours décrite à un haut niveau d'abstraction (fonctionnel), mais le partitionnement en différents modules a été réalisé. Les interconnexions et la synchronisation sont explicites mais les interfaces logicielles/matérielles sont encore abstraites. A ce niveau, le système est décrit comme un ensemble de modules virtuels, ports virtuels, canaux virtuels. Ensuite, pendant la conception cette description sera raffinée pour former l'architecture finale (réelle). D'habitude, aux niveaux intermédiaires, les interfaces sont abstraites pour permettre des descriptions et connexions des éléments hétérogènes : modules différents, avec leur propres interfaces de communication.

Pendant le raffinement d'architectures, les interfaces logicielles/matérielles sont conçues comme des interfaces d'adaptation entre ces interfaces différentes.

Le plus bas niveau considéré est décrit par l'architecture RTL des sections implémentées en matériel, couplées au niveau ISA pour les parties logiciels. Il a les caractéristiques des deux niveaux : RTL (Chapitre II.3.1) et ISA (Chapitre II. 3.3), mais aussi des interconnexions, interfaces et synchronisation explicitement décrites.

Ce qui rend la description des systèmes MPSoC difficile à gérer et à simuler est leur hétérogénéité. Les MPSoCs sont composés de sous-systèmes qui peuvent être décrits de différents niveaux d'abstraction. Par exemple, dans le même système, des composants matériels au bas niveau peuvent être décrits ensemble avec des applications logicielles, décrites d'une manière fonctionnelle. Ils peuvent communiquer par des protocoles explicites ou par des transactions. Pour la simulation on aura donc besoin des adaptateurs entre les différents composants ou les différents manières de communication.

Table 5. Niveaux d'abstraction des MPSoC

		ABSTRACTION :	
		DESCRIPTION LOGICIELLE/MATERIELLE	INTERFACES INTERCONNEXIONS, SYNCHRONISATION
NIVEAU D'ABSTRACTION	HAUT : FONCTIONNEL	NON, APPLICATION (MONO/MULTI-THREAD)	ABSTRAITES
	INTERMEDIAIRE : ARCHITECTURE VIRTUELLE	OUI, APPLICATION (MONO/MULTI-THREAD – MAPPE SUR UNE ARCHITECTURE)	INTERCONNEXIONS ET SYNCHRONISATION - EXPLICITES ; INTERFACES - ABSTRAITES
	BAS : RTL/ISA	OUI, EXPLICITEMENT DECRITES : LOGICIEL – ISA, MATERIEL - RTL	EXPLICITEMENT DECRITES

Les métriques d'évaluation des performances pour les systèmes MPSoC peuvent être vues comme une union de métriques spécifiques aux sous-systèmes logiciel, matériel et d'interconnexion : par exemple le temps d'exécution, la capacité de la mémoire, et la consommation d'énergie.

Une grande variété de **paramètres de conception** émergent pour chacun des sous-systèmes impliqués. La plupart du temps, ces sous-systèmes sont décrits à différents niveaux d'abstraction et ils présentent des alternatives multiples de réalisation. Ces choix sont considérés pendant l'analyse du système, et exploitées pour son optimisation.

Différentes **outils d'évaluation des performances** existent. Ils peuvent être développés pour des sous-systèmes spécifiques, comme ceux présentés en [Bac 04] [Gup 00] [Li 97] [Mar 01] [Max 04] ou pour le MPSoC complet, comme décrit en [Laj 00] [Log 04] [Logh 04]. La section suivante désigne quelques environnements représentatifs d'évaluation des performances pour les MPSoCs.

4 L'état de l'art pour l'évaluation des performances des MPSoC

Comme il est montré dans les sections précédentes, beaucoup de méthodes et outils d'évaluation des performances sont disponibles pour différents sous-systèmes: matériels, logiciels, interconnexion, et même pour les MPSoCs. Ils incluent une grande variété de stratégies de mesure, de niveaux d'abstraction, de métriques évaluées et de techniques employés. Cependant, il y a toujours un écart entre les outils particuliers d'évaluation qui considèrent seulement des sous-systèmes isolés et une évaluation des performances pour le système MPSoC global.

Le système MPSoC global contient l'assemblage de sous-systèmes matériels, logiciels et d'interconnexion. D'où on saisit le besoin de pouvoir évaluer les performances globales de tous les sous-systèmes composants, aux différents niveaux d'abstraction.

La cosimulation est une méthode bien adaptée pour l'évaluation des performances des systèmes hétérogènes. La cosimulation offre flexibilité et modularité, nécessaires pour coupler diverses exécutions pour différents sous-systèmes, aux différents niveaux d'abstraction et même spécifiés dans différentes langages. La précision de l'évaluation des performances par cosimulation dépend des modèles choisis pour représenter les sous-systèmes, et de leur synchronisation globale.

Peu d'outils d'évaluation pour les MPSoCs sont rapportés dans la littérature [Bac 04] [Gup 00] [Hen 05] [Laj 00] [Logh 04] [Mar 01] [Max 04]. La principale restriction dans les approches existantes est l'utilisation d'un modèle homogène de représentation

pour le système global (qui ne serait pas adapté à notre approche de description hétérogène du MPSoC). Une autre restriction est l'utilisation des méthodes d'évaluation, qui ne permettent pas l'exploration d'architecture et de tester un nombre massif de solutions, car ils sont lents ou imprécis.

L'approche SymTA/S [Hen 05] augmente le niveau d'abstraction de la description des systèmes MPSoC hétérogènes complexes. Il se sert d'un modèle standard des événements pour représenter la communication et le calcul. Ce modèle standard des événements provient de la représentation des systèmes en temps réel. Il inclut entre autres des événements périodiques, des chaînes d'événements (avec des déviations), et des chaînes en rafale (« burst »). Le modèle considère le comportement complexe tel que le contrôle des interruptions et le calcul dépendant des données. L'approche permet l'analyse précise de l'exécution. Toutefois, pour l'appliquer il est exigé de décrire le système à l'aide de ce modèle spécifique proposé.

L'approche MPARM décrite en [Logh 04] et [Log 04], désigne une plateforme complète de simulation. Elle contient : plusieurs sous-systèmes logiciels s'exécutant en parallèle sur des ISS des processeurs ARM (enveloppés dans des interfaces décrites en SystemC), leurs mémoires, des modules matériels (de synchronisation) et le sous-système d'interconnexion. La plateforme est décrite en SystemC, niveau cycle-près. Les paramètres de configuration de la plateforme sont : le type de l'interconnexion, le nombre des éléments de calculs (processeurs), les paramètres de la mémoire cache des processeurs, et les paramètres des mémoires globales.

Des aspects variés du MPSoC sont évalué par l'interconnexion des différentes simulateurs (par exemple simulation SystemC jointe avec l'exécution sur un ISS) et différents modèles des composants. Par exemple, en [Log 04], l'objectif est l'analyse des performances pour le sous-système d'interconnexion. En [Logh 04], l'objectif est l'analyse de la consommation d'énergie pour le MPSoC global.

L'approche de co-estimation, présentée en [Laj 00], est basée sur l'exécution concurrente et synchronisée de multiples estimateurs pour l'analyse des systèmes embarqués logiciels/matériels. Divers estimateurs fonctionnant à différents niveaux d'abstraction peuvent être interconnectés de cette façon. L'approche est intégrée dans la plateforme de conception de système POLIS et la plateforme de simulation PTOLEMY. En cette approche, l'analyse est dédiée à l'estimation de puissance. A

l'aide de la co-estimation, plusieurs choix de conception peuvent être faits: le partitionnement logiciel/matériel, les choix des composants ou des paramètres.

Une manière semblable d'aborder ce sujet est la méthodologie de conception assistée des MPSoCs, nommée **ROSES** [Bag 01] [Ces 02]. ROSES permet la cosimulation des différents sous-systèmes qui peuvent être décrits aux niveaux d'abstraction différents et dans des langages différents.

Cependant, une fois appliqué à un niveau d'abstraction inférieur, toutes les approches d'évaluation basées sur la cosimulation sont lentes (à cause des temps de simulations élevés qu'elles impliquent). Elles ne peuvent pas explorer des grands espaces de solutions. L'idée est de construire des modèles d'évaluation des performances à des hauts niveaux d'abstraction, et de maintenir une haute précision.

L'alternative serait de combiner la simulation avec des méthodes analytiques afin d'accélérer l'évaluation (due aux temps réduits d'évaluation avec des approches analytiques). Cela est similaire aux méthodes employées pour l'exploration d'architectures des CPUs (par exemple sont [Che 01] et [Tiw 94]).

Perspectives

L'étude réalisée dans ce chapitre mène à la conclusion que la littérature offre beaucoup de méthodologies et outils d'évaluation des performances pour différents sous-systèmes, y compris matériel, logiciel, interconnexion, et même pour les MPSoCs. Ces outils considèrent différents niveaux d'abstraction, stratégies et métriques évaluées. Cependant, ils restent des espaces de conception non explorés et il y a toujours un écart entre les outils d'évaluation pour des sous-systèmes particuliers qui tiennent compte seulement des composants isolés d'un système et d'une évaluation complète d'un MPSoC.

Le principal problème des outils actuels est qu'il n'existe encore pas un outil générique, couvrant tous les aspects d'évaluation des performances de MPSoC et diagnostiquant n'importe quel type de sous-système. L'évaluation globale d'un MPSoC est fondamentale, tandis qu'elle associe des sous-systèmes matériels avec des sous-systèmes logiciels, néanmoins visant un niveau élevé d'abstraction afin de

maîtriser la complexité. Seulement quelques tentatives dans l'évaluation de MPSoC ont visé l'analyse complète de système, comme par exemple ceux décrits dans la section précédente (Chapitre II.4) mais ils ont traité les modèles approximatifs et imprécis.

Suite à l'étude suivie, nous avons identifié les principales tendances dans l'évaluation des performances pour un MPSoC. Ils sont les facteurs clé de l'évaluation de performances, manquant de la recherche courante. Nous avons identifié cinq facteurs majeurs que l'on va nommer ensuite. Ils ne sont pas totalement indépendants ; dans différents contextes de la conception ils peuvent être recouvrants. Chacun sera détaillé dans les paragraphes qui suivent.

- **(1) Une méthode complète d'évaluation des performances** pour MPSoCs hétérogènes, qui considère l'effet global de chaque sous-système individuel : matériel, logiciel ou réseau d'interconnexion ;
- **(2) Une description unifiée** pour les divers modèles de description de chaque composant et leur interaction ;
- **(3) Une interconnexion cohésive** pour tous ces sous-systèmes communiquant différemment ;
- **(4) Une interface cohérente** et structurée avec les concepteurs/utilisateurs, pour répondre aux différents besoins dans l'évaluation des performances et indiquer les directions d'optimisation, à base des résultats de l'évaluation ;
- **(5) Une accélération de l'évaluation des performances** afin d'éviter une étape trop longue dans le flot de conception.

(1) Suite à notre étude pour l'évaluation des performances de différents sous-systèmes, nous pouvons conclure que la meilleure manière d'aborder un système hétérogène comme le MPSoC est d'assembler des différentes techniques auparavant développées et de les amener à un dénominateur commun. En tenant compte de ce fait, nous allons répondre au premier défi énoncé, et proposer dans le Chapitre IV de coupler divers modèles de simulation de sous-systèmes à différents niveaux

d'abstraction et indiqués dans différentes langages, dans un environnement de cosimulation comme ROSES [Ces 02].

(2) La description de l'architecture virtuelle facilite l'assemblage des fonctions de calcul et communication, dans une vue unifiée. Cela fournit une perspective pour le deuxième défi. L'évaluation des performances au niveau d'architecture virtuelle fournit également l'avantage d'une vue cohérente du système global. La partie « fonctionnelle » des modèles de simulation des sous-systèmes matériel, logiciel, ou d'interconnexion pourrait être représentés par le programme (en langage C++) décrivant le comportement de chaque module virtuel, alors que leur communication virtuelle pourrait accueillir la synchronisation de l'évaluation.

(3) Le troisième défi explore les manières de relier ensemble tous ces composants virtuels communiquant différemment. Une proposition serait de joindre les composants virtuels par des enveloppes d'adaptation. Une enveloppe d'adaptation permettra l'interconnexion des sous-systèmes différents, par l'adaptation de leurs protocoles de communication. Le travail déjà existant est effectué dans le domaine de la communication logicielle/matérielle, produisant automatiquement des enveloppes de cosimulation [Bag 01], [Yoo 01] et [Gau 01]. Dans le cas de l'évaluation des performances, ces enveloppes devraient avoir les rôles multiples : en plus d'apporter tous les différents outils spécifiques d'évaluation des performances à un dénominateur commun et de les synchroniser, mais aussi la gestion de tous les aspects liés à la performance.

(4) Le quatrième défi implique la conception des enveloppes d'adaptation. Ces enveloppes doivent être conçus avec une structure générique, combinant des aspects fonctionnels et de communication. Malgré leur généralité, des aspects spécifiques doivent être mis en application, selon la nature du sous-système analysé, son niveau d'abstraction et des métriques mesurées.

(5) Un aspect essentiel dans l'évaluation des performances, est le cinquième défi, énonçant l'interface avec l'utilisateur et ses différents besoins dans l'évaluation. Un outil complet pour simplement diagnostiquer le système, n'existe pas. Par conséquent, il devrait être facile de inclure des « sondes de mesure » pour l'acquisition des statistiques des profils d'exécution matériels, logiciels ou d'interconnexion.

Un autre aspect important, connecté à ce cinquième défi, est la modularité pour permettre la modification les modèles de performances : des modèles plus abstraits afin d'accélérer l'évaluation, ou au contraire, plus précis pour capturer un aspect essentiel. En conséquence, par la variation des différents composants, l'outil d'évaluation serait extrêmement approprié à l'exploration de l'espace de conception. Dans ce cas, un modèle formel de performance pourrait rapidement prévoir les performances d'un nouveau système.

Conclusions

L'évaluation des performances est une étape clé dans le processus de conception. Les facteurs influencés par l'étape de l'évaluation sont le temps de conception et l'optimalité du résultat.

La première contribution de ce chapitre a été une analyse des éléments fondamentaux utilisés dans l'évaluation des performances des différents sous-systèmes embarqués. Ensuite, une deuxième contribution, a utilisé ces aspects pour l'étude des solutions proposées actuellement dans l'évaluation des performances. Le résultat de ces analyses a illustré que les outils actuels sont bien adaptés pour des systèmes réduits ou pour des sous-systèmes particuliers. Par conséquence il serait désirable de disposer d'une méthode d'évaluation rapide et précise et qui puisse tenir compte de l'ensemble des sous-systèmes.

La conclusion est le besoin impératif pour un outil global d'évaluation des performances des MPSoCs. La direction qui s'entrevoit de cette étude est de concevoir une méthode pour l'évaluation des performances globale en s'appuyant sur la composition des outils individuels des différents sous-systèmes.

CHAPITRE III.

Modèles d'évaluation de performances pour les sous-systèmes logiciels

Introduction

Ce chapitre présente l'outil que nous avons développé pour l'évaluation rapide et précise des temps d'exécution pour le sous-système logiciel embarqué, nommé « ChronoSym ». L'outil ChronoSym est appliqué pour l'évaluation des systèmes d'exploitation générés automatiquement dans le flot de conception ROSES, et présentés dans le Chapitre I.2.1 de cette thèse. Il présente l'avantage de permettre la manipulation des amples systèmes MPSoC, basé sur une haute vitesse de simulation et une bonne précision.

L'idée centrale de la solution proposée est d'augmenter la vitesse d'évaluation des temps d'exécution pour le logiciel embarqué en utilisant son exécution directement sur la machine hôte (procédé nommé « exécution native »). Pour satisfaire les contraintes de précision, ce modèle doit être cohérent avec la réalisation finale du système d'exploitation et de tenir compte de la synchronisation avec le sous-système matériel. Par suite, en plus de l'évaluation des performances de l'application, il est nécessaire de mesurer aussi les performances du SE ; par exemple les temps d'exécution pour le SE.

De plus, ce modèle est inclus dans un schéma de cosimulation logicielle/matérielle. Un modèle d'interface logicielle/matérielle sera proposé pour considérer : les interactions détaillées avec les modules matériels, les interruptions matérielles, le traitement des entrées/sorties et tout en tenant compte de l'avancement du temps dans les modules logiciels.

Dans ce chapitre, notre modèle de performances sera nommé spécifiquement : « modèle d'exécution native temporelle », d'après la méthode qui est derrière. Même si on l'a conçue pour le but de l'évaluation des temps d'exécution, cette méthode est plus large et pourrait être appliquée à plusieurs étapes (validation, etc.) ou à plusieurs métriques (nombre d'accès mémoire, etc.).

La première section présente les différentes techniques de cosimulation logicielle/matérielle, focalisant sur les deux principales techniques d'exécution du logiciel : avec un simulateur de processeur (ISS) et par exécution native. Les différents profils de vitesse et précision de ces techniques d'exécution sont mis en évidence. La deuxième section présente la méthode proposée pour l'évaluation des performances du sous-système logiciel embarqué. Elle est dédiée à la description du modèle du logiciel embarqué proposé pour l'évaluation des performances et l'outil ChronoSym qui génère ce modèle. La troisième section détaille les éléments de base du modèle du logiciel embarqué : la modélisation de partie du code SE dépendant du matériel (le modèle du HAL), la fonction d'annotation avec les temps d'exécution (fonction **`DELAY()`**) et le module d'adaptation à la cosimulation avec le système global (le module TBFM[1]). La quatrième section présente l'application de l'outil ChronoSym à deux applications, un modem VDSL[2] et l'application McDrive.

1 L'évaluation des performances des sous-systèmes logiciels

Cette section a le rôle d'introduire les techniques les plus usuelles pour l'évaluation (par simulation) du logiciel embarqué : l'exécution native et la simulation à l'aide d'un ISS. Ensuite, les différents types de cosimulation logicielle/matérielle impliquant d'exécution logicielle native (la cosimulation fonctionnelle et la cosimulation basée sur le modèle du Système d'Exploitation) et la simulation logicielle sur un ISS sont détaillés. A la fin de cette section on compare ces techniques de cosimulation du point de vue de la vitesse et la précision de l'évaluation des performances.

[1] Timed Bus Functional Model
[2] Very-high-data-rate Digital Subscriber Line

1.1 Techniques de cosimulation logicielle/matérielle

1.1.1 Techniques d'exécution logicielle : ISS vs. exécution native

Pour l'évaluation des sous-systèmes logiciels, les travaux existants utilisent la technique de simulation à base d'un simulateur du processeur (ISS) ainsi que la technique d'exécution native. Mais, comme nous allons le montrer dans cette section, aucun de ces techniques n'offre pas une bonne précision et à la fois une vitesse élevée.

L'ISS est un outil (un programme écrit en langage évolué – par exemple C) qui est compilé et s'exécute sur la machine hôte, et qui émule la fonctionnalité d'un processeur. Il modélise le processeur cible au niveau du jeu d'instructions, en considérant l'état interne du processeur cible comme par exemple les registres (registres de programme **`PC`**, registres d'instructions **`RI`**, etc,), la mémoire interne (par exemple un tableau **`M`**), ou la mémoire externe du processeur. Il permet ainsi la simulation précise d'une application qui est censée s'exécuter sur le processeur. Les actions qu'il exécute pour cette simulation sont : la lecture de l'instruction[3], le décodage[4], la lecture des opérandes en mémoire[5], et l'exécution de l'instruction[6].

On peut décrire très schématiquement le fonctionnement d'un ISS, en explicitant les étapes de la boucle d'exécution :

```
//initialisations
PC <- 0 ;
…
pour toujours faire :
      RI <- M[PC] ;
      selon la signification du RI :
            mettre à jour les registres (et la mémoire) ;
            mettre à jour le PC.
```

Il peut y avoir deux sortes d'ISS : des ISSs interprétatifs et des ISSs avec compilation. De la première catégorie font partie la plupart des ISS commerciaux disponibles sur le marché. Ils utilisent la technique de simulation par interprétation, et ils respectent les étapes de lecture de l'instruction, décodage, lecture des opérandes en mémoire, et exécution de l'instruction. Mais, l'interprétation de chaque instruction, incluant tous

[3] fetch
[4] decode
[5] read
[6] execute

les cinq étapes citées ci-dessus réduit considérablement les performances en vitesse de ce type d'ISS.

Pour compenser les performances réduites des ISS interprétatifs, des travaux ont été effectués sur les simulateurs de processeurs utilisant la technique de simulation par compilation : l'ISS avec compilation [Ziv 96]. Chaque instruction du processeur cible est traduite en un ensemble d'instructions pour le processeur hôte. Cependant, les limitations de ces ISSs sont : le manque de portabilité, la nécessité d'un grand espace de mémoire et la difficulté de modéliser les éléments spécifiques au processeur (les interruptions). Par conséquence, malgré leurs problèmes de performances, les ISS interprétatifs sont encore utilisés pour la simulation précise au niveau cycle.

Les ISS (de chaque type) peuvent avoir différents degrés de précision, dépendant de comment ils modélisent l'architecture du processeur : le chemin de données effectif, les registres internes, les états de la mémoire, etc., ou s'ils simulent seulement la sémantique des instructions en langage machine, sans tenir compte de l'architecture réelle du processeur. Aussi, le degré de précision dépend de la simulation au niveau de chaque cycle d'horloge, ou des simulations optimisées : simulations événementielles, simulations de plusieurs cycles à la fois, etc.

Pour plus de détails sur la simulation à l'aide de l'ISS, il faut faire référence à [Row 94] [Ziv 96].

Malgré le grand avantage de la précision, les ISSs ont des vitesses insatisfaisantes pour la complexité des applications courantes. L'interprétation de chaque ligne en code assembleur réduit considérablement les performances en vitesse de ce type de simulation. Ainsi le temps d'évaluation et implicitement le processus de conception des systèmes sont ralentis considérablement. Pour cela, les ISS sont rarement utilisés aux phases initiales de conception.

L'exécution native est réalisée sur la machine hôte (machine de développement) sans prendre en compte l'architecture du processeur cible pour lequel le logiciel à été prévu. Le code spécifique au processeur cible (par exemple le code assembleur) ne

peut pas être exécuté dans ce cas. Aussi, la simulation native des logiciels complexes (incluant plusieurs tâches et un système d'exploitation) nécessite un modèle de simulation pour le système d'exploitation. L'avantage de cette technique de simulation est la vitesse, mais la contrepartie est la perte en précision. Cette approche requiert des efforts supplémentaires afin de pouvoir extraire des informations sur les performances au niveau temps d'exécution du logiciel à simuler.

Pour appliquer ces deux méthodes aux MPSoC, la pratique la plus adéquate est de les intégrer dans la cosimulation du système [Nic 01] [Row 94]. Dans le processus de cosimulation avec le matériel, la simulation du logiciel peut être réalisée par trois techniques : (1) simulation fonctionnelle, (2) simulation basé sur un modèle de SE (qui est la représentation fonctionnelle, simplifiée du SE) et (3) simulation à l'aide d'un ISS. Pour les deux premières techniques, le logiciel est exécuté nativement ; ils peuvent inclure la notion de temps comme annotation.

Ces trois techniques se distinguent par les environnements d'exécution utilisés, le temps de simulation et la précision des résultats fournis, et elles seront présentées dans la suite de cette section. D'abord on présente les particularités de ces trois types de cosimulation. Dans les sections qui suivent, on présentera leur différences en termes de vitesse et précision.

1.1.2 Cosimulation fonctionnelle

La cosimulation fonctionnelle est la cosimulation de haut niveau entre l'exécution native du logiciel décrit à un haut niveau d'abstraction et une exécution native d'une description fonctionnelle du matériel (Figure 11). La description fonctionnelle du matériel est un programme qui représente la description de son comportement. Les sous-systèmes logiciel et matériel sont représentés en tant que modules fonctionnels, communiquant par des transactions. Évidemment, cette simulation considérera seulement le comportement du système.

Bien sûr, il y a des cas où le logiciel s'exécute nativement ensemble avec des simulations du matériel.

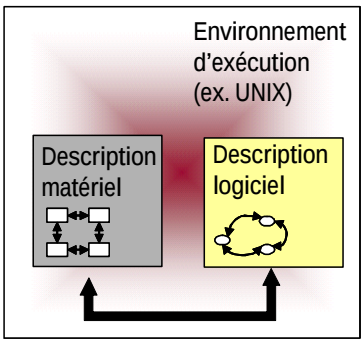


Figure 11. Cosimulation logicielle/matérielle fonctionnelle

Il y a eu peu de recherche sur la cosimulation synchronisée entre l'exécution native du logiciel et du matériel. Ainsi, une telle cosimulation n'offre pas la possibilité d'évaluer les performances du temps d'interaction entre le logiciel et le matériel. Les détails des interactions logicielles/matérielles (par exemple entrées/sorties, interruptions) ne sont pas simulés.

Un autre problème de ce type de simulation est la prise en compte du système d'exploitation. Les tâches de l'application logicielle s'exécutent en parallèle, sans faire appel à un ordonnanceur du SE (on verra plus tard en section 1.3 de ce chapitre, Figure 15 (b)). Ils s'exécutent sur l'environnement du simulateur hôte, par exemple SystemC [SystemC]. Dans ce cas-ci, SystemC est utilisé en tant que environnement de simulation et ordonnanceur.

L'exécution native du SE est largement répandue dans le domaine du logiciel embarqué. Elle permet le déboguage de l'application logicielle et pour faciliter la conception du SE. Les détails concernant le SE natif peuvent être trouvés dans [Tan 95]. Puisque les SE natifs sont les SE réels portés pour la simulation sur la machine hôte de simulation, ils peuvent surmonter le problème des modèles de simulation du SE.

1.1.3 Cosimulation basée sur le modèle du Système d'Exploitation

Le cas de la cosimulation basée sur le modèle du SE (Figure 12) illustre un autre cas d'exécution native de l'application logicielle. Par rapport au cas précédent de simulation purement fonctionnelle du code de l'application, maintenant on utilise un modèle de SE [Bra 02] [Des 00] [Ger 03] [VCC] [VxSim]. Le modèle du SE est une description simplifiée du fonctionnement du SE, qui émule son API et inclut aussi ses

fonctions spécifiques comme l'exécution multitâche, le traitement des interruptions et la gestion de périphériques.

La simulation basée sur le modèle du SE permet de modéliser l'avancement séquentiel des tâches logicielles multiples qui s'exécutent sur le même processeur (on verra plus tard en section 1.3 de ce chapitre, Figure 15 (c)). Cependant, le SE utilisé n'est pas le SE réel car seulement l'API du SE est émulée. De ce fait, l'impact du SE sur l'exécution du système n'est pas correctement simulé, par exemple le surcoût de l'exécution du SE n'est pas considéré.

Comme on a déjà montré, pour des raisons de rapidité, il est préférable d'utiliser l'exécution native pour l'évaluation des performances du sous-système logiciel. Cependant, l'utilisation conventionnelle du système d'exploitation natif (par exemple UNIX) manque toujours la modélisation de la partie matérielle du système et la représentation des temps d'exécution, vu qu'elle exécute une simulation fonctionnelle.

Pour la cosimulation du logiciel avec les éventuelles parties matérielles, on utilise ce qu'on appelle un « bus de cosimulation ». Ce bus de cosimulation est un ensemble de fonctions qui font la translation entre l'interface de communication du matériel et l'interface de communication du logiciel. Le bus de cosimulation doit utiliser des primitives appartenant aux deux simulateurs : logiciel et matériel. Généralement, il se sert des primitives du SE s'exécutant sur la machine hôte (par exemple UNIX).

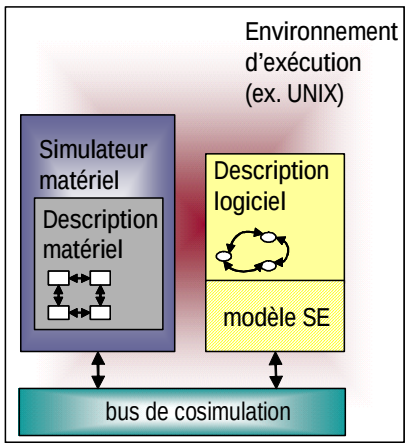


Figure 12. Cosimulation logicielle/matérielle basée sur le modèle du Système d'Exploitation (SE)

En [VCC] il est présenté un modèle du SE qui spécifie analytiquement le temps d'exécution de l'ordonnancement des tâches, en tant qu'une valeur du temps cumulée

dans une fonction. Par exemple le délai de l'ordonnancement « earliest-deadline first »[7] est calculé avec la fonction **`a+b*n*log(n)`**, où **`n`** est le nombre de tâches et **`a`** et **`b`** sont des paramètres résultés après un processus d'ajustement de courbes[8]. Une telle fonction de délai manque d'exactitude puisqu'elle donne une valeur d'une granularité grossière.

Carbon Kernel [CarK] et SoCOS [Des 00] sont des exemples de modèles de simulation pour le SE. Les deux modèles simulent les interfaces de programmation de l'application (API). Un inconvénient significatif de ces modèles est qu'ils ne reflètent pas le comportement du vrai SE en termes de fonctionnalité ou temps d'exécution. Ils ne sont pas de vrais SE, mais des modèles de simulation du SE.

Une autre approche qui propose l'exécution native en utilisant un SE virtuel (qui est un modèle de SE, imitant sa fonctionnalité) est celui de WindRiver Systems Inc. Il fournit VxSim comme modèle de simulation de VxWorks [VxSim]. Le désavantage de cette approche est de ne pas pouvoir simuler le logiciel avec le matériel situé autour du processeur sur lequel le SE s'exécute.

1.1.4 Cosimulation basée sur l'ISS

La cosimulation associant un l'ISS et une simulation matérielle est illustrée dans la Figure 13. Comme on a déjà montré, l'ISS est un simulateur qui exécute le logiciel à l'aide des concepts spécifiques au matériel. Ainsi, il offre une bonne précision et il est capable de simuler différents types d'architectures et d'applications. Ce type d'évaluation est très utile par sa bonne précision au niveau cycle d'horloge ; il est utilisé avec prépondérance dans les phases finales de la conception.

En termes de simulation du SE, en [Row 94] est présentée la technique de cosimulation à l'aide de l'ISS. Cet ISS peut exécuter entre 10K et 100K instructions par seconde, n'étant pas adapté pour simuler des logiciels comportant plusieurs milliers de lignes de code. En plus, si ces simulateurs sont interconnectés en parallèle, ou si ils communiquent avec des simulateurs matériels, la vitesse de simulation diminue.

---

[7] Algorithme d'ordonnancement à priorité dynamique. Une tâche est d'autant plus prioritaire que sa date d'échéance absolue est proche de la date courante.

[8] curve fitting

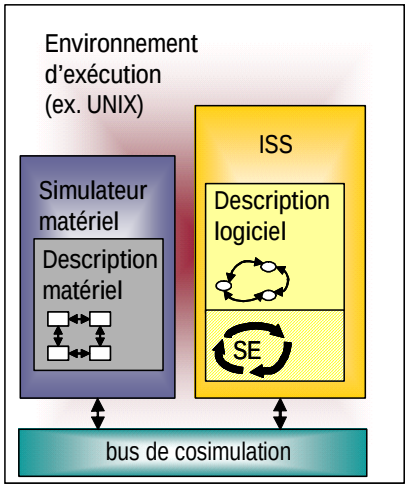


Figure 13. Cosimulation logicielle/matérielle basée sur l'ISS

1.2 La vitesse de l'évaluation des performances par cosimulation logicielle/matérielle

Les trois modèles de cosimulation présentés dans la section précédente ont des vitesses différentes de simulation, particulièrement pour la simulation du logiciel.

- La plus haute vitesse est obtenue en employant la ***cosimulation fonctionnelle***. Dans ce cas les modules logiciels et matériels sont exécutés de façon native sur la machine hôte (Chapitre III.1.1.2).
- La ***cosimulation à l'aide de l'ISS*** donne la plus basse vitesse puisque l'ISS simule l'exécution détaillée de processeur, interprétant chaque ligne du programme assembleur (Chapitre III.1.1.4).
- La ***cosimulation basée sur le modèle du SE*** maintient une vitesse de simulation élevée, puisque le code de l'application logicielle est exécuté nativement sur la machine hôte (Chapitre III.1.1.3).

Habituellement, l'exécution native du logiciel est de deux ou même trois ordres de grandeur plus rapide que la simulation à l'aide de l'ISS [Row 94] [Unr 00]. La Figure 14 est un exemple avec le rôle d'expliquer cette différence.

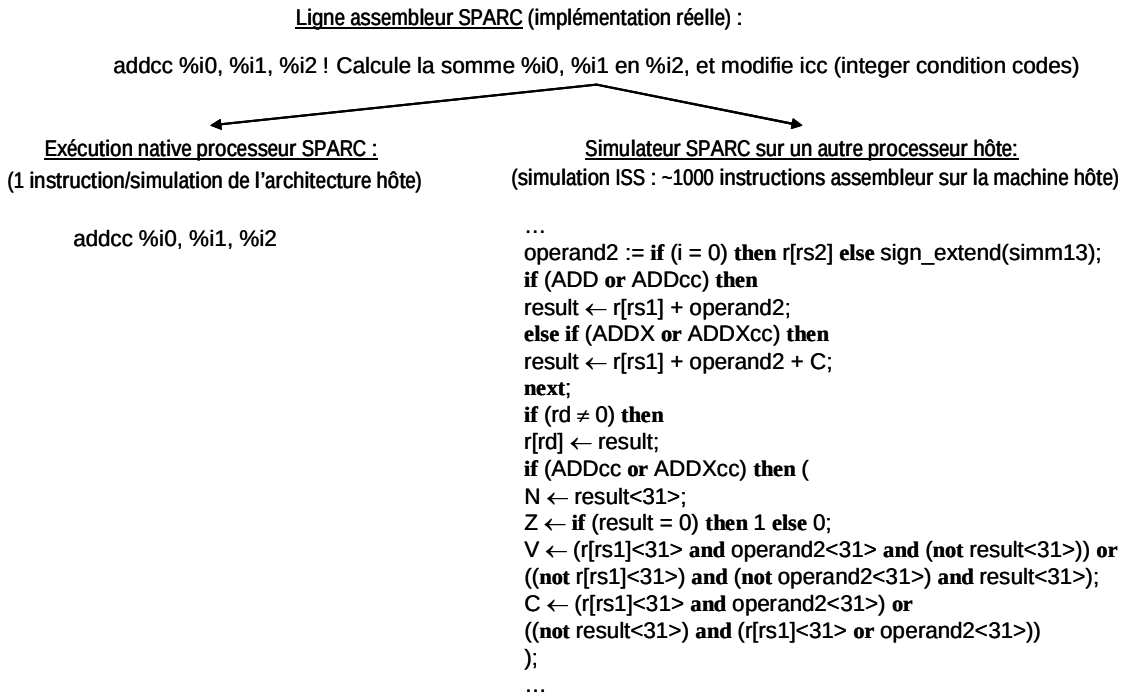


Figure 14. Comparaison entre les nombres d'instructions utiles pour simuler/exécuter une seule ligne d'assembleur du processeur cible

Pour exécuter la ligne de code « **`add %i0, %i1, %i2`** » sur un processeur SPARC comme architecture cible, l'on a montré dans les sections précédentes qu'il y a deux possibilités. La première possibilité est de compiler et d'exécuter nativement ce code sur la machine hôte (qui peut être n'importe quel processeur, incluant un processeur SPARC). La deuxième possibilité est d'utiliser un simulateur de processeur (ISS) dédié au processeur SPARC. Dans ce dernier cas, c'est l'ISS qui sera compilé et exécuté sur le processeur hôte ; le fragment de code que l'on veut tester sera une entrée pour l'ISS.

La branche gauche de la Figure 14 est un exemple d'exécution native : dans ce cas sur un processeur SPARC comme processeur hôte. La correspondance est évidement 1 :1. Même si l'on va exécuter cette instruction sur un autre processeur hôte, la correspondance sera toujours linéaire. L'instruction respective peut se traduire en une ou plusieurs instructions assembleur équivalentes sur le (deuxième) processeur hôte. Ainsi, comme l'exécution native garde le même ordre de grandeur d'instructions sur le processeur hôte comme sur le processeur cible, elle bénéficie d'une vitesse élevée d'exécution.

Par contre, pour exécuter le code à l'aide d'un ISS, le nombre d'instructions qui vont s'exécuter réellement sur le processeur hôte augmente considérablement. La différence est due au fait que l'ISS simule le code qui lui est fourni, mais aussi la fonctionnalité du processeur : par exemple les traitements d'exceptions, les interruptions, les étapes du pipeline comme la lecture de l'instruction, le décodage, la lecture des opérandes en mémoire et l'exécution etc. On a déjà vu (en Chapitre III.1.1.1) que le code de l'ISS est décrit dans un langage de haut niveau, qui doit être lui-même exécuté chaque fois qu'une instruction s'exécute. La branche gauche de la Figure 14 montre ce cas. On n'a pas illustré dans la figure toutes les étapes du simulateur (qui comptent environ 500 lignes de code de haut niveau), mais seulement le code correspondant à une instruction « **`addcc`** ». Le code est décrit dans un langage nommé ISP[9]. Pour plus de détails, il faut faire référence au Manuel de l'Architecture du Processeur SPARC [Sparc].

La comparaison entre ces deux types de simulation (native et à l'aide d'un ISS) montre comment l'exécution native peut rapporter environ 100-1000 fois d'accélération, comparé à la simulation à base de l'ISS.

1.3 La précision de l'évaluation des performances par cosimulation logicielle/matérielle

Les trois modèles de cosimulation n'ont pas toutes la même précision :

- La ***cosimulation basée sur l'ISS*** est la plus précise, puisque l'ISS simule avec l'exactitude d'une instruction ou d'un cycle d'horloge (Chapitre III.1.1.4).
- La ***cosimulation fonctionnelle*** est la moins précise puisque le système est décrit seulement par son comportement. Il ne tient pas compte du processeur cible (par exemple de la sérialisation des tâches logicielles) ou des interactions détaillées entre le logiciel et le matériel (par exemple les interruptions) (Chapitre III.1.1.2).
- La ***cosimulation basée sur le modèle du SE*** considère des détails architecturaux, comme par exemple la sérialisation des tâches par le modèle du SE. Ainsi, en comparaison avec la

[9] Instruction-Set Processor notation

cosimulation fonctionnelle, elle possède une meilleure précision. Cependant, elle ne donne pas la même précision que l'ISS puisque le vrai SE n'est pas simulé (Chapitre III.1.1.3).

On analyse la précision temporelle pour les trois types de cosimulation sur un exemple de code pour deux tâches, **`Tâche 1`** et **`Tâche 2`**, représentés dans la Figure 15 (a). Ces tâches s'exécutent sur le même processeur, et dans une vraie réalisation ils auront besoin d'un SE pour gérer leur exécution « parallèle ».

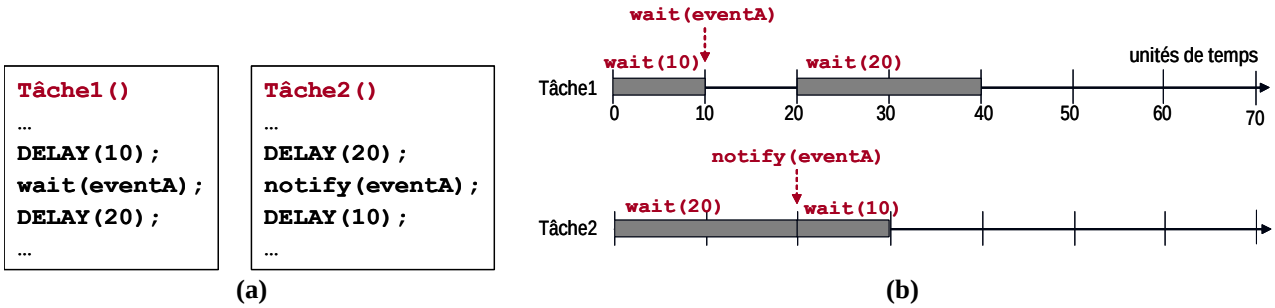


Figure 15. (a) Exemple de code de tâches ; (b) la précision temporelle dans le cas de la cosimulation fonctionnelle

La trace de ***la cosimulation fonctionnelle*** des deux tâches est représentée dans la Figure 15 (b). Les surfaces ombragées représentent que la tâche s'exécute. La tâche appelle la fonction « **`Delay()`** » pour annoncer l'écoulement du temps. Sur l'axe de l'exécution, cette fonction se traduira dans une exécution de la fonction « **`wait()`** », avec le même quanta de temps. La **`Tâche 1`** attend un **`événement A`**, alors que la **`Tâche 2`** lance cet événement.

- On suppose que **`Tâche 1`** exécute **`wait(10)`** et **`Tâche 2`** exécute **`wait(20)`** au **`temps 0`**. Puisque la sérialisation des tâches n'est pas simulée dans ce cas de cosimulation, les deux tâches fonctionneront concurremment entre les moments de **`temps 0`** et **`10`** pour la simulation fonctionnelle sur la machine hôte (la simulation fonctionnelle sur la machine hôte ne fait pas avancer le temps d'exécution de **`0`** à **`10`**, donnant l'impression de parallélisme).

- Au **temps 10**, **Tâche 1** arrête son exécution après avoir appelé **wait(eventA)**, qui attend une notification sur **l'événement A**.
- **Tâche 2** fonctionne ses **20 unités de temps**. Ensuite, elle appelle **notify(eventA)** qui donne la notification d'événement à la **Tâche 1**.
- Puis, **Tâche 1** peut reprendre son exécution ; et les deux tâches fonctionnent encore concurremment entre les moments **de temps 20 et 30**.

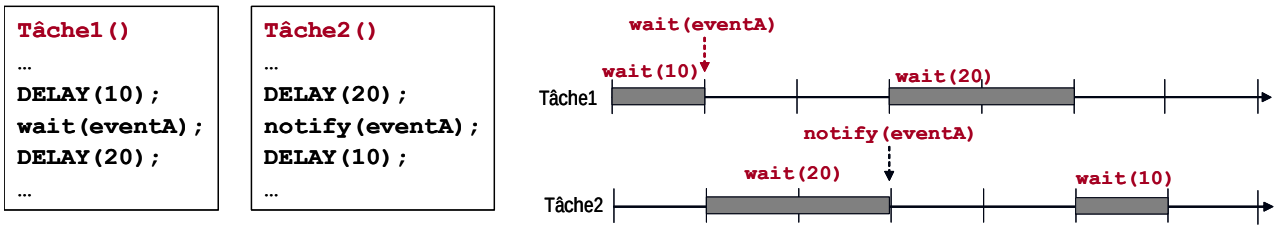


Figure 16. Le cas de la cosimulation basée sur le modèle du SE (sur le même exemple de code de tâches)

Dans ***la cosimulation basée sur le modèle du SE*** (Figure 16), la sérialisation des tâches est simulée en utilisant le modèle du SE. Supposons que le SE donne initialement la main à la **Tâche 1**.

- **Tâche 1** commence à s'exécuter au **temps 0**. Elle exécute **wait(10)** [10] en passant **10** unités de temps.
- Pendant cette période, **Tâche 2** ne fonctionne pas. Elle se déclenchera au **moment 10 de temps**, quand **Tâche 1** aurait fini son exécution. Ainsi, l'exécution en série de tâche est simulée.
- Au **temps 10**, l'appel de la fonction **wait(eventA)** cause le changement de contexte de **Tâche 1** à **Tâche 2**.
- Puis, **Tâche 2** exécute **wait(20)** jusqu'au moment de **temps 30**.

[10] Dans ce cas, la fonction wait() est considérée comme un API du SE.

- Pendant cette période, **Tâche 1** ne fonctionne pas.
- Au **temps 30**, l'appel du **notify(eventA)** cause un autre changement de contexte de **Tâche 2** à **Tâche 1**. La simulation continue de la même façon.

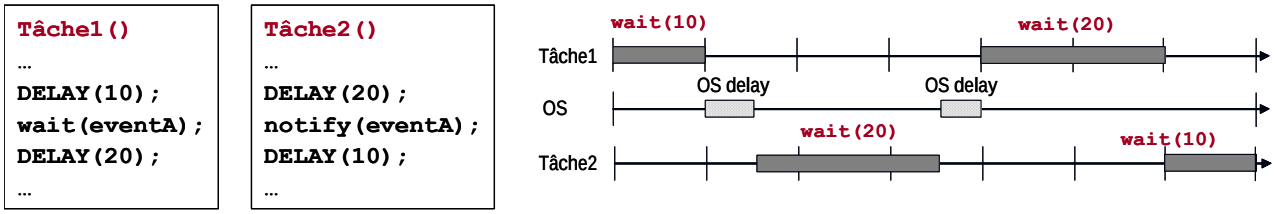


Figure 17. Le cas de la cosimulation basée sur l'ISS (sur le même exemple de code de tâches)

Dans ***la cosimulation basée sur l'ISS***, le vrai SE est simulé. Ainsi, les effets produits par le surcoût du SE sont aussi simulés. Cette cosimulation est illustrée dans la Figure 17.

- **Tâche 1** s'exécute jusqu'au **temps 10** comme dans le cas précédent.
- Au moment de **temps 10**, la fonction **wait(eventA)**[11] est appelée.
- L'ISS simule l'exécution de la fonction **wait(eventA)** en incluant le changement de contexte de **Tâche 1** à **Tâche 2** sur le processeur cible. Puisque le changement de contexte des tâches prend en réalité du temps, la simulation à l'aide de l'ISS rapporte un délai de l'exécution pendant l'exécution du **wait(eventA)**.
- La **Tâche 2** est reprise après le **temps 10** et le délai pour le changement de contexte du SE, comme représenté sur la Figure 17.

Dans le cas de la cosimulation basé sur le modèle du SE, ce délai dû au surcoût du SE, n'est pas pris en considération puisque le vrai SE n'est pas simulé.

[11] Dans ce cas, la fonction wait() est aussi considérée comme un API du SE

2 L'outil d'évaluation du temps d'exécution, ChronoSym

Fondamentalement, pour l'évaluation des temps d'exécution pour le sous-système logiciel embarqué, on exploite l'exécution native pour la simulation du logiciel.

Cette section offre (a) une vue d'ensemble sur l'intégration du modèle d'exécution natif dans le SoC et (b) l'emplacement de l'outil d'annotation automatique avec les temps d'exécution, ChronoSym, dans le flot de conception pour les systèmes sur puce. Cette structure se reflète dans les sous-sections, dont la première présente la conception du modèle natif de simulation pour le sous-système logiciel, en montrant une vue d'ensemble sur les parties du SE qui doivent être re-modélisés (la partie HAL) et aussi sur l'introduction des annotations temporelles dans le code de l'application et du SE. La deuxième sous-section présente notre stratégie d'annotation et de génération du modèle natif pour le logiciel embarqué, dans le cadre du flot ROSES. Les avantages et limitations de notre technique sont brièvement présentés en la troisième sous-section.

2.1 Le modèle de simulation pour le sous-système logiciel

Les trois desiderata de base de l'évaluation des temps d'exécution à l'aide du modèle d'exécution native temporelle, pour des sous-systèmes logiciels complexes sont :

- (1) l'exécution native du code réel du logiciel, comprenant le vrai code du SE, sur un simulateur hôte. Cela est difficile à réaliser car le code du système d'exploitation (SE) comporte des sections spécifiques au processeur, souvent décrites en langage assembleur.
- (2) la simulation des temps d'exécution du logiciel et de ses interactions avec le matériel. Cela est difficile à réaliser car l'exécution native est une simulation fonctionnelle qui ne respecte pas les temps réels d'exécution.
- (3) l'accumulation dynamique (sur le chemin d'exécution) des temps annotés pour le logiciel embarqué pendant une simulation réelle du système. Aussi, le sous-système logiciel ne peut pas être évalué en isolation des autres sous-systèmes embarqués (sous-systèmes matériels et d'interconnexion).

Le défi peut être énoncé comme la nécessité de définir un modèle de simulation pour le système d'exploitation qui devance ces contraintes,

- (a) en traduisant les parties spécifiques au processeur en un modèle indépendant de l'architecture cible,
- (b) en annotant le code de l'exécution native avec des temps spécifiques à l'architecture cible
- (c) en tenant compte des interactions entre les sous-systèmes adjacents dans une exécution native des MPSoC.

On a montré jusqu'à présent que l'évaluation des performances du sous-système logiciel embarqué est réalisée par la cosimulation avec les autres sous-systèmes – répondant au point (c) défini précédemment. La Figure 18 montre comment on construit le modèle de simulation du logiciel à partir de sa description.

En considérant une architecture spécifique de SoC qui contient le processeur, la mémoire, un composant matériel, et le bus du processeur, on localise le logiciel embarqué dans la mémoire du processeur, prêt à être exécuté sur le processeur embarqué (Figure 18 (a)). Le modèle d'exécution native temporelle que nous proposons préserve la structure hiérarchique réelle du logiciel embarqué présentée en Chapitre I.2.2. Le logiciel se compose de la description de l'application logicielle, du code du SE non-spécifique au processeur et du code du SE spécifique au processeur (HAL) (Figure 18 (b)).

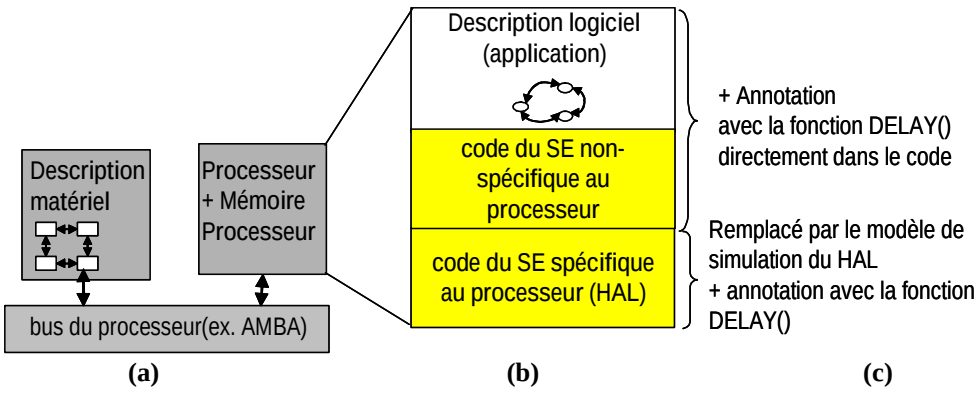


Figure 18. Modèle d'exécution native, temporelle : (a) description du système ; (b) description du logiciel embarqué ; (c) actions à faire pour la construction du modèle logiciel embarqué.

Le modèle d'exécution native temporelle sera obtenu différemment pour les sections du code non-spécifique au processeur, et pour les sections du code spécifique au processeur (Figure 18 (c)). Dans le premier cas, le code réel de l'application et du SE non-spécifique seront préservés. Ils seront annotés avec les délais d'exécution réels sur le processeur ciblé, par l'insertion des fonctions `Delay()` dans le code (répondant au point (b) défini précédemment). La couche HAL sera remplacée par un modèle d'exécution équivalent, qui pourra être exécuté dans l'environnement natif : modèle de HAL décrit en code C (répondant au point (a) défini précédemment). A son tour, ce modèle sera aussi annoté avec les délais d'exécution par l'insertion des fonctions `Delay()`.

Le modèle d'exécution native temporelle remplace le modèle de simulation du processeur et du code logiciel inscrit dans la mémoire du processeur comme illustré dans la Figure 18.

2.2 La stratégie et les différents éléments de l'outil ChronoSym

Pour mettre en oeuvre le modèle de simulation proposé dans la section antérieure, un outil nommé « **ChronoSym** » a été développé et intégré dans le flot de conception des MPSoC (Figure 19). Comme on l'a montré, dans le flot ROSES on part du modèle de description de haut niveau de l'application. Deux phases parallèles génèrent des interfaces logicielles (c.-à-d. le SE) spécifiques à l'application et des interfaces matérielles. Les interfaces matérielles sont conçues pour pouvoir générer la réalisation de bas niveau (c.-à-d. RTL).

Le modèle conventionnel de cosimulation à l'aide d'un ISS peut être établi après la conception du SE et du matériel, comme représenté sur la partie gauche de la figure. L'alternative est de générer un modèle équivalent, basé sur l'exécution native temporelle du logiciel (proposé dans la section précédente), comme représenté sur la partie droite de la figure. Comparé à la cosimulation basée sur l'ISS, dans la cosimulation logicielle/matérielle, le modèle natif d'exécution remplace seulement l'ISS tout en ayant le même modèle de simulation pour le matériel.

Dans l'exemple montré en Figure 19, la description du matériel, l'interconnexion, ensemble avec les interfaces matérielles et de cosimulation (le BFM et TBFM respectivement – leur différence sera expliquée en Chapitre III.3.3) s'exécutent en

SystemC. Dans le premier cas de cosimulation, un ISS exécute le code du logiciel, et dans le deuxième notre modèle construit avec ChronoSym pour du même logiciel embarqué s'exécute nativement sous l'environnement UNIX.

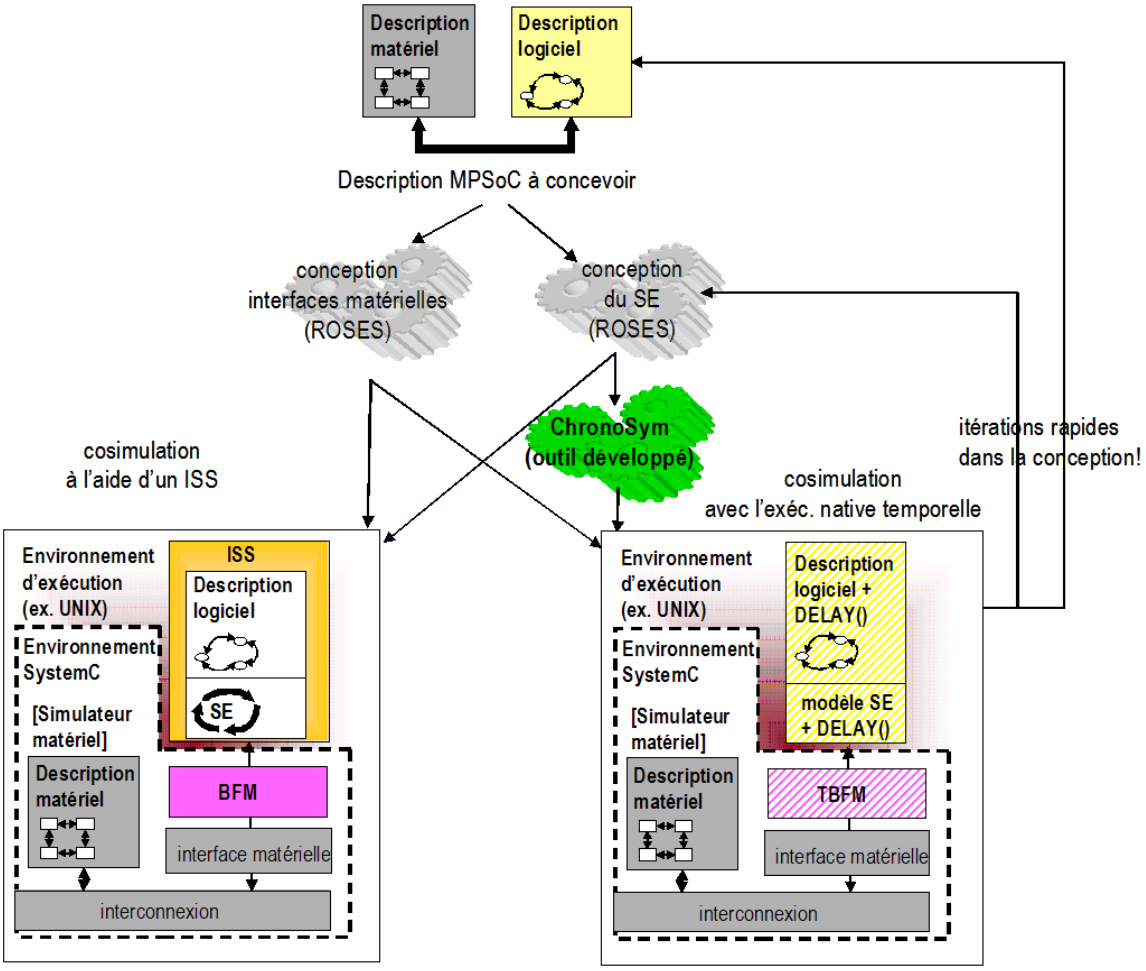


Figure 19. Intégration du ChronoSym dans le flot de conception des MPSoC

La génération automatique de notre modèle d'exécution native et temporelle se fait à partir des bibliothèques déjà existantes pour le SE [Gaut 01]. ChronoSym est appliqué au code du logiciel, après que le SE soit conçu. L'entrée de l'outil ChronoSym est le code du SE et de l'application logicielle. La sortie est le modèle d'exécution natif, temporel ; composé des couches hiérarchiques, divisés en services spécifiques et non-spécifiques au processeur. ChronoSym utilise des approches différentes pour ces deux types de services.

Le modèle d'exécution native pour l'évaluation du logiciel embarqué est fondé sur trois éléments de base, qui seront tous générés par ChronoSym :

- (1) le modèle de simulation du HAL ;
- (2) la fonction **`Delay()`** émulant l'écoulement des temps d'exécution ;
- (3) le module de synchronisation logicielle/matérielle (TBFM).

(1) L'exécution native du code logiciel réel peut être réalisée car le code de l'application et le code du SE non-spécifique au processeur sont exécutés nativement sur le simulateur hôte, puisqu'ils sont portables sur différents processeurs. Cependant, le code du SE spécifique au processeur, c.-à-d. la couche HAL, ne peut pas être exécuté sur le simulateur hôte. Ainsi, il est nécessaire de construire un modèle de simulation pour le HAL et de l'exécuter sur le simulateur hôte. On écrira donc, en langage C, les fonctions nécessaires pour la simulation pour le changement de contexte, le traitement des interruptions, et des entrées/sorties (Chapitre III.3.1).

(2) Pour la simulation temporelle du logiciel embarqué, nous annotons le code du logiciel avec les délais temporels de son exécution. Une fonction nommée **`Delay()`** exprime l'écoulement du temps pour le logiciel. Elle est automatiquement insérée dans le code réel du logiciel non-spécifique au processeur, en utilisant le générateur de SE de ROSES [Gaut 01]. Par contre, l'annotation automatique du HAL est réalisée sur son modèle de simulation (que l'on a développé et introduit dans les bibliothèques de ROSES), en utilisant aussi le générateur de SE de ROSES.

La stratégie utilisée correspond à une estimation temporelle de haut niveau. Elle est basée sur une compilation partielle du code source réalisée par le compilateur C de ARM, « **`armcc`** » (ou par le compilateur C de GNU, « **`gcc`** » en effectuant une compilation croisée[12], avec ARM comme processeur cible). Ainsi, on peut trouver la correspondance entre la description de haut niveau du logiciel et l'assembleur sur le processeur cible. ChronoSym réalise l'analyse automatique du code et la génération automatique du modèle de simulation native temporelle. Ses principales fonctions sont :

- l'analyse du code assembleur et code C d'origine ;

[12] cross-compiling

- le calcul des délais équivalents en conformité avec une base interne de données ;
- l'insertion des fonctions `Delay()` parmi les lignes du code C.

(3) Puis, les simulations du logiciel et du matériel sont synchronisées. À cet effet, une interface de cosimulation avec le rôle de synchronisation, nommée **TBFM,** a été développée. Les interfaces de cosimulation logicielle/matérielle (BFM) de ROSES vont être adaptés pour gérer ce nouveau type de cosimulation : pour interagir directement avec le SE au lieu d'interagir avec l'ISS, et pour prendre en compte les temps d'exécution du logiciel qui sont des fonctions paramétrés avec ce délai. Cette nouvelle interface l'on va appeler TBFM – Timed BFM, pour mettre en évidence la prise en compte de l'aspect temporel.

La stratégie utilisée et ses différents éléments seront présentés dans la section suivante (Chapitre III.3).

En s'appuyant sur le modèle d'exécution native et temporelle, la cosimulation logicielle/matérielle devient rapide et précise. Cela permettra aux concepteurs d'effectuer des itérations rapides dans le flot de conception des MPSoC. Les flèches en arrière de l'étape de cosimulation, se dirigent vers la conception du SE et aux étapes de partitionnement logiciel/matériel. Ainsi, les itérations rapides permettent l'exploration des plusieurs choix de conception dans la conception du SE et les choix de partitionnement logiciel/matériel, tout en respectant les contraintes données de temps de mise sur le marché.

2.3 Les avantages et limitations du modèle de simulation proposé

Le modèle de simulation proposé utilise un format mixte de spécification, basé sur l'état de l'art existant: l'exécution native et l'ISS. Comme tout modèle hybride, il prend les avantages de deux types d'exécution, et essaye d'éliminer leur points faibles.

Comparé aux méthodes conventionnelles d'exécution native (simulation fonctionnelle et simulation basée sur le modèle du SE), l'approche présentée simule le SE réel et les interactions détaillées logicielles/matérielles. Ainsi, la méthode rapporte une meilleure précision, tout en exploitant l'avantage de vitesse de l'exécution native.

En plus, le modèle d'exécution native est annoté avec les temps d'exécution, à la précision du niveau ISA. Les annotations correspondent aux délais du code assembleur qui va s'exécuter sur la machine cible: c'est la précision de l'ISS. Le modèle inclut aussi une interface d'adaptation et synchronisation, décrite au niveau RTL (comme dans le cas de cosimulation avec un ISS).

Les limitations de cette approche sont liées aux limitations de l'annotation temporelle. Pour un système avec une architecture complexe, les méthodes classiques d'estimation ne sont pas adéquates. C'est le cas où on considère les mémoires cache, le mécanisme de mémoire virtuelle. Aussi, la difficulté apparaît pour des architectures super-scalaires, des architectures VLIW[13] qui considèrent le parallélisme au niveau instructions/données. Dans ces cas difficiles à analyser, des approches basées sur des simulateurs de cache, etc. [Laj 00] [Logh 04] [Mah 05] ou des statistiques sont souvent employées [Bju 01] [Yal 05].

3 Détails de développement ChronoSym

Cette section présente les trois éléments de base de notre modèle d'exécution native temporelle : le modèle de simulation pour la couche d'abstraction du matériel, la fonction d'annotation **`Delay()`** et l'interface de cosimulation représentée par le module TBFM. Chacun de ces trois éléments est détaillé dans une des trois prochaines sous-sections.

3.1 Modèle de simulation pour la couche d'abstraction du matériel (HAL)

Le modèle de simulation pour la couche d'abstraction du matériel (HAL) émule l'API du HAL. Dans cette section, on explique comment établir le modèle de simulation pour chaque type d'API du HAL : le changement de contexte, les entrées/sorties, et le traitement des exceptions.

Dans nos expérimentations on va utiliser le système d'exploitation spécifique à l'application généré des bibliothèques de ROSES [Gaut 01]. Les codes présentés sont spécifiques à ce système d'exploitation, mais la méthode peut être appliqué pour n'importe quel autre système d'exploitation. Les portions de code que l'on a présentés

[13] Very-Long Instruction Word

sont des exemples utilisés pour une meilleure compréhension des concepts. Ils sont loin de réaliser une description complète.

3.1.1 Le changement de contexte des tâches

Les Figure 20 (a) et (b) montrent un exemple de code pour le changement de contexte décrit en assembleur pour le processeur ARM7 et respectivement son modèle de simulation sur UNIX.

Dans cet exemple, le contexte des tâches et les opérations de changement de contexte sont modélisées en utilisant la bibliothèque « multi-threading » de UNIX. En particulier on utilise la structure de données **`ucontext_t`** (pour **`ucp_list[]`** dans la Figure 20 (b)) et les fonctions **`get/setcontext()`**.

```
__cxt_switch            ;r0 = ancien indicateur de pile,
                        ;r1 = nouveau indicateur de la pile

  STMIA r0!,{r0-r14}    ;sauvegarde les registres de la tâche courante
  LDMIA r1!,{r0-r14}    ;reconstitue les registres de la nouvelle
tâche
  SUB pc, lr, #0        ;retour
  END
```

(a)

```
   void context_sw(int cur, int new)
   {
    Delay(34);
    getcontext(ucp_list[cur]);  // UNIX multi-threading lib.
    setcontext(ucp_list[new]); // UNIX multi-threading lib
    Delay(3);
   }
```

(b)

Figure 20. Le code du changement de contexte et son modèle de simulation de HAL : (a) code assembleur pour le processeur ARM7 ; (b) modèle de simulation UNIX

Pour être plus spécifique, la fonction **`getcontext()`** émule dans le modèle de simulation, l'instruction « **`STMIA`** » (Figure 20 (a)), qui sauvegarde les valeurs des registres du processeur ARM7 dans la pile de la tâche courante. Le contexte courant de l'exécution pour la simulation logicielle est sauvegardé dans une structure de données, nommée **`ucp_list[cur]`**.

La fonction **`setcontext()`** émule l'instruction « **`LDMIA`** » du changement de contexte original de la simulation logicielle, tout en rétablissant le contexte sauvegardé de l'exécution de la simulation logicielle. Le délai inséré au début du

modèle du HAL correspond aux deux premières instructions assembleur du code réel du HAL. Le délai introduit à la fin correspond au retour de la fonction (et au changement de contexte dû à ce fait).

3.1.2 Les fonctions d'entrée/sortie

Les fonctions d'entrée/sortie représentent des opérations de lecture/écriture de/dans la mémoire. Pour modéliser les fonctions d'entrée/sortie, nous employons le modèle fonctionnel de bus (BFM) conventionnel, qui est inclus dans le TBFM.

Le BFM transforme les accès mémoire dans des événements sur les signaux de l'interface du processeur : par exemple il sélectionne le bus d'adresses/donnés ou les signaux de contrôle.

3.1.3 La fonction de traitement des exceptions du processeur

Les processeurs peuvent disposer de plusieurs types d'exceptions. Par exemple, le processeur ARM7 a sept types exceptions différentes: (1) « **reset** », (2) « **undefined instruction** »[14], (3) interruption logicielle (« **softwareI** »[15]), (4) « **prefetch abort** »[16], (5) « **data abort** »[17], (6) **IRQ**[18], et (7) **FIQ**[19] [ARM7].

Figure 21. Établir le modèle temporel de simulation pour l'API du HAL

```
//code assembleur originel
//pour la routine de traitement d'interruption
(softwareI)
  softwareI_Routine
  STMIA r13,{r0-r14}^   ;Sauvegarde reg. utilisateur
  MRS r0,sprs           ;Obtient SPSR
  STMDB r13!,{r0,lr}    ;Sauv. SPSR et LR_SVC
  LDR r0,[lr,#-4]       ;Charge l'instr. softwareI
  BIC r0,r0,#0xff000000
    BL __trap_trap
```

```
//modèle de sim. du HAL
   softwareI_Enter(){
      CPSR_save = CPSR;
      SPSR_save = SPSR;
      CPSR = SVC;
   }
```

[14] instruction non définie
[15] software interrupt
[16] arrêt de recherche de l'instruction
[17] arrêt de transaction
[18] Interrupt Request
[19] Fast Interrupt Request

```
  LDMIA r13!,{r0,lr} ;Restaure l'addr de retour,spsr
..MSR spsr_cf,r0.....;Restaure spsr pour softwareI
..LDMIA r13,{r0-r14}^;Restaure registres
                     ;et retour en mode utilisateur
  NOP
  MOVS pc,lr         ;Retour de softwareI
```

```
//modèle de sim. du HAL
   softwareI_Return(){
      CPSR = CPSR_save;
      SPSR = SPSR_save;
   }
```

(a)

```
//code assembleur originel pour
//l'utilisation du softwareI
__swi(0) void __trap_trap(int, int, int);
__trap_trap(0, id, 0);
```

```
//modèle de simulation
//temporel pour
//l'API du HAL
 softwareI_Enter();
Delay(24);
_trap_trap(0, id, 0);
 softwareI_Return();
 Delay(23);
```

(b)

Pour établir le modèle de simulation du HAL, il n'est pas exigé de modéliser toutes les exceptions. Par exemple, l'exception de « l'instruction non définie » n'est pas liée à l'émulation de l'API du HAL. Dans le cas du processeur ARM, cinq exceptions sont connexes à l'émulation de l'API du HAL : **`softwareI`**, **« `prefetch abort` »**, **« `data abort` »**, **`IRQ`**, **`et FIQ`**.

La Figure 21 montre l'exemple de modélisation pour la fonction de traitement de l'interruption logicielle[20] de l'API du HAL, pour le processeur ARM7. Les codes exprimés à gauche des Figure 21 (a) et (b) représentent le traitement de l'interruption logicielle (**`softwareI`**).

Le code de la fonction **`_softwareI_Routine`** est décrit en assembleur (Figure 21 (a)). Le code en C appelle une fonction de haut niveau utilisée pour le traitement des interruptions logicielles, nommé **`__trap_trap()`**, avec le numéro de l'interruption **`0`** (défini par **`__swi(0)`**[21]).

Quand la fonction **`__trap_trap()`** est appelée (Figure 21 (b)), l'exécution va directement à l'élément de la table des vecteurs de **`softwareI`**. Ensuite, le traitement de **`softwareI`**, **`_softwareI_Routine`** (décrit en Figure 21 (a)) est exécuté.

[20] SWI handler

[21] Dans le cas du processeur ARM, la directive __swi() est employée quand une fonction est déclarée comme routine de traitement des interruptions logicielles.

Pour modéliser de tels traitements des exceptions, la stratégie employée est de modéliser l'ensemble minimal des états du processeur et les opérations correspondantes. Dans le cas du processeur ARM, l'ensemble minimal des états du processeur est constitué des registres (**CPSR**[22] et **SPSR**[23]) qui contiennent l'information de l'état de processeur (**ARM** ou **THUMB**), la désactivation des interruptions[24] **IRQ** et **FIQ**, et le mode du processeur parmi sept modes possibles [ARM7]. Les registres contenant le mode du processeur doivent être simulés pour identifier le statut de l'interruption courante et le mode courant de processeur. Cela peut indiquer si une interruption peut être traitée ou pas, ou si un accès mémoire cause une violation de protection ou pas.

Le côté droit de la Figure 21 (a) montre la modélisation de la fonction de traitement de l'interruption logicielle (**softwareI**). Deux fonctions **softwareI_Enter()** et **softwareI_Return()** modélisent les opérations d'entrée et de retour de la routine **softwareI**. Dans les deux fonctions, seulement les opérations liées aux registres de mode (**CPSR** et **SPSRs**) sont simulées.

Par exemple, pour simuler l'opération de la fonction de traitement de l'interruption pour sauvegarder **SPSR** dans de la tâche courante et de reconstituer la pile, on modélise la pile de la tâche en étant le vecteur **stack[cur][mode]** et en introduisant/soustrayant le **SPSR** de cette pile. Le registre contenant le statut courant du processus (**CPSR**) du nouveau mode du processeur est simulé tout en plaçant **CPSR** en mode superviseur (**SVC**[25]).

Le côté droit de la Figure 21 (b) présente que les deux fonctions sont ajoutées avant et après l'appel de « **softwareI** ». Les autres traitements d'exceptions : « **prefetch abort** », « **data abort** », **FIQ**, et **IRQ** sont modélisés de la même manière.

[22] Current Process Status Register
[23] Saved Process Status Register
[24] interrupt disables
[25] supervisor mode

3.2 Simulation des temps d'exécution du logiciel embarqué, en tenant compte des interruptions du matériel

3.2.1 Annotation du code logiciel avec les temps d'exécution du matériel

Le flot d'annotation pour le code non-spécifique au processeur avec les temps d'exécution est décrit ensuite (Figure 22). L'entrée est le code source décrit en langage C. La sortie est le même code annoté avec les temps d'exécution sur un processeur cible. L'outil peut insérer des annotations au niveau de chaque instruction assembleur, instruction C ou au niveau de chaque fonction, dépendant de la granularité souhaitée.

La première étape est un prétraitement : le code source décrit en C est compilé avec un compilateur pour le processeur cible ; par exemple **`armcc`** ou **`gcc`** avec ARM7 comme processeur cible. Puis, la correspondance entre le code source et le code assembleur est trouvée. Cela est fait en utilisant les possibilités du compilateur de générer un fichier intermédiaire avec les instructions C et assembleur intercalées. Une autre possibilité est d'introduire des marqueurs dans le code sous la forme des commentaires qui permet une localisation facile. Dans notre approche on utilise les deux solutions.

La deuxième étape est « le calcul et l'annotation avec les délais d'exécution ». Le temps d'exécution de chaque instruction d'assembleur est calculé en utilisant la fiche technique du processeur cible, fournie par le fournisseur de processeur [ARM7].

Cette étape est générique : elle peut être appliquée pour différentes applications (décrites en langage C) et pour différents processeurs cible. La bibliothèque contenant les informations de la fiche technique doit être construite pour chaque processeur cible. Elle contient les mnémoniques des instructions en assembleur, et leur temps d'exécution en fonction des paramètres qui sont leur les opérandes et le contexte d'exécution.

La deuxième étape se base sur un outil qui réalise la correspondance entre le code C et assembleur. Cet outil a été développé avec **`Lex`** et **`Yacc`** [LexYacc]. Le code C est gardé intact alors que les lignes du code assembleur sont remplacées, chacune avec une fonction **`Delay()`** qui a comme argument le temps d'exécution de l'instruction assembleur qu'il remplace.

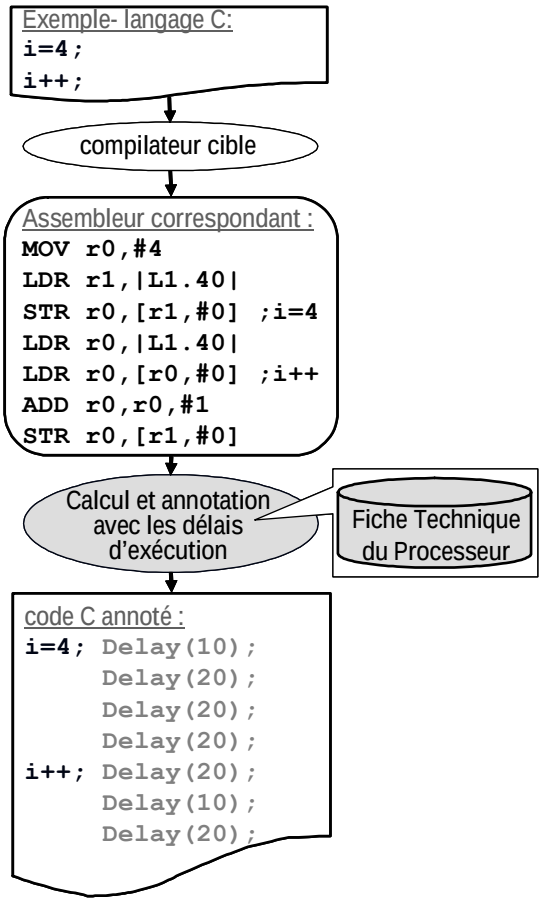


Figure 22. Annotation automatique avec les délais d'exécution

Par exemple, dans la Figure 22, la ligne de code source, **`i=4`** correspond aux instructions assembleur : **`MOV`**, deux **`LDR`** et un **`STR`**. Des fonctions **`Delay()`** sont annotées à la ligne correspondante du code source.

Le modèle de simulation de HAL est annoté avec des délais temporels, de la même manière que le code non-spécifique au processeur. Les valeurs des délais proviennent du code réel du HAL, souvent décrit en assembleur, ainsi conférant une bonne précision temporelle au modèle. La description fonctionnelle est remplacée avec le modèle développé en ce but, comme il est décrit en Chapitre III.3.1.

Dans l'exemple de la Figure 22, pour chaque instruction assembleur, une fonction **`Delay()`** est utilisée. Pour l'efficacité de la vitesse de la simulation, des fonctions **`Delay()`** (correspondantes aux lignes du code source) pourraient être fusionnées.

Cette stratégie fait partie d'un compromis entre la vitesse de simulation et la précision temporelle.

L'exactitude de l'évaluation des temps d'exécution est cruciale pour l'exactitude de la cosimulation de logicielle/matérielle avec les modèles d'exécution natifs temporels et donc pour l'évaluation du système en ensemble.

3.2.2 La fonction Delay()

Le prototype de la fonction **`Delay()`** est décrit en langage C. Mais elle est vraiment exécutée en collaboration avec SystemC, par l'intermédiaire du bus de cosimulation (TBFM). En association avec le TBFM, elle peut simuler l'avancement du temps pour le sous-système logiciel et les interruptions du processeur.

Le mode de fonctionnement du TBFM sera présenté dans la section suivante (Chapitre III.3.3). Les détails de la collaboration entre la fonction **`Delay()`** et le TBFM seront montrés en cette section.

En assurant l'interaction avec le matériel, la fonction **`Delay()`** accomplit deux rôles, (Figure 23) :

- (1) la synchronisation des temps du logiciel avec le matériel ;
- (2) la simulation de l'interruption de processeur.

On suppose que la fonction **`Delay(10)`** est appelée dans le code annoté du logiciel, comme il est représenté dans la Figure 22.

La Figure 23 (a) montre le cas quand il n'y a aucune interruption du processeur pendant la période de **`10 unités de temps`**. Dans ce cas, la fonction **`Delay(10)`** retourne après la période entière de **`10 unités de temps`**. Le temps de simulation du logiciel est avancé par **`10 unités de temps`**, comme résultat d'exécution de la fonction **`Delay(10)`**.

La Figure 23 (b) montre le cas où il y a une interruption de processeur au **`temps 5`** pendant la période des **`10 unités de temps`**. Dans la vraie exécution, quand l'interruption arrive, l'exécution de la tâche courante est interrompue et la routine du traitement d'interruption du processeur est appelée. Pour simuler ceci, la fonction **`Delay()`** appelle le modèle de simulation de la fonction de traitement de

l'interruption (modèle de traitement de l'interruption de **IRQ** ou **FIQ** comme décrit dans la Figure 21).

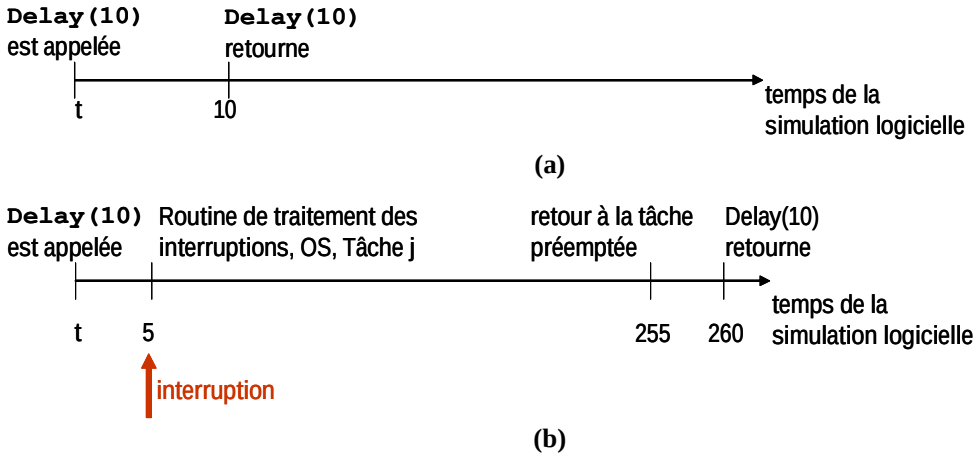


Figure 23. Avancement du temps de simulation : (a) Cas d'exécution sans interruption ; (b) Cas d'exécution avec interruption.

La fonction de traitement de l'interruption peut même appeler des fonctions du SE (par exemple « **lock release** ») pour permettre le changement de contexte des tâches. La Figure 23 (b) illustre que le temps d'exécution du traitement de l'interruption pour la fonction(s) du SE et de tout autre tâche(s) que celui interrompu, prend **250 unités de temps**. Après que le traitement de l'interruption (et/ou l'exécution d'autres tâches) soit fini, l'exécution de la tâche originale qui a appelé la fonction **Delay(10)** reprend au **temps 255** comme représenté sur la figure. Dans ce cas-ci, après la période du délai restant, c.-à-d. **5 unités de temps**, la fonction **Delay(10)** finit.

Les interactions entre la fonction **Delay()** et le module TBFM sont analysés ensuite.

Quand la fonction **Delay(d)** est appelée, elle envoie la valeur du délai **d** au TBFM (Figure 24). Le module TBFM avance le temps de simulation du logiciel en attendant des événements sur des signaux d'interruption du processeur (par exemple les signaux **nIRQ**, **nFIQ** du processeur ARM7).

Dans le cas où il n'y a aucune interruption du processeur pendant la période de temps **d**, le module TBFM avance le temps courant de simulation logicielle, par le temps **d**.

Le retour à la fonction **`Delay()`** fournit le temps courant de simulation du logiciel et le statut d'interruption du processeur (aucune interruption, dans ce cas-ci).

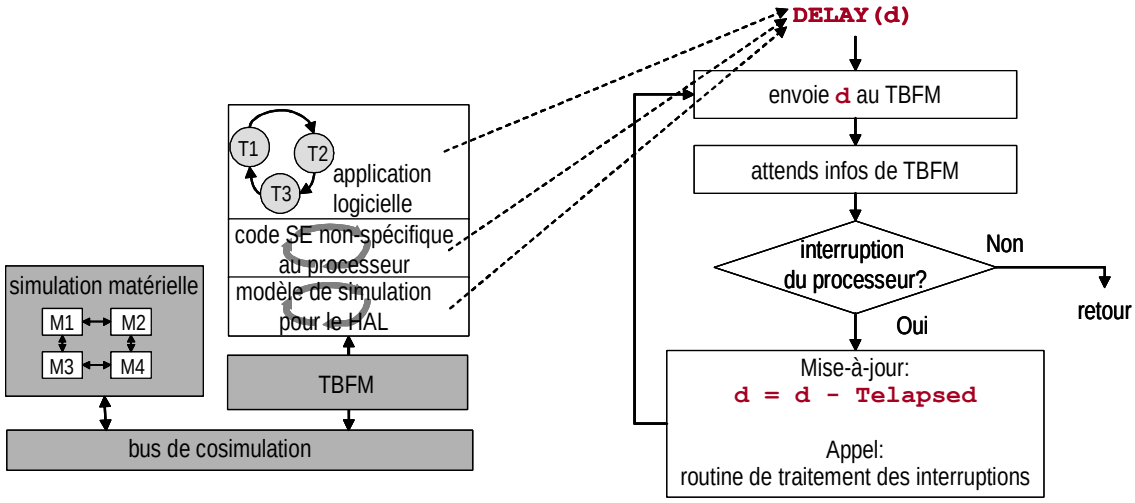


Figure 24. La collaboration entre la fonction Delay() et le TBFM

Dans le cas où une interruption du processeur arrive pendant la **`période de temps d`**, le module TBFM cesse d'avancer le temps de simulation du logiciel au moment quand l'interruption est arrivée. Puis, il envoie à la fonction **`Delay()`** la période courante de simulation du logiciel et le statut d'interruption de processeur : interruption déclenchée dans ce cas-ci.

3.3 TBFM (Timed Bus Functional Module)

On a montré que pour synchroniser l'exécution native et la simulation matérielle on a développé une interface de cosimulation nommée TBFM. Elle est basée sur le module BFM conventionnel (Chapitre I.2), et rajoute des informations liés aux annotations temporelles présentes dans le modèle du logiciel. Cette section montre le mode de fonctionnement de cette interface.

Il y a deux types d'interactions entre le logiciel et le matériel. Elles sont gérées de deux manières différentes.

- ***(1) Les interactions du logiciel vers le matériel***, c.-à-d. les accès mémoire. Elles sont gérées par le BFM conventionnel. Ce fonctionnement conventionnel est inclus dans le TBFM.
- ***(2) Les interactions du matériel vers le logiciel***, c.-à-d. les interruptions du processeur. Elles sont gérées dans la simulation

temporelle du logiciel. Dans ce but, le TBFM contient le BFM conventionnel et une fonction pour la synchronisation logicielle/matérielle, requise dans la simulation temporelle du logiciel.

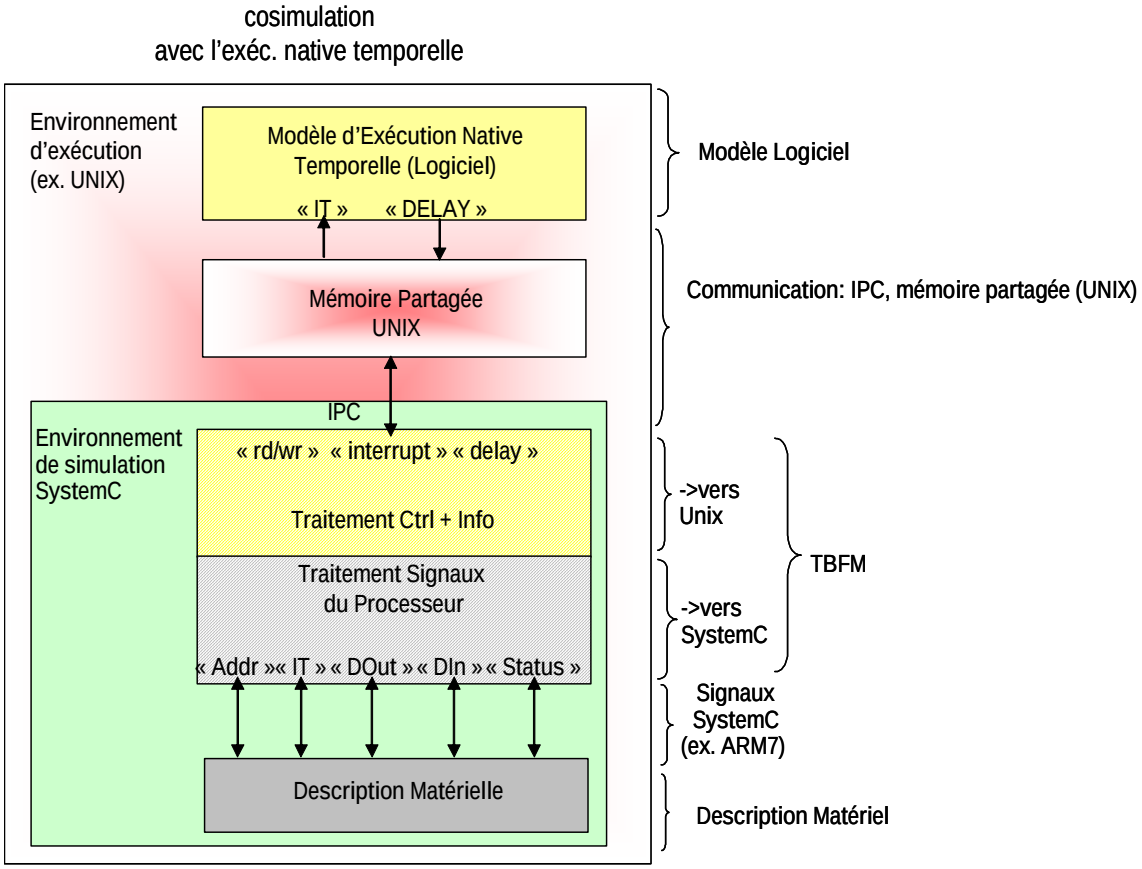


Figure 25. La réalisation du TBFM

La structure d'un TBFM a deux composantes :

- (a) une partie orientée vers la simulation native du SE (dans notre cas, un processus UNIX)
- (b) une partie orientée vers l'interface au niveau pin/signal du processeur.

Ensuite, on va expliquer les fonctionnalités deux parties par rapport à la gestion des interactions entre le logiciel et le matériel.

Pour communiquer, le modèle d'exécution native du logiciel utilise une mémoire partagée, des signaux et des sémaphores (mécanismes IPC UNIX). Il met toutes les données à transmettre dans une mémoire partagée, et signale la présence des nouvelles données par des jeux de sémaphores et signaux.

Ensuite, c'est le TBFM qui transfère des accès externes à partir de l'exécution native du logiciel vers la simulation du matériel (par exemple une simulation SystemC). Il transforme un accès externe du logiciel (par exemple lecture/écriture), en transitions de signal sur l'interface du processeur au niveau pin/signal.

Le mécanisme de communication du matériel vers le logiciel se comporte de la même manière, par l'intermédiaire du TBFM qui transfère les signaux du processeur en données dans une mémoire partagée. Le TBFM transfère également les interruptions du processeur au SE. Dans ce cas, si l'interruption arrive sur le pin(s) d'interruption du processeur, le TBFM envoie un signal (signal Unix) au SE (c.-à-d. au processus Unix).

Le principe d'exécution est représenté en pseudo-code. Après commuter entre les simulations logicielle et matérielle (« **softwareITCH_Simulateurs()** »), le TBFM met à jour les signaux du processeur « **MiseAJour_Signaux_Processeur()** » avec les nouvelles données de la mémoire partagée. La fonction « **wait()** » signifie qu'il laisse passer un cycle de simulation. Cette simulation mettra à jour les signaux du processeur. Ensuite, le TBFM met à jour la mémoire partagée, avec les nouvelles valeurs lus sur les signaux (« **MiseAJour_SHM()** ») et avance le temps de simulation avec le délai reçu du logiciel (« **Execution_Delay()** »).

```
while (true) {
        softwareITCH_Simulateurs();
        MiseAJour_Signaux_Processeur();
        wait();
        MiseAJour_SHM();
        Execution_Delay();
}
```

4 Expérimentations

Cette section présente l'application de notre méthode d'évaluation des temps d'exécution pour le logiciel embarqué sur deux exemples : l'application synthétique McDrive et le modem VDSL. Après la description de ces deux exemples (en la première sous-section), on présente (dans la deuxième sous-section) la génération automatique du modèle de simulation pour le logiciel embarqué. Les résultats des expérimentations sont présentés et analysés (dans la troisième sous-section) ; pour pouvoir faire des comparaisons, on réalise des cosimulations avec le logiciel simulé sur un ISS, ensuite exécuté nativement en utilisant un modèle de SE, et finalement par notre méthode proposée (simulation native, temporelle).

4.1 Spécification des applications – les systèmes VDSL et McDrive

Pour illustrer les concepts présentés dans ce chapitre, nous avons étudié notre propre exemple appelé McDrive et un exemple industriel nommé VDSL [Dia 01].Nous avons conçu dans le groupe SLS les systèmes McDrive et VDSL sur des architectures MPSoC.

L'application McDrive est un benchmark synthétique, illustrative pour un système embarqué monopuce. L'exemple est un distributeur automatique utilisé dans un restaurant McDrive, basé sur le modèle producteur-consommateur. Il décrit naturellement un système électronique qui ne communique pas beaucoup avec l'environnement, mais qui a une activité interne intensive (semblable aux SoCs). Les tâches et les modules de l'intérieur du système, échangent un bon nombre de données.

McDrive est une application « synthétique » qui contient en total environ 2000 lignes de code. Son architecture inclut un processeur ARM7 et trois composants matériels (IPs) – qui sont des automates d'environ 10 états. Sur le processeur, il y a deux tâches principales (à part la tâche de mise en veille), qui s'appellent « **`cooker`** » et « **`vendor`** ». Les autres modules émulent un comportement de « **`client`** » et de « f**`ournisseur`** ». Nous avons aussi un module Additionnel de « **`reset`** » qui ne s'exécute qu'au démarrage du système. La communication est réalisée par des interconnexions point-à-point.

VDSL (Very-high-data-rate DSL) est un protocole de communication utilisant les lignes téléphoniques qui fait partie des techniques xDSL (Digital Subscriber Line). Il est encore au stade de prototype, et de nombreuses entreprises proposent leur propre version du protocole VDSL.

L'application que nous avons utilisée pour nos expérimentations est un modem spécifique au VDSL. Sa taille est d'environ 5000 lignes de code. Le point de départ a été la réalisation du modem VDSL en utilisant des composants discrets, où on a remplacé la partie de traitement du signal fixe (non-configurable), le contrôle du modem et l'interfaçage avec le PC hôte. Nous avons reparti cette fonctionnalité sur deux processeurs ARM7 et un bloc matériel réalisant la chaîne de transmission. Ce partitionnement nous a été suggéré par l'équipe qui a réalisé le prototype du modem VDSL [Dia 01].

Nous avons d'abord décrit ces applications en langage VADeL, à l'aide des modules virtuels, canaux virtuels et ports virtuels. Ensuite, nous avons utilisé l'outil de conception de systèmes sur puce, ROSES, [Ces 02] pour générer la réalisation RTL de l'application ciblée sur l'architecture fixée pour chaque exemple.

Le flot de conception sur l'application VDSL est illustré en Figure 26. Comme représenté sur la figure, trois tâches logicielles sont mappés sur un processeur ARM7 et six autres tâches sur un autre processeur ARM7. Les tâches communiquent par l'intermédiaire des primitives du SE tels que « **`pipe`** », « **`signal`** », et mémoire partagée (« **`shm`** »[26]). Ces primitives sont spécifiques au SE que l'on a utilisé pour nos expérimentations [Gaut 01].

Pour être spécifique, l'outil ROSES génère les SEs pour deux processeurs ARM7 et les parties logiques pour pouvoir interconnecter les sous-systèmes logiciels avec les sous-systèmes matériels. Ces sous-systèmes peuvent être maintenant interconnectés pour la réalisation RTL complète.

L'outil ROSES donne également des modèles de cosimulation logicielle/matérielle basés sur l'ISS et sur les modèles de SE comme représenté sur la Figure 26. Le

[26] shared memory

modèle de cosimulation logicielle/matérielle avec le modèle d'exécution native temporelle est généré avec ChronoSym, et il sera détaillé ensuite.

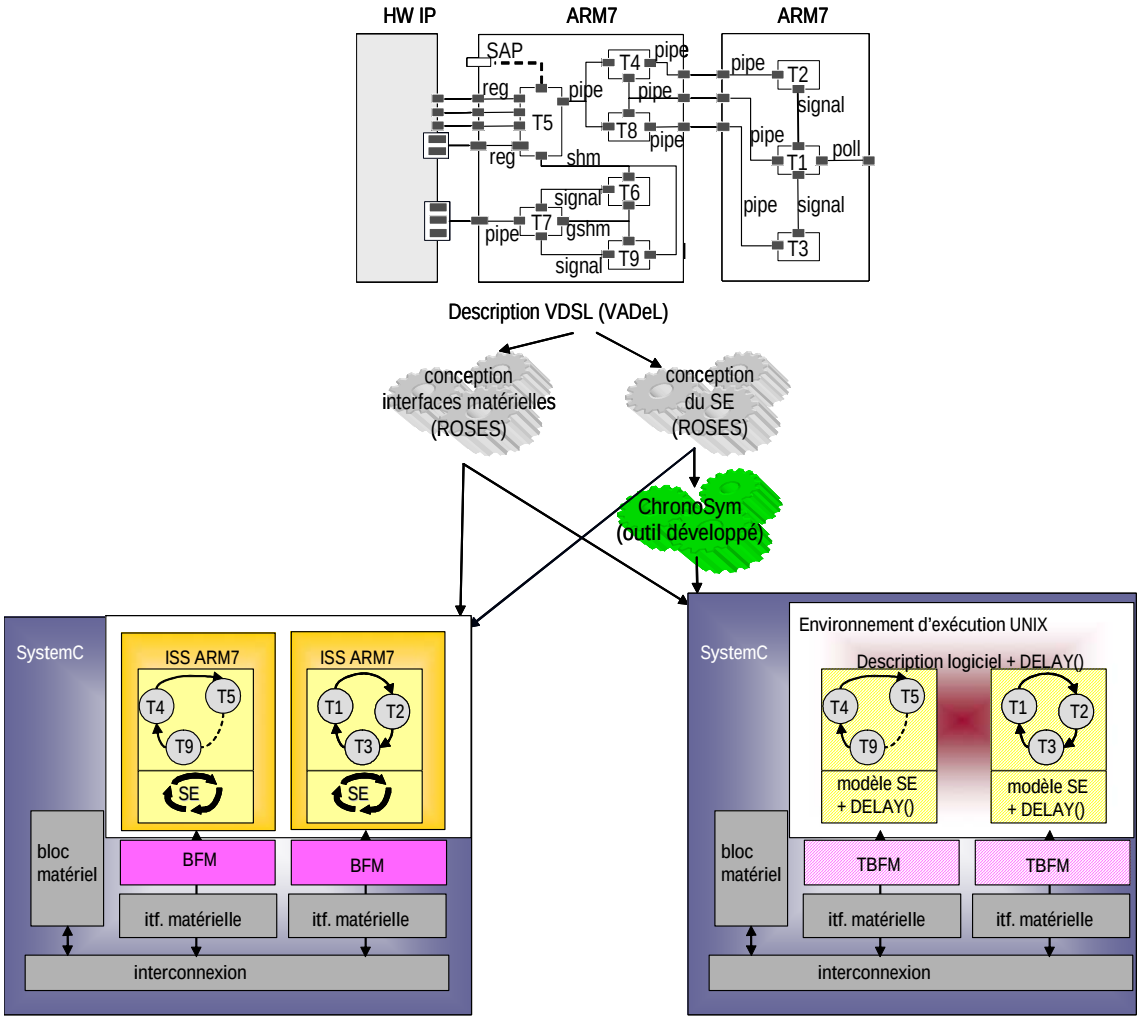


Figure 26. Design et modèles de cosimulation pour le système VDSL : (a) l'application VDSL mappée sur une architecture ; et différents types de simulation logicielle : (b) basée sur l'ISS ; (c) basée sur le modèle du SE ; (d) basée sur l'exécution native, temporelle (proposée)

4.2 Génération automatique du modèle de simulation

La réalisation du modèle d'exécution native est une étape concurrente au raffinement de l'architecture. Pour le processeur on obtient, à côté de l'interface matérielle de communication, le modèle de simulation pour le SE et pour l'application. L'architecture cible pour VDSL et le modèle d'exécution native pour le SE sont présentés dans la Figure 27.

Le sous-système logiciel composé de l'application et du SE spécifique correspondant constitue l'entrée de l'outil ChronoSym. Les deux parties composantes, spécifique et non-spécifique au processeur, sont traitées différemment par l'outil.

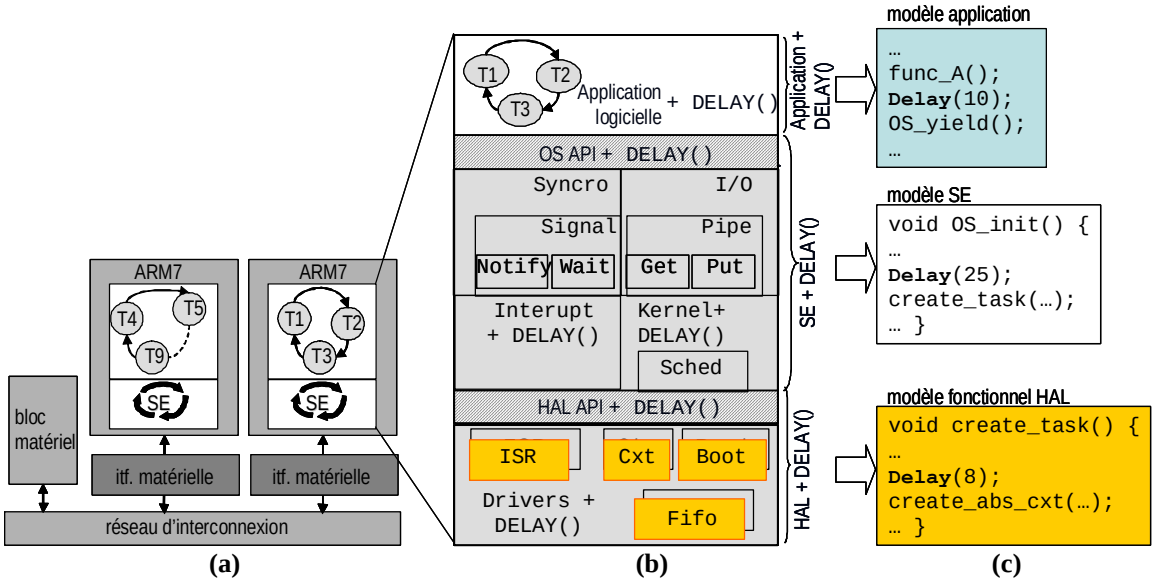


Figure 27. L'architecture cible pour VDSL et le modèle d'exécution native du SE: (a) l'architecture cible ; (b) le modèle d'exécution native du SE ; (c) exemples du code SE natif, annoté

Le grand bénéfice de l'architecture hiérarchique du SE est que seulement le HAL est spécifique à l'architecture. Les modifications pour la modélisation de l'exécution native sous l'environnement UNIX interviennent seulement dans les services du HAL : les pilotes des entrées/sorties, la routine du traitement d'interruption. Des exemples du code utilisé sont mis en évidence dans le SE détaillé dans la Figure 27 (b). Des détails sur la signification de ces services peuvent être trouvés en [Gaut 01].

L'outil génère automatiquement le modèle de simulation pour l'application et pour le SE, tout en gardant le squelette du logiciel et sa partie fonctionnelle (le code C du SE et de l'application). Un exemple schématique est présenté dans la Figure 27 (c).

La fonction **`Delay()`** a été automatiquement introduite dans le code du logiciel. Une annotation manuelle serait fastidieuse et peu fiable. Les bénéfices apportés par l'outil d'annotation automatique sont évidents. En insérant la fonction **`Delay()`** au niveau de chaque instruction assembleur, le code annoté de l'application (et respectivement du SE) augmente avec jusqu'au **`100%`** lignes de code C pour l'application et pour le SE :

par exemple pour McDrive que le code annoté (sans « headers ») du SE augmente de 1000 lignes à 2000 lignes, et pour VDSL il augmente de 3000 lignes à 5000 lignes de code C.

4.3 Résultats

Nous avons exécuté les trois cas de cosimulation pour les systèmes McDrive et VDSL. Dans toutes les situations, la simulation matérielle s'exécute au niveau RTL, en SystemC. Tous les modèles de simulation pour le sous-système logiciel (ISSs, modèles du SE, et modèles natif annoté avec les temps d'exécution) fonctionnent sur des processus UNIX séparés. Entre la simulation matérielle et logicielle, nous avons utilisé une mémoire partagée UNIX pour la communication interprocessus.

Table 6. Comparaison entre les temps de simulation el les précisions, pour l'application McDrive

	COSIMULATION A L'AIDE DE :		
	ISS	**MODELE SE**	**MODELE D'EXECUTION NATIVE SE**
TEMPS DE SIMULATION (ACCELERATION)	6.5 sec.	0.5 sec. (13 fois)	**0.6 sec. (10.8 fois)**
NO. DE CYCLES SIMULES	73,800 cycles	50,800 cycles	**61,100 cycles**

Table 7. Comparaison entre les temps de simulation el les précisions, pour l'application VDSL

	COSIMULATION A L'AIDE DE :		
	ISS	**MODELE SE**	**MODELE D'EXECUTION NATIVE SE**
TEMPS DE SIMULATION (ACCELERATION)	32 sec.	12 sec. (2.7 fois)	**15 sec. (2.1 fois)**
NO. DE CYCLES SIMULES	1,486,000 cycles	1,026,400 cycles	**1,227,600 cycles**

Les Table 6 et Table 7 donnent les temps d'exécution et la précision de trois types de cosimulation pour les systèmes McDrive et respectivement VDSL.

En termes de vitesse de simulation, comparée à la cosimulation basée sur l'ISS, le modèle de cosimulation basée sur le SE et la méthode proposée donnent une

accélération de 13 et 10.8 fois, respectivement dans le cas du système de McDrive. Ces méthodes donnent des accélérations de 2.7 et respectivement 2.1 fois, comparé à la cosimulation basée sur l'ISS, dans le cas du système VDSL.

Ensuite on va expliquer les différences en termes de vitesse et précision.

Il faut spécifier que, dans les deux cas des systèmes McDrive et VDSL, les simulations natives (cosimulation basée sur le modèle du SE et la méthode présentée) donnent des accélérations significatives de simulation. Cependant, elles n'atteignent pas l'accélération théorique maximale (**100-1000 fois**) expliquée dans la Figure 14 de Chapitre III.1.2.

Les raisons sont que (1) le modèle de simulation du matériel précis au niveau cycle (RTL) est fixé dans tous les trois types de cosimulation et (2) la simulation logicielle et matérielle synchronisent l'un avec l'autre par l'intermédiaire des IPC[27] de UNIX, qui est très coûteuse en temps d'accès.

Dans du le cas du système McDrive, la simulation matérielle prend **48,9%** du temps total de simulation et la synchronisation prend **31,8%.** Dans le cas du système VDSL, la simulation matérielle et la synchronisation prennent **71,2%** et **21,5%,** respectivement. Nous prévoyons que l'accélération de la simulation augmentera quand la simulation du matériel sera effectuée à des niveaux plus élevés d'abstraction.

En termes de précision de la synchronisation, nous supposons que la cosimulation basée sur l'ISS ne donne aucune erreur. En conséquence, les erreurs de synchronisation pour les simulations à base de modèle de simulation du SE (noté « $\mathbf{erreur_{ISS\text{-}SE\_natif}}$ ») et à base de modèle de simulation du SE natif et temporel respectivement (noté « $\mathbf{erreur_{ISS\text{-}SE\_natif\_temp}}$ »), sont calculée comme suit:

$$erreur_{ISS-SE\_natif} = \frac{\left\| nbCycles_{ISS} - nbCycles_{SE\_natif} \right\|}{nbCycles_{ISS}} * 100\% \qquad (1)$$

$$erreur_{ISS-SE\_natif\_temp} = \frac{\left\| nbCycles_{ISS} - nbCycles_{SE\_natif\_temp} \right\|}{nbCycles_{ISS}} * 100\% \qquad (2)$$

[27] Inter-Process Communication

où **nbCyclesISS** est le nombre de cycles de simulation obtenus pour la cosimulation basée sur l'ISS, **nbCyclesSE** est le nombre de cycles de simulation obtenus pour la cosimulation avec le modèle du SE, ou respectivement avec le modèle d'exécution natif temporel.

Dans la cosimulation avec le modèles d'exécution natif temporel, on a obtenu une erreur de **17,2%** dans le cas du système McDrive et de **17,4%** dans le cas du système VDSL, par rapport à la cosimulation basée sur l'ISS. Cette erreur est beaucoup plus petite que l'erreur de la cosimulation basée sur le modèle du SE, de **31,4%** dans le cas du système McDrive et de **30,9%** dans le cas de système VDSL, respectivement.

Cette amélioration de la précision résulte de la simulation du SE réel et des interactions logicielles/matérielles dans le modèle d'exécution native, temporel. Tandis que la cosimulation basée sur le modèle du SE émule seulement l'API du SE, et pas son exécution réelle.

Comparée à la cosimulation basée sur l'ISS, la source d'erreur dans le modèle d'exécution natif et temporel consiste principalement dans l'évaluation des délais d'exécution. Dans notre expérimentation, nous prenons une méthode simple d'évaluation des délais, expliquée dans Chapitre III.3.2. Si l'erreur d'évaluation des délais est réduite par des techniques d'évaluation avancés, comme ceux présentés en [VCC] [Laj 99], nous nous attendons à l'amélioration davantage de la précision de la cosimulation.

Perspectives

L'objectif majeur de notre travail était de concevoir un environnement global et flexible pour l'évaluation des performances des MPSoC. Cet environnement doit être efficace et précis pour pouvoir intervenir dans les étapes globales d'optimisation, comme le partitionnement logiciel/matériel ou la sélection des composants, mais aussi dans les étapes locales d'optimisation, comme la modification des paramètres pour un réseau d'interconnexion ou du système d'exploitation embarqué.

Les expérimentations présentées dans le Chapitre IV concernent l'exploration des sous-systèmes d'interconnexion. En complément, la méthode présentée en ce Chapitre peut être utilisée pour l'exploration de l'architecture du SE. Cette possibilité est non

explorée jusqu'à présent, et elle peut offrir une bonne efficacité pour la réalisation du SE embarqué, facteur critique dans les MPSoC actuels. Ainsi, l'outil ChronoSym aidera à l'exploration des architectures : dans le partitionnement logiciel/matériel et le choix des composants optimaux pour le sous-système logiciel.

Dans le cadre des perspectives globales pour cette méthode, un grand bénéfice sera atteint dans la connexion de l'outil ChronoSym avec la méthode globale d'évaluation des performances (proposée en Chapitre IV). Elle augmenterait la précision de l'évaluation des performances, sans perdre l'avantage de la vitesse.

Les perspectives locales d'amélioration ou de spécialisation de l'outil sont nombreuses, selon les applicabilités et les besoins des concepteurs dans l'évaluation des performances. Une première amélioration serait d'augmenter l'efficacité de l'outil en fournissant plusieurs granularités pour la fonction **`Delay()`**. Les fonctions pourraient être fusionnées ou considérés séparément, selon les nécessités de l'évaluation. Par exemple, autour des primitives de communication **`Send()/Receive()`**, ces fonctions doivent être incluses avec une granularité très fine. Dans la partie comportementale, il suffit de les associer aux différents blocs de base, avec une granularité plus grossière. Cette stratégie fait partie du compromis entre la vitesse de simulation et la précision temporelle.

La généricité de la méthode offre la possibilité d'extension de ChronoSym pour évaluer à part les temps d'exécution, les accès mémoire, la surface de mémoire nécessaire ou l'énergie consommée par l'application logicielle. En gardant le même flot des opérations, il est besoin seulement de modifier le parseur de code, pour détecter les blocs d'instructions qui affecteront ces deux métriques. Ensuite, il suffit de corréler les instructions avec la surface occupée ou avec la puissance consommée, en étudiant les différents coûts en surface/puissance selon le media sur lequel ils seront exécutés, et selon leur exécution locale.

Conclusions

La technique la plus utilisée pour l'évaluation des performances des sous-systèmes embarqués est la simulation. Les méthodes classiques de simulation, comme la simulation à l'aide d'un simulateur de processeur (ISS), ne répondent plus aux

demandes étendues en complexité du logiciel embarqué. C'est pourquoi l'exécution native est une alternative intéressante.

Cependant, le code du sous-système logiciel embarqué contient des parties spécifiques au processeur décrites par exemple en langage assembleur. Pour les simuler, ils nécessitent des modèles de simulation. Ces modèles de simulation doivent permettre une évaluation à la fois rapide et précise et doivent être obtenus le plus vite possible.

Notre contribution majeure présentée en ce chapitre est le développement d'une méthode de cosimulation logicielle/matérielle en utilisant l'exécution native temporelle du SE réel et de l'application, pour la simulation du logiciel. Le principe de base de l'exécution native temporelle est (1) d'exécuter nativement le vrai code du logiciel, y compris du SE, sur une machine hôte de simulation et (2) de simuler à la fois les temps d'exécution du logiciel et les interactions logicielles/matérielles.

Nous avons conçu l'outil ChronoSym pour pouvoir implémenter cette méthode et l'appliquer au flot de conception ROSES. Les expérimentations montrent que l'exécution native temporelle permet une accélération de 2,1 à 10,8 fois, dans la cosimulation logicielle/matérielle, comparé à la cosimulation basée sur l'ISS. L'erreur d'exécution est de seulement environ de 17%.

Les perspectives qui découlent de ce travail visent l'exploration de l'architecture du SE et le sous-système logiciel en général, mais aussi le partitionnement logiciel/matériel. Un grand bénéfice en précision et vitesse sera atteint dans la connexion de l'outil ChronoSym avec la méthode globale d'évaluation des performances (proposée en Chapitre III). L'extension de l'outil ChronoSym serait une annotation non-uniforme avec des fonctions **`Delay()`** : plus précise pour les parties critiques de code et plus grossière pour les parties qui n'interviennent pas directement dans la performance. Une autre perspective est la spécialisation de l'outil ChronoSym dans l'évaluation des accès mémoire, de la surface de mémoire nécessaire ou l'énergie consommée.

CHAPITRE IV.

Développement d'une approche globale pour l'évaluation des performances des MPSoC

Introduction

Ce chapitre propose une méthode globale capable d'évaluer les performances des systèmes hétérogènes embarqués MPSoCs. La clé de voûte dans la définition d'un modèle global d'évaluation est d'une part l'utilisation de la technique de cosimulation et d'un autre part la génération des interfaces flexibles qui relient entre eux différents modèles particuliers d'évaluation, pour chaque sous-système.

Pour tester la plateforme proposée d'évaluation des performances, on définira comme but la prédiction de la structure optimale d'interconnexion pour une application spécifique, fixée. La partie flexible de l'architecture est le sous-système d'interconnexion qui peut varier d'une réalisation à l'autre : par exemple un bus AMBA ou un réseau sur puce. Cela sera réalisé en s'appuyant sur deux aspects : d'abord par permuter parmi plusieurs sous-systèmes d'interconnexion et ensuite en modifiant leurs paramètres.

Le défi consiste en la définition et la génération des interfaces d'adaptation flexibles permettant d'exécuter l'application avec plusieurs sous-systèmes d'interconnexion. L'idée est de conserver la même description de l'application pour chaque nouvelle architecture ainsi testée. Les interfaces d'adaptation permettent la connexion « transparente » de l'application aux différentes interconnexions.

On illustrera ces contributions dans un cas d'application réelle, au flot de conception du groupe SLS.

La première section de ce chapitre définira le modèle d'évaluation des performances par composition pour des systèmes hétérogènes, basé sur le modèle de cosimulation. Ensuite, la deuxième section présente les interfaces d'adaptation entre l'application et l'interconnexion, définis pour la cosimulation. Dans la troisième section, on particularise les éléments de deux premières sections pour le flot de conception ROSES : en particulier on détaille les bibliothèques utilisées pour sélectionner différents sous-systèmes d'interconnexion, l'interface de communication de l'application et leur adaptation. La quatrième section présente l'application de la méthode pour un codeur DivX, en focalisant sur l'évaluation de différentes structures d'interconnexion. La cinquième section énumère brièvement les avantages de notre méthode.

1 Modèle d'évaluation globale des performances pour les systèmes MPSoC par composition

Cette section présente les besoins pour l'évaluation globale des systèmes hétérogènes MPSoC et fixe les pré-requis d'un modèle pour l'évaluation de performances.

Notre contribution consiste en la définition d'un environnement capable d'évaluer les performances de l'ensemble des sous-systèmes différents, à l'aide du modèle de cosimulation. La première sous-section est consacrée à la proposition du modèle d'évaluation des performances basé sur la cosimulation. Ensuite, les sous-sections suivantes présentent les modèles utilisés dans cet environnement d'évaluation des performances : pour le sous-système d'interconnexion et le sous-système de l'application embarquée s'exécutant sur un ensemble de sous-systèmes logiciels et matériels.

1.1 Modèle d'évaluation des performances basé sur la cosimulation

L'analyse des différentes solutions dans le domaine de l'évaluation des performances, nous a menés à la conclusion que la meilleure manière d'aborder un système hétérogène comme MPSoC est de combiner des techniques spécialisées (voir le Chapitre II). Cela revient à construire le modèle de cosimulation de ce système.

Le flot d'évaluation de performances utilisé dans notre approche est présenté en cette section. Ce flot sera intégré dans le flot complet de conception ROSES qui produit l'architecture synthétisable de la réalisation MPSoC choisie Chapitre I.1.3. En ce

Chapitre, ce flot sera appliqué pour l'exploration des architectures de l'interconnexion à base d'évaluation de leurs performances.

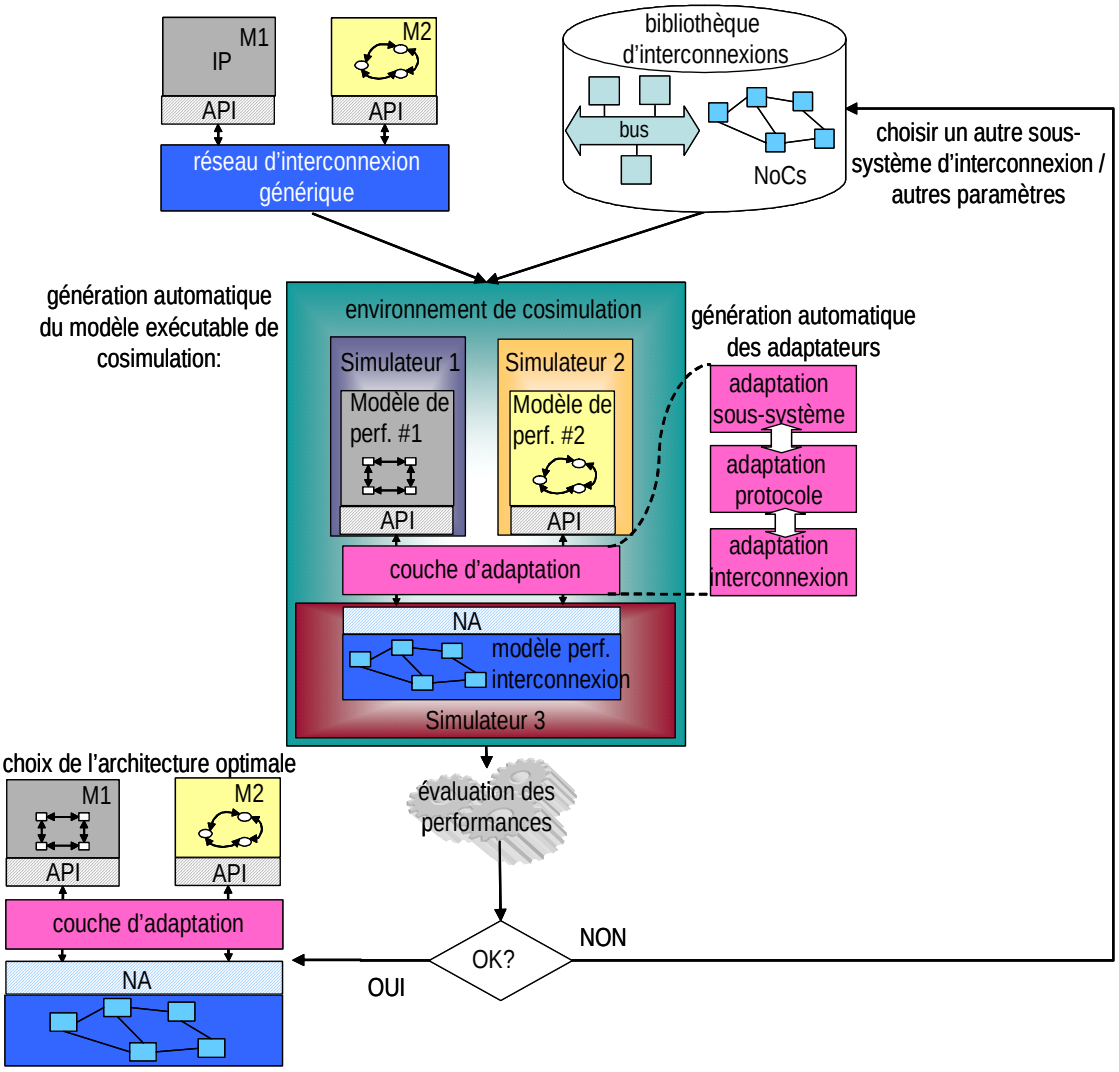


Figure 28. Flot d'évaluation fournissant la meilleure solution architecturale – exploration du sous-système d'interconnexion

La partie fonctionnelle du modèle de performances (comme elle a été définie en Chapitre II.2) est la description au niveau transactionnel de l'architecture MPSoC (pour l'application et l'interconnexion), enrichie avec des annotations de temps d'exécution précises, évalués pour l'architecture réelle. Dans notre approche, elle est décrite sous forme ***d'architecture virtuelle*** [Ces 02].

Les principaux composants de cette architecture virtuelle sont : l'application et le sous-système d'interconnexion. Dans les flots de conception MPSoC modernes, des étapes différentes sont consacrées à la réalisation de l'application et du sous-système

d'interconnexion. D'une part, l'application utilise une API spécifique pour demander des services au sous-système d'interconnexion. D'autre part, le sous-système d'interconnexion utilise aussi son propre API pour permettre l'accès aux services de communication. Souvent, dans les architectures hétérogènes MPSoC, ces deux APIs ne correspondent pas. Par exemple :

- L'application communique par passage de messages[1].
- A son tour, le réseau d'interconnexion implémente des protocoles spécifiques et différents d'une structure à l'autre. Par exemple, pour un bus AMBA, la communication est de type Master/Slave[2]. et basée sur le protocole **`single_read`**/**`single_write`** ; **`burst_read`**/**`burst_write`**.

La **méthodologie d'évaluation de performances** est la cosimulation. L'environnement de cosimulation exécute l'ensemble de tous les simulateurs (pour l'application et pour l'interconnexion). La cosimulation peut être réalisé en employant n'importe quelle approche de cosimulation existante, comme par exemple [Ces 02] [Laj 00] [Log 04]. Le modèle de cosimulation utilisé dans ce travail est celui construit en ROSES, utilisant l'environnement SystemC (présenté dans le Chapitre I.1.2).

Le modèle de performance est le modèle exécutable de cosimulation de l'architecture complète. On appliquera le modèle de cosimulation pour l'évaluation globale de performances des systèmes hétérogènes de la manière suivante :

(a) Chaque sous-système est représenté par son ***modèle de performance***, qui est son modèle de simulation enrichi avec des annotations pour l'évaluation des performances (annotations de temps) et des fonctions pour inspecter certaines métriques (débit, latence, temps d'exécution). Chaque modèle de performance est exécuté sur un simulateur approprié.

(b) Les modèles de performance des composants communiquent via un bus de cosimulation par des ***interfaces d'adaptation***. En principe, ces interfaces adaptent le

[1] message passing
[2] protocole "maître/esclave"

protocole de communication de l'application et divers protocoles de communication du sous-système d'interconnexion (fournis par les interfaces NA).

Les interfaces d'adaptation sont structurellement composées de trois parties : (1) une spécifique au modèle de simulation du sous-système d'application, (2) une spécifique au modèle de simulation du sous-système d'interconnexion, (3) une qui fait la translation de protocoles entre les deux parties précédentes. La conception des interfaces d'adaptation sera détaillée dans Chapitre IV.2.

Pour pouvoir générer automatiquement le modèle exécutable de cosimulation (qui est le modèle d'évaluation des performances), il faut décrire tous les sous-systèmes (l'application et l'interconnexion) sous la forme d'une architecture virtuelle. En plus, les interfaces d'adaptation doivent être intégrés dans des bibliothèques. Ces bibliothèques et la description de l'architecture virtuelle du système seront les entrées de l'outil ROSES. Ainsi, en utilisant ROSES, on pourra paramétrer et choisir automatiquement les interfaces les mieux adaptées au cas de conception à évaluer.

Dans le cas où la conception répond aux contraintes imposées, l'architecture choisie sera implémentée sur le circuit final. Autrement, ses paramètres seront modifiés ou une autre structure d'interconnexion sera choisie. Après avoir trouvé le meilleur sous-système d'interconnexion pour l'application donnée, plusieurs améliorations peuvent être faites et divers paramètres peuvent être accordés plus finement. Par exemple, pour réduire la surface occupée sur puce, on peut utiliser un autre type de réseau (par exemple avec moins de noeuds ou de connexions entre noeuds), ou on modifie la fréquence pour optimiser l'énergie consommée.

1.2 Modèle de l'application embarquée

L'application embarquée s'exécutera sur des CPUs et/ou les blocs matériels qui communiqueront en utilisant le sous-système d'interconnexion. Cette section indique les modèles envisageables pour l'application.

Dans notre flot, pour pouvoir évaluer les performances du sous-système d'interconnexion, on peut utiliser plusieurs modèles d'application, différentes en ce que concernent le comportement, mais aussi l'interface de communication. Ils seront décrits ensuite :

- ***Le comportement décrit avec les détails de la microarchitecture*** : le code de l'application sera compilé sur l'architecture choisie et simulé à l'aide des ISS pour les processeurs choisis, et/ou des simulateurs de matériels. Tous les détails de réalisation (signaux réels) sont décrits. Ce modèle offre une bonne précision, mais une vitesse de simulation réduite.
- ***Le comportement décrit du point de vue fonctionnel*** : abstrait l'environnement d'exécution (les signaux, pins, processeurs réels, etc.). Tous les modules de l'application sont simulés au niveau fonctionnel. Un cas extrême est quand le code de l'application est remplacé par des générateurs du trafic représentant seulement la trace d'exécution de l'application. Ce type de modèle de simulation est le moins précis, mais le plus rapide. Pour augmenter la précision, ce modèle peut être enrichi avec des annotations temporelles.

- ***La communication décrite avec les détails de la microarchitecture*** : s'appuie sur la description des ensembles de ports physiques, des signaux de communication et des protocoles détaillés. C'est une représentation très précise, mais lente à simuler.
- ***La communication décrite du point de vue fonctionnel*** : s'appuie sur des primitives de haut niveau (d'envoi/réception) qui abstraient tous les détails de réalisation de la communication tels que la synchronisation ou le parallélisme. Ces primitives de haut niveau définissent l'ensemble de services requis/fournis de la communication. Ce modèle de communication est le plus efficace, mais le moins précis.

Dans notre cas, afin de simplifier la conception et réaliser une simulation rapide, on va s'intéresser à la représentation de haut niveau. Ainsi, l'application est décrite au

niveau fonctionnel, en SystemC. Pour la précision de ce modèle, l'information temporelle est contenue dans des annotations incluses dans le code. La trace d'exécution de l'application peut être simulée avec la précision d'un cycle d'horloge. Notre modèle combine les avantages de ces deux niveaux présentés.

L'information temporelle est ajoutée au modèle de haut niveau par la fonction `Delay()`, qui simule le temps d'exécution équivalent pour chaque portion de code annotée. Les valeurs d'annotation simulent des délais de calcul et les temps requis d'un sous-système CPU pour des opérations telles que l'accès mémoire ou l'autorisation d'accès à l'interconnexion. Elles sont capturées dans les tables, spécifiques pour chaque configuration choisie, c.-à-d. l'architecture du sous-système CPU. Le procédé d'annotation et les détails concernant le modèle d'évaluation des performances pour les sous-systèmes logiciels sont présentés dans le Chapitre III.

Considérant la représentation de haut niveau pour le système à évaluer, dans notre cas de conception, les différentes tâches parallèles (threads d'exécution) de l'application communiquent par des primitives de haut niveau. Ces primitives représentent l'interface de l'application avec le sous-système d'interconnexion. Elle définit les méthodes nécessaires pour que l'application demande des services à l'interconnexion, et que l'interconnexion fournisse ces services. Ces services exécutent la transmission/réception de l'information par des primitives de haut niveau (de type envoi/réception), qui font abstraction du protocole réel et des fils physiques. L'information véhiculée est elle aussi formatée pour une communication de haut niveau : les messages.

1.3 Modèle du sous-système d'interconnexion basé sur OCCN

Les mêmes modèles avec les détails de la microarchitecture et purement fonctionnel décrites pour le modèle de l'application embarquée (dans la section antérieure) sont valables pour le sous-système d'interconnexion. En cette thèse tous les sous-système d'interconnexion sont décrits au niveau transactionnel, précis au cycle d'horloge. Les annotations sont faites avec les temps d'exécution réels de l'interconnexion, en faisant abstraction de la vraie réalisation des protocoles ou des signaux de communication.

Le modèle de simulation, proposé pour l'évaluation des performances, préserve hiérarchiquement la structure de l'architecture réelle. Ainsi, chaque composant, c.-à-d.

routeur, interface réseau et éléments de calcul, aura sa représentation. À ce niveau, les connexions sont représentées par les relations logiques entre éléments fonctionnels. La transmission de l'information se fait par envoi/réception des paquets de données. Ce modèle est avantageux pour l'exploration rapide d'architecture.

L'architecture de l'interconnexion est choisie parmi les modèles disponibles des interconnexions, décrits par les concepteurs de réseaux (comme par exemple [Pier] et [Han]). Ces modèles peuvent être décrits en trois manières différentes, en utilisant l'architecture virtuelle (décrite en langage VADeL) et/ou la méthodologie OCCN (Chapitre I.3) :

- (1) en utilisant l'architecture virtuelle, par exemple VADeL ;
- (2) en utilisant la méthodologie OCCN ;
- (3) en utilisant un modèle OCCN enveloppé dans composants VADeL.

Les avantages de la méthodologie OCCN (les cas (2) et (3)) consistent en l'utilisation des APIs OCCN (présentée en Chapitre I.3). Ils fournissent des facilités de conception pour la création rapide et la réutilisation des transactions précises au niveau de chaque cycle d'horloge, à travers une variété d'environnements et de plateformes de simulation basés sur SystemC [Cop 03].

2 Conception des interfaces d'adaptation pour l'évaluation des performances

Cette section est dédiée à la présentation des interfaces d'adaptation introduites dans le modèle global d'évaluation des performances (par cosimulation). Notre contribution consiste en la conception de ces interfaces et leur description de manière unitaire pour pouvoir les générer automatiquement, à partir des bibliothèques des interfaces.

Ces interfaces ont été définies que pour la simulation de haut niveau ; la première sous-section présente des possibles réalisations pour ces interfaces, dans le modèle final. Cela est nécessaire, car habituellement, le concepteur désire d'avoir un modèle de simulation le plus proche possible de la réalisation. La deuxième sous-section présente la manière de décrire ces interfaces, et leurs parties composantes.

2.1 Réalisation des interfaces d'adaptation

Les interfaces d'adaptation sont utilisées pour l'évaluation des performances, par la cosimulation du système global. Pour la réalisation du système, on souhaite généralement garder les interfaces d'adaptation utilisées pendant la cosimulation (et l'évaluation des performances). Ces interfaces peuvent être réalisées en logiciel et/ou matériel, selon le compromis de conception qui a été fait (par une étape préalable de partitionnement logiciel/matériel). L'adaptation logicielle est située dans le système d'exploitation (SE) de l'application concernée, alors que l'adaptation matérielle implique souvent une réalisation séparée sous la forme d'un bloc matériel dédié.

Mais, souvent, l'étape d'évaluation des performances est réalisée avant que le système d'exploitation soit encore programmé, ou le matériel encore conçu. C'est pour cela qu'on utilise des interfaces d'adaptation génériques. Leur implémentation n'est pas encore décidée, et ils sont décrites à un niveau abstrait : par les services qu'ils offrent/demandent aux modules (de l'application ou de l'interconnexion) auxquels ils sont connectés.

La représentation de la Figure 29 illustre la possibilité de réalisation des interfaces d'adaptation en logiciel (dans le SE) et/ou en matériel. Ces couches et leurs fonctionnalités sont données en exemples en Chapitre I.2.2.

L'adaptation logicielle peut être réalisée en plusieurs sous-couches hiérarchiques du SE. En HAL, l'adaptation correspond à la réalisation des pilotes de communication (par exemple des appels de lecture/écriture dans la mémoire). Au niveau du « **`Kernel`** »[3] du SE, l'adaptation correspond à la réalisation de haut niveau des pilotes de communication (par exemple la gestion des données ou des messages comme des entités abstraites et la gestion des erreurs). L'adaptation logicielle définit ainsi l'API du SE vers l'application qui rend transparentes à l'application les détails de l'adaptation et communication (par exemple l'application utilise l'API nommée **`MPI_Send()/MPI_Receive()`**, sans s'intéresser au protocole du sous-système d'interconnexion).

[3] noyau

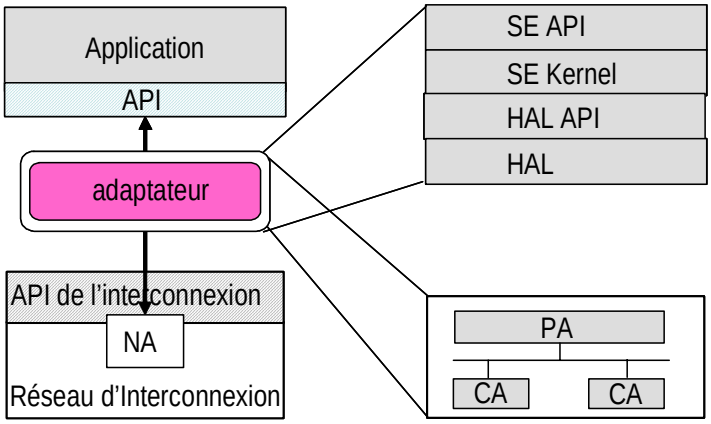


Figure 29. La réalisation logicielle et/ou matérielle de la couche d'adaptation

Une réalisation matérielle pour les interfaces d'adaptation fournit le matériel nécessaire pour réaliser la translation des demandes provenant de l'application (par son API) vers les points d'accès au réseau (notés « NA »).

L'interface d'adaptation matérielle contient un module d'adaptation au processeur (noté « PA »[4]) et un ou plusieurs adaptateurs de canaux de communication (notés « CA »[5]) interconnectés par un bus de communication. Le module PA est en charge de l'intégration du processeur dans l'instance de cosimulation. Sa principale fonction est d'assurer l'échange d'information entre l'API de l'application s'exécutant sur le CPU et le sous-système d'interconnexion. Il est aussi en charge d'une conversion de données, si cela est nécessaire ; par exemple, la conversion d'un type fixe de donnée (entier) en représentation fixe de données en binaire.

L'adaptateur de canal de communication (CA) assure l'adaptation des interconnexions dans une instance de cosimulation d'une architecture virtuelle. Sa principale fonctionnalité est le transfert de l'information de/vers l'interconnexion en effectuant des appels aux APIs fournies par l'interconnexion. Dans le cas où le canal à adapter est un canal virtuel groupant plusieurs canaux de communication (par exemple un bus), la fonctionnalité d'un adaptateur de canal ne se restreint pas aux appels de primitives de communication offertes par le canal ; il a un comportement plus complexe qui respecte la sémantique du protocole groupant les canaux de communication dans un canal virtuel.

[4] Processor Adapter
[5] Channel Adapter

Le modèle de performances (global) du système MPSoC (Figure 30) contient : les modèles de performances (les modèles de simulation de haut niveau avec des annotations temporels) pour les sous-systèmes matériels, logiciels et d'interconnexion, et les interfaces d'adaptation. Une interface d'adaptation sera utilisée pour chaque module logiciel ou matériel.

La génération des interfaces d'adaptation et leur assemblage dans une instance de cosimulation peuvent être réalisé automatiquement dans le flot ROSES. La méthodologie de génération automatique des interfaces d'adaptation s'appuie sur l'introduction des fonctions de protocole spécifiques dans une bibliothèque d'adaptateurs (similaire à la méthode décrite en Chapitre I).

L'architecture de base des interfaces d'adaptation est uniforme (commune à tous les types d'interconnexion). En plus, il faut particulariser chaque interface avec des fonctions spécifiques au type de protocole auquel il appartient. Ainsi, les interfaces d'adaptation sont facilement intégrées dans une bibliothèque extensible, paramétrable de composants. Ces interfaces seront assemblés et configurés à partir des éléments de base (les fonctions de protocole), en respectant le prototype montré en Figure 31.

Le fait que le système MPSoC soit décrit à l'aide des éléments de l'architecture virtuelle rend envisageable la génération automatique des interfaces d'adaptation. Les interfaces seront générées comme des instances particulières à partir de cette bibliothèque générique. Les paramètres de conception extraits de la description du système servent à particulariser les interfaces d'adaptation.

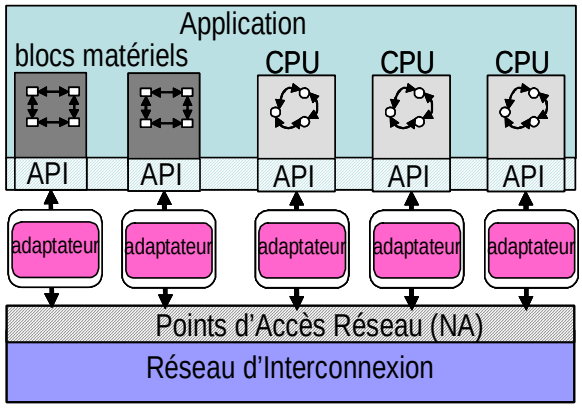


Figure 30. Génération de la couche d'adaptation du modèle de l'architecture MPSoC exécutable

2.2 Spécification des interfaces d'adaptation

Dans la spécification des interfaces d'adaptation entre l'application et l'interconnexion on commence toujours par la spécification des ports des interfaces et des services associés. Les ports de l'interface d'adaptation seront compatibles aux ports de l'interface adaptée : interface de l'application et de l'interconnexion. Pour fournir les services d'adaptation, il faut définir l'ensemble de fonctions de protocole qui assurent ces services. Le comportement de l'interface est déterminé par la composition minimale des fonctions de protocole. Ces éléments seront détaillés en cette section.

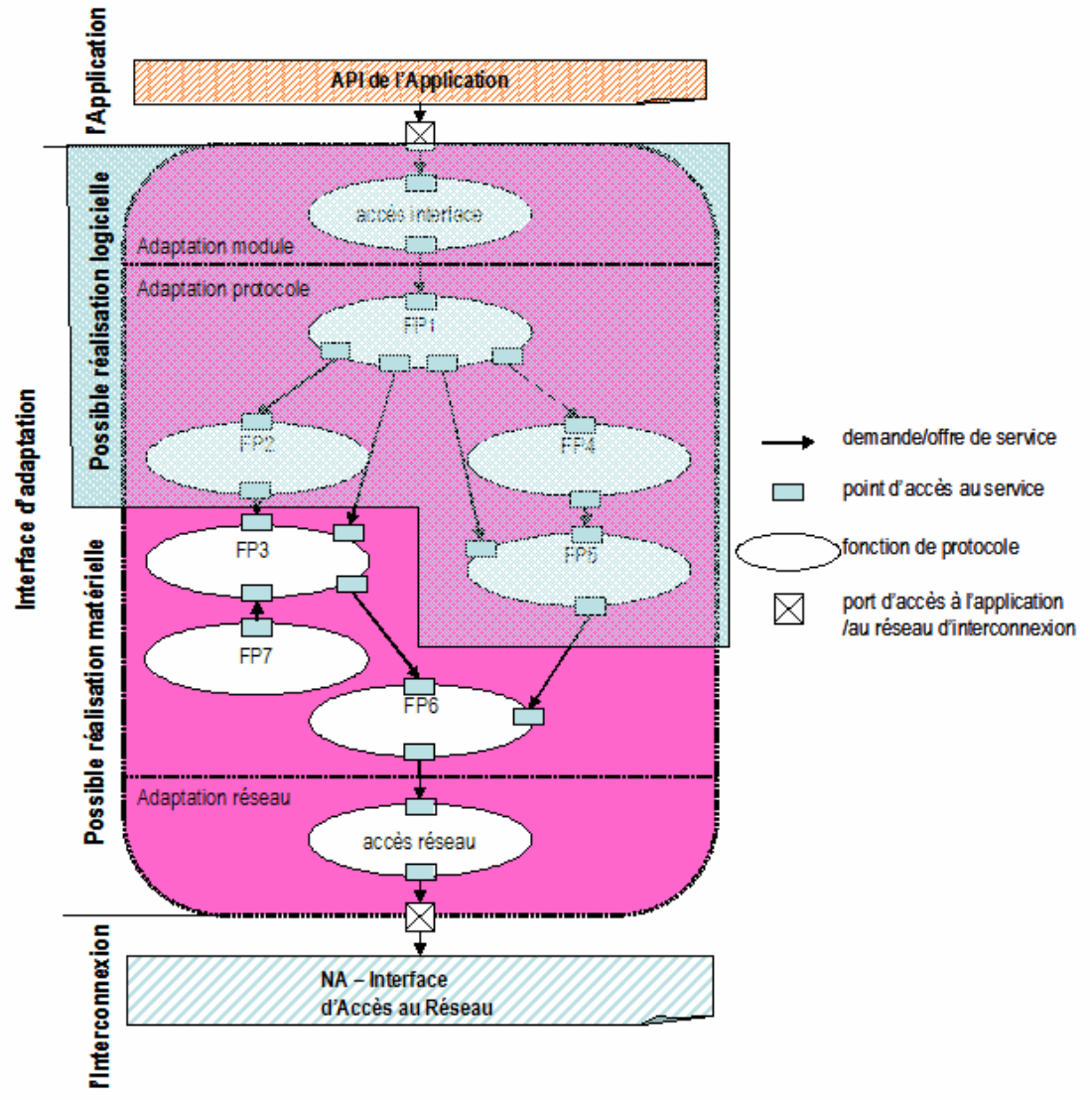


Figure 31. La structure interne d'un adaptateur entre l'application et le sous-système d'interconnexion

La structure d'un adaptateur générique entre l'application et l'interconnexion (entre l'API de l'application et l'interface d'accès au réseau (NA)) contient trois parties, comme illustrée dans la Figure 31) :

- (1) l'adaptation au module de l'application (à l'API de l'application) ;
- (2) l'adaptation au sous-système d'interconnexion (au NA de l'interconnexion) ;
- (3) l'adaptation de protocole entre ces deux éléments.

Chacune de ces trois parties est composée par des fonctions de protocole, qui sont les éléments fonctionnels de base. Ils fournissent et demandent des services aux autres différentes fonctions de protocole par des points d'accès aux services (SAPs). Par exemple, dans une réalisation logicielle, ces SAPs peuvent être vus comme les paramètres des fonctions de protocoles, alors que dans une réalisation matérielle ils sont des ports aux sous-modules de l'interface d'adaptation matérielle. Les relations de dépendance entre différentes fonctions de protocole sont déterminées par l'interconnexion de différents points d'accès aux services. Les différents services composants d'une interface d'adaptation et leurs relations sont capturés d'une manière formelle en utilisant le modèle basé sur des services décrit en [Gra 05] [Sar 05].

L'interface d'adaptation doit faire le transfert entre les services demandés par l'application et les services offerts par les sous-systèmes d'interconnexion. Les services de l'interface sont accédés par ses ports. Chaque port d'accès de l'application ou de l'interconnexion a une fonction de protocole associée, nommés « **`accès interface`** » et « **`accès réseau`** ». La manière de fournir des services en utilisant les services existants (dans les bibliothèques d'adaptation) définit le protocole de transfert. Il est souvent le cas où par le même port on peut accéder à plusieurs services, ou aux services avec réalisations différentes. Ils sont utiles pour réaliser différents types d'interfaces d'adaptation, ou pour pouvoir choisir la réalisation minimale, la plus adaptée pour une application particulière. Dans notre cas les services sont complètement découplés de leur réalisation : le modèle d'interface est générique, non-spécifique à une réalisation particulière ou à un protocole particulier.

Ces interfaces d'adaptation ont le rôle d'intégrer des éléments différents dans le modèle global de cosimulation. Pour la cosimulation, la réalisation des interfaces peut être vue comme un astuce de communication dans le media de simulation : par exemple des mécanismes RPC, des pipes, des mémoires partagées UNIX. Dans la vraie réalisation, les différentes fonctions de protocole seront réalisées en logiciel (par exemple dans la partie HAL du système d'exploitation) et/ou en matériel (comme des circuits d'adaptation).

Deux exemples de spécification des interfaces d'adaptation entre l'application et un réseau-sur-puce sont illustrés dans la Figure 32. L'on considère l'exemple d'une application qui transmet des messages de longueur variable. La première interface d'adaptation offre à l'application des services de segmentation de messages. De l'autre côté, le NA de l'interconnexion reçoit par l'intermédiaire de l'interface des paquets de longueur fixe, chacun précédé par un en-tête correspondant au protocole du réseau. Les services pour la transmission d'un paquet sur le réseau sont : à part les services d'adaptation au module et au réseau, on définit une structure particulière des fonctions de protocole qui retrouve le message dans la mémoire locale, construit le paquet final par segmentation du message et par la construction de son en-tête.

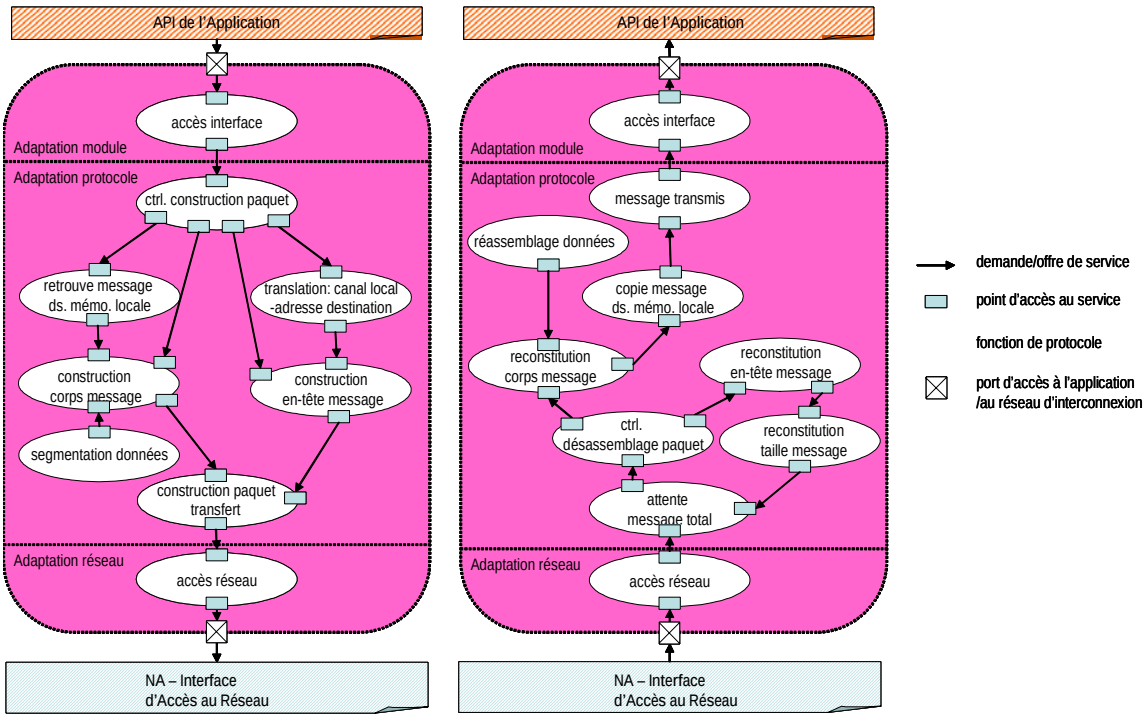


Figure 32. Exemples de spécification des interfaces d'adaptation entre l'application et le réseau-sur-puce (a) module d'adaptation à la transmission ; (b) module d'adaptation à la réception.

L'interface d'adaptation opposé sert à la réception d'un paquet sur le réseau : les fonctions de protocole réalisent les étapes inverses – les appels des NAs du réseau sont traduits dans des fonctions spécifiques à l'application. Il implémente les services inverses, d'assemblage de messages : par l'identification des paquets appartenant au même message, la soustraction de l'en-tête du paquet et la concaténation des corps des paquets reçus.

3 La particularisation de la méthode d'évaluation des performances proposée pour l'outil ROSES

Le modèle d'évaluation de performances est constitué d'un côté des composants logiciels, matériels et d'interconnexion de l'architecture à évaluer, et de l'autre côté, des interfaces d'adaptation. En ROSES, l'application et le sous-système d'interconnexion sont conçus à l'aide des concepts de l'architecture virtuelle. Cette section montre leur représentation sur la plateforme d'évaluation des performances. Nous allons nous intéresser à la construction du modèle de cosimulation et des interfaces d'adaptation. La génération des interfaces d'adaptation peut être automatisée due à la représentation sous forme d'architecture virtuelle du système.

Dans cette thèse on ne s'intéresse pas à la description ou à la modélisation d'un système, mais uniquement à l'évaluation des performances d'un système déjà conçu. Ainsi, la première sous-section présente en détail les sous-systèmes d'interconnexion disponibles, et leur interface de communication (API). La deuxième sous-section présente l'interface de communication (API) de l'application que l'on a utilisé dans le cadre de nos expérimentations. Une fois que l'on a défini les deux éléments de base de la plateforme, la troisième sous-section illustre les adaptateurs spécifiques générés pour chacun des sous-systèmes d'interconnexion présentés (le serveur de mémoire distribuée, le bus AMBA, le réseau-sur-puce Octagon).

3.1 Bibliothèques de conception pour les sous-systèmes d'interconnexion

En cette thèse, on analyse trois sous-systèmes d'interconnexion : le serveur de mémoire distribué (DMS[6]), le bus AMBA et le réseau-sur-puce Octagon. Pour tous

[6] Distributed Memory Server

les sous-systèmes d'interconnexion on utilise leur description simulable de l'architecture, qui est un modèle transactionnel précis au niveau cycle, décrit en SystemC. Ils sont tous décrits à l'aide des concepts l'architecture virtuelle, en langage VADeL [Ces 02]. Pour les bus AMBA et le réseau Octagon ces composants VADeL enveloppent une description OCCN [Cop 03] [Copp 03].

Cette section détaille l'architecture et les primitives de communication de ces trois sous-systèmes d'interconnexion.

3.1.1 Le sous-système d'interconnexion serveur de mémoire distribué – DMS

Le serveur de mémoire distribué (DMS) (Figure 33) est une architecture flexible et scalable. Elle convient au transfert de données pour des systèmes MPSoC qui font des nombreux accès à une mémoire distribuée [Han 04].

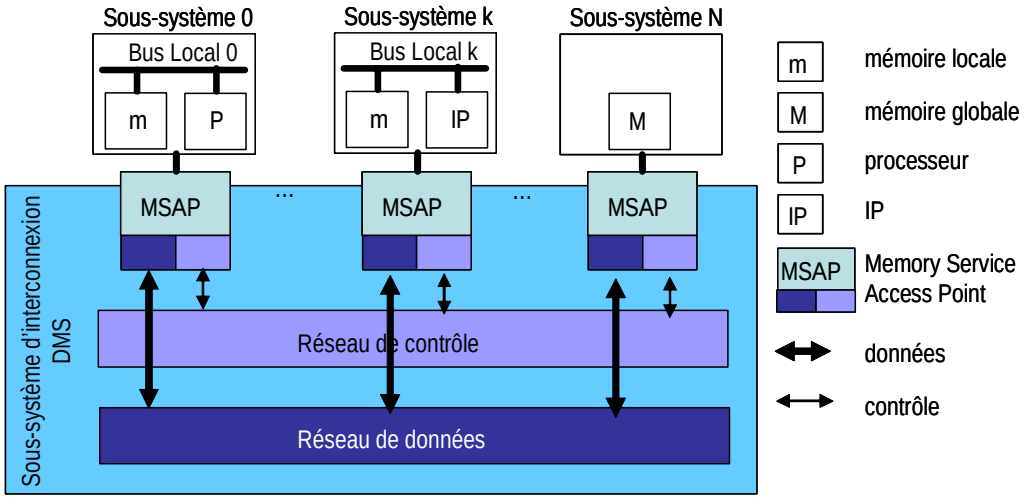


Figure 33. L'architecture du sous-système d'interconnexion DMS

Le DMS se compose de :

- points d'accès à la mémoire (locale ou globale) (MSAPs[7]) ;
- un réseau de contrôle ;
- un réseau de transmission de données.

Un point d'accès à la mémoire (MSAP) agit en tant que moteur de transfert de données, mais aussi qu'interface d'accès au réseau (NA).

[7] Memory Service Access Point

- En tant que **moteur de transfert de données**, **`MSAP`** réalise le transfert entre la mémoire locale du sous-système interconnecté et à autres mémoires des sous-systèmes par l'intermédiaire du « réseau de données ». Ainsi il exempte le sous-système (par exemple le processeur) qui y est interconnecté de cette tâche. Le processeur réalise seulement ses calculs et pour la communication il appelle seulement l'ensemble de primitives d'envoi/réception (son API de haut niveau). C'est l'interface MSAP qui prend en charge tout le transfert de données, incluant les accès à la mémoire locale du processeur.
- En tant que **interface de réseau (NA)**, l'interface **`MSAP`** fournit uniquement l'API de haut niveau au sous-système qui y est interconnecté.

En conséquence, l'interface **`MSAP`** permet le découplage entre les calculs de différents sous-systèmes et leur communication. Ceci est un principe essentiel dans la conception moderne des MPSoCs [Rij 03]. Une exécution parallèle entre les différentes tâches et la communication est possible en utilisant les propriétés de l'interface **`MSAP`** [Han 04].

Pour mettre en place la communication entre deux sous-systèmes, le réseau de transfert de données doit être configuré. Une phase de configuration précédera chaque communication. Pendant cette phase de configuration, des interconnexions point-à-point sont définies entre chaque paire de ports attachés à différents **`MSAPs`**, connectant des sous-systèmes différents. Ainsi, finalement, le réseau DMS se résume à des connexions point-à-point établies dynamiquement pendant ces phases de configuration.

Une interconnexion point-à-point est décrite par une paire $\texttt{lch}_x\texttt{-lch}_y$. Puisque ces interconnexions sont unidirectionnelles, deux ports sont réservés dans l'interface correspondante de **`MSAP`** pour chaque module. Ceci assure une communication bidirectionnelle. La Figure 34 montre un tel système où **`Sous-système 0`** communique avec **`Sous-système 1`** par l'intermédiaire des canaux point-à-point « ouverts » dans le réseau de données de DMS.

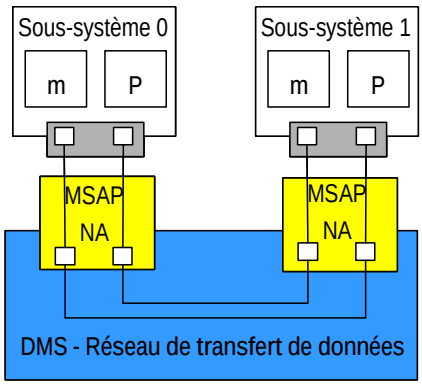


Figure 34. Interconnexion point-à-point dans le sous-système d'interconnexion DMS

L'interface fournie à l'application par DMS contient les primitives de transfert :

```
void mpi_send (lch, laddr, size);

void mpi_recv (lch, laddr, size);
```

Pendant que le transfert de données est exécuté par l'intermédiaire du réseau de données, les demandes de transaction et toute autre information de synchronisation sont assurées par des signaux envoyés par le réseau de contrôle. Par exemple, l'appel **`mpi_send()/mpi_recv()`** d'un module est bloqué jusqu'à ce que le **`mpi_send()/mpi_recv()`** correspondant soit lancé par le module communiquant avec lui.

3.1.2 Le sous-système d'interconnexion du bus AMBA

Par opposition au réseau DMS se traduit finalement par des connexions point-à-point (comme il est montré dans la section précédente et plus détaillé en [Han 04]), la communication par le bus est sujet à la congestion. En examinant l'activité du bus, on peut déterminer le modèle de trafic pour une application donnée, c.-à-d. le degré du parallélisme ou les transferts en mode « burst »[8].

Le bus AMBA AHB agit en tant que moyen de communication avec rendement élevé. Il est utilisé pour interconnecter efficacement des processeurs, des mémoires embarqués sur puce et même des interfaces externes des mémoires, hors puce. Les caractéristiques du bus AMBA AHB incluent la configuration de données (par

[8] transmission en rafale

exemple 64/128 de bits) et les transferts en mode « burst ». Il est facilement utilisable dans les méthodes de conception et test [Amba].

Une conception typique pour un système contenant le bus AMBA AHB implique les composants suivants :

- **les modules « maîtres »** qui lancent des opérations lecture/écriture ;
- **les modules « esclaves »** qui répondent aux opérations de lecture/écriture ; ils ont chacun sa propre plage d'adressage sur le bus, définie au moment de la description de l'architecture ;
- **l'arbitre du bus** qui autorise le transfert de données pour un seul maître à la fois;
- **le décodeur du bus** qui décode l'adresse de chaque transfert sur le bus et fournit les signaux sélectionnés.

La Figure 35 présente tous ces éléments et leurs d'interconnexion ; cette figure peut représenter l'architecture à réaliser, ou seulement un modèle de simulation. Le bus AMBA peut être utilisé dans une configuration simple, ou dans des hiérarchies des bus interconnectée par des « **`bridges`** »[9].

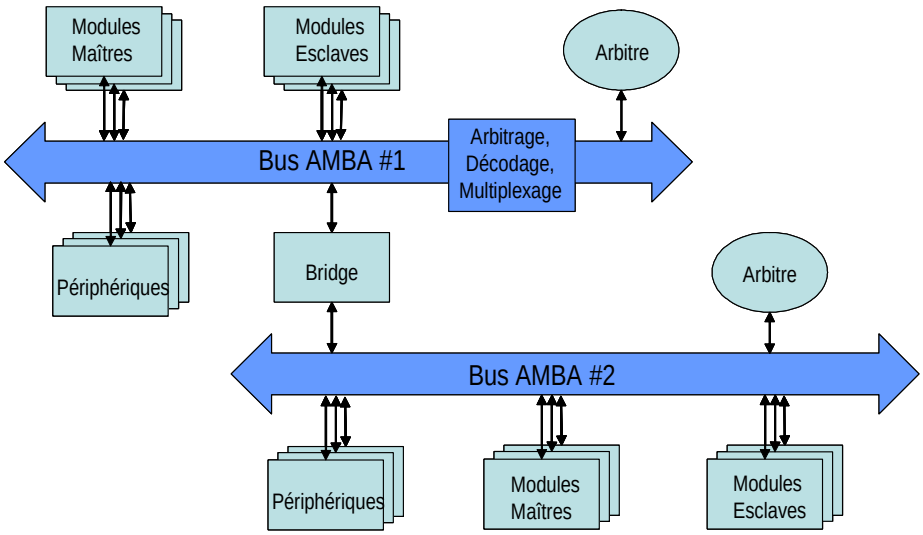


Figure 35. Architecture à base du bus AMBA

[9] ponts de connexion

L'interface fournie à l'application par le bus AMBA est basée sur l'API OCCN **`MasterPort/SlavePort`**, décrits en [Copp 03]. Les primitives de transfert appelées par les modules « maîtres », appartenant au protocole de communication sur le bus sont :

```
single_write (data, size, addr);
single_write_leaving (data, size, addr);
single_read (data, size, addr);
single_read_leaving (data, size, addr);
```

Ces primitives écrivent ou lisent des données d'un seul mot, avec une longueur de taille « **`size`** » bytes, du module indiqué par paramètre « **`addr`** ». Les primitives **`single_write()/single_read()`** sont bloquantes pour le bus. Le bus sera débloqué après chaque transfert avec les primitives **`single_write_leaving()/single_read_leaving()`**.

3.1.3 Le sous-système d'interconnexion réseau-sur-puce Octagon

Les réseaux-sur-puce interconnectent différentes ressources des MPSoCs, comme des noyaux de processeur, DSPs, FPGAs, mémoires, par l'intermédiaire des interfaces réseau et des routeurs, et à l'aide d'une politique de routage. La communication est basée sur le transfert des paquets.

Le réseau-sur-puce Octagon, schématiquement présenté dans la Figure 36, utilise une topologie simple et régulière pour l'interconnexion de ses routeurs. Les ressources sont associées aux routeurs, par des interfaces de réseau (NI[10]).

Le principal avantage de la communication par des NI est le découplage total entre le comportement et la communication d'un module [Han 04]. Par conséquent, la communication inter-module n'est jamais considérée conjointement avec le comportement, étant complètement indépendante. Ceci laisse exprimer le comportement et la communication à des niveaux d'abstraction différents. Par exemple, des descriptions de haut niveau ou des IPs matériels peuvent être

[10] Network Interface

interconnectés via un réseau décrit au niveau microarchitectural, ou via un réseau abstrait décrit au niveau transactionnel.

La topologie de base de cette famille de NoCs interconnecte un nombre pair, générique, de noeuds **N=2n** (où **n** est un nombre entier). En particulier, Octagon a huit nœuds. Chaque noeud est relié par trois liens vers trois autres nœuds, dans la forme d'un anneau bidirectionnel et avec des interconnexions en diagonale pour chaque couple des nœuds avec symétrie radiale. Les nœuds suivent des règles d'interconnexion précises : dans le sens contraire des aiguilles d'une montre, dans le sens des aiguilles d'une montre et respectivement avec un raccordement en travers.

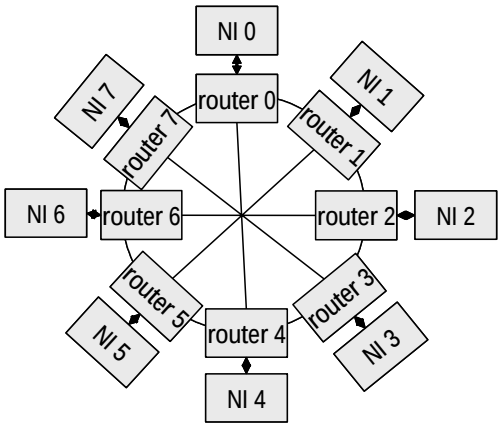


Figure 36. L'architecture du sous-système d'interconnexion Octagon

L'algorithme de routage détermine la route de la source vers la destination. Il est essentiel dans la conception des réseaux-sur-puce d'avoir un algorithme de routage efficace : qui minimise le nombre des étapes, nécessite une surface minimale de stockage et bénéficie d'une fréquence élevée. Octagon adopte un algorithme de routage déterministe et local à chaque nœud. Il est basé sur la recherche du chemin le plus court entre source et destination. La technique de routage des paquets s'appelle « wormhole »[11]. Elle réduit rigoureusement l'espace nécessaire de mémorisation pour le réseau et réalise la communication en pipeline pour chaque paquet. Pour plus de détails, il faut faire référence à [Cop 03].

Pour le réseau-sur-puce Octagon, un message est fragmenté en « flits »[12]. Un « flit » est défini comme la plus petite subdivision d'un paquet qui peut être transporté par le

[11] protocole "trou-de-ver"
[12] flow control digit

NoC, en un seul échange. La dimension (en nombre de bits à transférer) est paramétrable : par exemple on utilisera 32 bits. Le premier flit d'un message contient toujours l'information de routage, en ouvrant le chemin. Les flits suivants sont guides sur le même chemin du réseau que le premier, jusqu'à ce que le message entier ait été envoyé - jusqu'à la détection du « dernier flit ».

L'interface fournie à l'application par Octagon contient les primitives de transfert :

```
void send(const msg_type& flit);
msg_type* recv().
```

Le type du message (« msg_type ») peut être n'importe quel type standard, ou défini par le concepteur au moment de l'instanciation de l'architecture. Il est basé sur une classe de type « template ».

3.2 L'interface de communication pour l'application embarquée

Le modèle du sous-système d'application a été décrit en Chapitre IV.1.2. Les sous-systèmes logiciels et matériels sont représentés par leurs modèles de performances de haut niveau. On utilise la description de haut niveau, encore indépendante de la réalisation, annotée avec les temps d'exécution sur les CPUs / matériels respectives. A ce niveau, les différents sous-systèmes de l'application communiquent par passage des messages (à l'aide des primitives MPI, que l'on présentera dans cette section).

L'interface de programmation de haut niveau pour l'application contient les appels de fonctions suivantes :

```
void mpi_send (lch, laddr, size);
void mpi_recv (lch, laddr, size);
```

Les paramètres intervenant dans ces APIs sont :

- « **lch** », identifiant d'une manière unique le canal de communication et le module qui appelle la primitive **mpi_send** ou **mpi_receive** avec ce paramètre ;
- « **laddr** », contenant l'adresse relative nécessaire pour retrouver le message dans la mémoire locale du module appelant.

L'adresse de base de cette zone de mémoire locale est déterminée au temps de l'initialisation du module correspondant ;

- « **`size`** » indiquant la longueur du message à transmettre.

Ces primitives de communication assurent deux rôles : de transmission de l'information et de synchronisation. Pour la synchronisation, il y a aussi des signaux auxiliaires de protocole qui sont utilisés :

- pour réaliser l'envoi de données, il faut d'abord notifier la présence des données sur un signal spécifique. Le module « émetteur » peut démarrer l'envoi seulement après que le module « récepteur » est capable de recevoir les données ;
- pour réaliser une réception de données, il faut recevoir un signal de notification de la présence des données dans le module « émetteur ». Seulement après, le module « récepteur » peut commencer la réception des données.

Ces primitives de communication ont un caractère bloquant. Ils restent bloqués dans une attente active pour l'envoi et pour la réception. Seulement quand ces fonctions (d'envoi/réception) sont finies, l'application peut commencer le traitement des données.

La description et le mode de fonctionnement de ces deux primitives, ainsi que les détails de comment on établi les paramètres, seront présentés sur des exemples dans les sections qui suivent (Chapitre IV.3.3.1).

3.3 Adaptateurs spécifiques pour l'évaluation des performances

Dans ce paragraphe on présentera l'adaptation en vue de l'évaluation des performance, pour une application spécifiée et différents sous-systèmes d'interconnexion : le réseau DMS, le bus AMBA et le réseau-sur-puce Octagon.

Le réseau DMS est l'interconnexion native pour l'application considérée. Ensuite, pour pouvoir remplacer le réseau DMS par le bus AMBA ou par le réseau-sur-puce Octagon, des interfaces d'adaptation seront générés. Ces adaptateurs rendent transparente la communication entre l'API original de l'application et le bus AMBA

ou Octagon NoC. Apres montrer la communication de l'application à l'aide du DMS (section 3.3.1), les différentes phases de l'adaptation sont décrites de manière générale, et ensuite particularisées pour le bus AMBA et pour le réseau Octagon (section 3.3.2).

3.3.1 La communication via le sous-système d'interconnexion DMS

Le réseau DMS est l'interconnexion native pour l'application considérée (un codeur DivX, qui sera présenté en Chapitre IV.4.1). L'application et l'interconnexion qui utilisent les mêmes APIs et les mêmes protocoles de communication. Dans ce cas particulier, l'adaptation n'est pas nécessaire. La plateforme d'évaluation des performances contiendra : les modèles de simulation (modèles d'évaluation des performances) de l'application distribuée sur plusieurs modules qui s'exécutent en parallèle et du réseau DMS directement interconnecté avec cette application.

On illustrera dans un exemple le principe de communication du DMS (Figure 37). Même si on n'utilise qu'un modèle de simulation, il a au niveau cycle-près le même comportement que le réseau DMS qui sera réalisé. Dans ce scénario simple, le module $\mathbf{M_1}$ envoie des données aux modules $\mathbf{M_2}$ et $\mathbf{M_3}$. Chaque module sera connecté au réseau DMS par une interface **MSAP**. Les liens qui connectent un module $\mathbf{M_x}$ à son **MSAP** correspondant sont groupés dans les canaux virtuels. Ils sont notés avec $\mathbf{VC\_M_x\_MSAP}$, ou $\mathbf{M_x}$ est le nom du module correspondant.

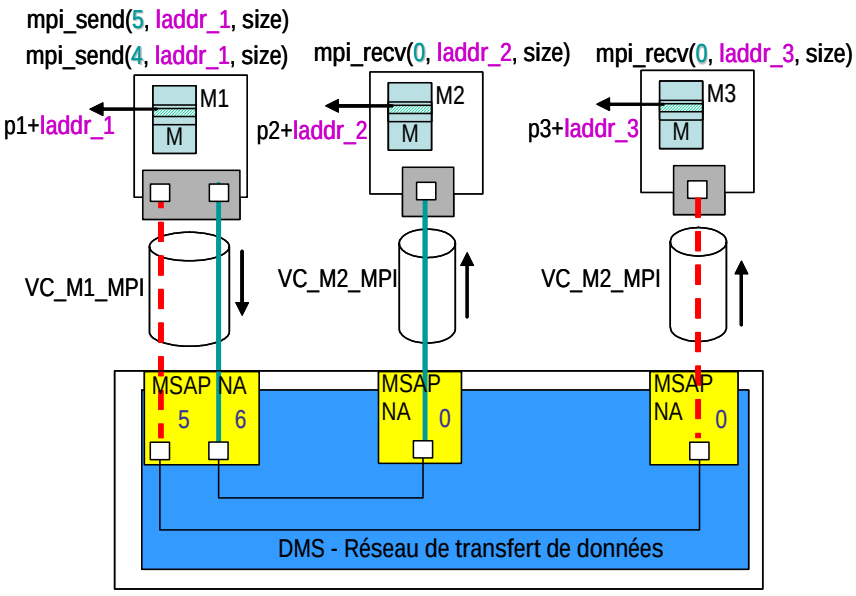


Figure 37. Le principe de communication DMS

Le système d'adressage du DMS est basé sur les identificateurs des ports connectés à l'interface MSAP. Les primitives **`mpi_send()`** et **`mpi_recv()`** appellent cet identificateur dans le paramètre désignant le canal local « **`lch`** ». Chaque interface **`MSAP`** peut compter 10 canaux. Les canaux désignés par 0 à 4 sont pour réception et les canaux désignés par 5 à 9 sont pour émission.

Avant de commencer le transfert effectif sur ce réseau, il faut réaliser une initialisation du réseau de données. Pendant cette phase, des interconnexions unidirectionnelles point-à-point sont établies : $\mathbf{M_1}$ vers $\mathbf{M_2}$ et $\mathbf{M_1}$ vers $\mathbf{M_3}$. Chaque connexion point-à-point est enregistré à l'intérieur du DMS, par la paire $\mathbf{[MSAP_x : port_i - MSAP_y : port_j]}$. Dans l'exemple présenté (Figure 37), les données de configuration du réseau qui constituent le système d'adressage du DMS, sont comme suit :

$\mathbf{M_1 \rightarrow M_2}$ **est configuré :** $\mathbf{[MSAP_1 : 5 - MSAP_2 : 0\ ]}$

$\mathbf{M_1 \rightarrow M_3}$ **est configuré :** $\mathbf{[MSAP_1 : 6 - MSAP_3 : 0\ ]}$

Après l'initialisation des modules, le MSAP connaîtra l'adresse de base de la mémoire locale du module auquel il est connecté. Ainsi, $\mathbf{p_1}$ contient l'adresse de base pour la zone de mémoire locale qui stocke des messages pour module $\mathbf{M_1}$, et analogue, $\mathbf{p_2}$ et $\mathbf{p_3}$ pour les modules $\mathbf{M_2}$ et $\mathbf{M_3}$.

Table 8. La table des primitives de communication utilisées

MODULES	*COMMUNICATION*	
	M1 -> M2	*M1 -> M3*
M1	***mpi_send(5, laddr_1, size)***	***mpi_send(6, laddr_1, size)***
M2	***mpi_recv(0, laddr_2, size)***	**--**
M3	**--**	***mpi_recv(0, laddr_3, size)***

Le transfert de données à partir de $\mathbf{M_1}$ vers $\mathbf{M_2}$ est fait par les étapes suivantes (Table 8) :

- L'application qui s'exécute sur le module $\mathbf{M_1}$ appelle la primitive **`mpi_send(5, laddr_1, size)`** avec les paramètres :

le canal local (« **lch** ») étant **5**, la taille du message étant « **size** », le déplacement par rapport à la mémoire locale étant « **laddr_1** » ;

- **$MSAP_1$** traitera la demande en recherchant le message localisé dans la mémoire du module **M_1** à l'adresse « **p_1 + laddr_1** », et en le transférant dans un buffer interne au **$MSAP_2$** ;
- L'application qui s'exécute sur le module **M_2** appelle la primitive **mpi_recv(0, laddr_2, size)** avec les paramètres : le canal local (« **lch** ») étant **0**, la taille du message étant « **size** », le déplacement par rapport à la mémoire locale étant « **laddr_2** » ;
- **$MSAP_2$** traitera la demande en transférant le message à partir de son buffer, à la mémoire locale du module **M_2**, à l'adresse **p_2+laddr_2**.

Le transfert de données à partir de **M_1** vers **M_3** se produit d'une façon semblable.

3.3.2 Les interfaces d'adaptation pour l'API « `mpi_send()` » et « `mpi_recv()` »

La fonctionnalité des interfaces d'adaptation consiste principalement dans la translation d'adresses et le formatage de l'information envoyée/reçue par l'application, pour la voie de transmission : segmentation/réassemblage de paquets. Les APIs utilisés par l'application pour communiquer (« **mpi_send/ mpi_recv (lch, laddr, size)** ») doivent être « traduites » en routines spécifiques au protocole de transmission du sous-système d'interconnexion effectivement choisi.

Nous allons remplacer un sous-système d'interconnexion initial (par exemple le DMS) avec des structures d'interconnexion différentes (par exemple le bus AMBA, le réseau-sur-puce Octagon). Pour le sous-système d'interconnexion DMS, toutes ces actions sont prises en charge par l'interface du réseau – le MSAP. Pour les autres sous-systèmes d'interconnexion, des interfaces d'adaptation doivent être conçues dans ce but. Leur rôle est de préserver intacte la fonctionnalité de l'application, comme si elle communiquerait encore sur l'interconnexion initiale, et de pouvoir utiliser (dans une manière « transparente ») des autres sous-systèmes d'interconnexion, différents.

On a identifié trois phases absolument nécessaires pour l'adaptation entre l'API de l'application et les fonctions de transmission de l'interconnexion.

- **(1) La première phase d'adaptation** a lieu pendant la programmation du système à évaluer. Il s'agit de déterminer le mapping de l'application sur le sous-système d'interconnexion : comment les différents modules sont liés aux ports de l'interconnexion, quels seront leurs identifiants, etc.
- **(2) La deuxième phase d'adaptation** a lieu pendant le fonctionnement des interfaces d'adaptation. Ici, le paramètre indiquant le canal local « `lch` », est traduit dans une adresse valide pour ce sous-système interconnexion. On doit définir un schéma de translation d'adresses qui projette l'espace d'adressage initial (du DMS), basé sur des identifications des canaux locaux, dans l'adressage de la nouvelle structure d'interconnexion (bus AMBA, NoC Octagon), basé sur des adresses ou des identificateurs de modules. Suivant ce schéma de translation, l'information sera transmise vers la destination appropriée dans le nouveau media d'interconnexion.
- **(3) La troisième phase d'adaptation** a aussi lieu pendant le fonctionnement des interfaces d'adaptation. Elle réalise le formatage des données en paquets adaptés à la bande du sous-système d'interconnexion utilisé (par exemple des paquets de 32 bits), chacun précédé par un en-tête contenant des signaux spécifiques. Cet en-tête aura une signification seulement pour ce sous-système d'interconnexion, et il sera utilisé pendant la transmission de données.

En considérant ces trois phases définies précédemment, la communication entre un module « émetteur » (respectivement « maître ») et un module récepteur (respectivement « esclave ») peut être résumée dans les actions suivantes.

Les primitives « **mpi_send(lch_1, laddr_1, size)** » appelé par le module de l'application **« émetteur »** (« **maître** ») seront traitées maintenant par l'interface d'adaptation **« émetteur »** (« **maître** »), au lieu de l'interface MSAP. Les tâches de l'interface d'adaptation sont :

- localisation et recherche les données de la mémoire locale du module « maître », en utilisant le paramètre d'adresse locale « **laddr_1** ».
- traduction du paramètre « **lch_1** » dans un identifiant de module valide pour retrouver le « récepteur » (« l'esclave ») ;
- la segmentation du message en paquets OCCN nommés PDUs (présentés dans Chapitre I.3), et l'ajout de l'en-tête.
- envoi de la structure de données résultante vers le réseau d'interconnexion en utilisant l'API de communication de ce dernier.

Du côté **interconnexion**, dès que les données de l'interface d'adaptation de « l'émetteur » (« maître ») arrivent, l'interconnexion prend en charge le transfert, à l'aide de ses mécanismes internes : par exemple l'arbitrage et l'envoi dans un bus, ou des algorithmes de routage et des transmissions via des canaux virtuels dans un réseau-sur-puce.

A son tour, l'interface d'adaptation « récepteur » (« esclave ») considérera l'appel **mpi_recv()** envoyé par le module « récepteur » (« esclave »). Les tâches de l'interface d'adaptation « récepteur » (« esclave ») sont :

- appel dans un fil d'exécution de la fonction **receive(lch_2, laddr_2, size)** de l'API de communication de OCCN, dans l'attente de recevoir les données du de l'interconnexion ;
- re-assemblage des PDUs pour reconstituer le message original, après avoir enlevé les en-têtes ;
- transférer le message dans la mémoire locale du module « récepteur » (« esclave »), à l'adresse indiquée par le paramètre d'adresse locale « **laddr_2** ».

Les trois phases pour la segmentation/réassemblage de paquets, ainsi que la communication émetteur/récepteur (maître/esclave) sont illustrées en suite, pour le bus AMBA et le réseau-sur-puce Octagon.

Les interfaces d'adaptation pour le sous-système d'interconnexion AMBA AHB

Pour remplacer du réseau DMS par le bus AMBA sur la plateforme d'évaluation des performances, les interfaces d'adaptation ont été programmés pour réécrire les primitives « **`mpi_send()`** » et « **`mpi_recv()`** » de l'application, afin de rendre transparente la communication par le bus AMBA qui utilise ses propres primitives « **`single_write`** »/ « **`single_read`** ». Aussi, il faut respecter les protocoles du bus et confirmer la réception de données, en utilisant des fonctions de contrôle (**`recv_completion()`**).

La première phase d'adaptation, pendant la programmation du système, réalise le mapping des modules de l'application sur des « maîtres »/« esclaves ». Les modules « maîtres » sont interconnectés au bus AMBA par des interfaces d'adaptation que l'on va nommer « adaptateurs maîtres ». Les modules « esclaves » sont interconnectés par des « adaptateurs esclaves ». Les modules de l'application communiquent avec leurs adaptateurs en utilisant l'API de MSAP. A leur tour, les interfaces d'adaptation communiquent avec l'interconnexion en utilisant les primitives de communication propres à l'interconnexion respective.

La deuxième phase, pendant le fonctionnement des interfaces d'adaptation, interprété la communication à base de canaux locaux « **`lch`** », dans une communication à base d'adresses, dans l'espace d'adressage du bus AMBA – adresses exprimées en format hexadécimal. Par exemple, la valeur **`3`** du « **`lch`** » pourrait correspondre à l'adresse logique **`0x3333.`** Pendant la configuration du système (et du réseau), il y a une phase de configuration, précédente à l'adaptation, où on construit des tables internes d'adressage, contenant ces correspondances. Pour cela, on prend en compte la plage d'adressage dans le mapping des modules esclaves, sur le bus, et les canaux de communications entre maîtres et esclaves.

La troisième phase, pendant le fonctionnement des interfaces d'adaptation, est en charge de la création des paquets à envoyer, contenant :

- le corps du message – respectivement 32 bits d'information utile pour la communication inter-module ;
- un en-tête avec les valeurs pour quelques signaux spécifiques au protocole AMBA (par exemple les signaux HBUSREQ, HBURST, HLOCK, HADDR), qui ont une signification pour le transport par le bus.

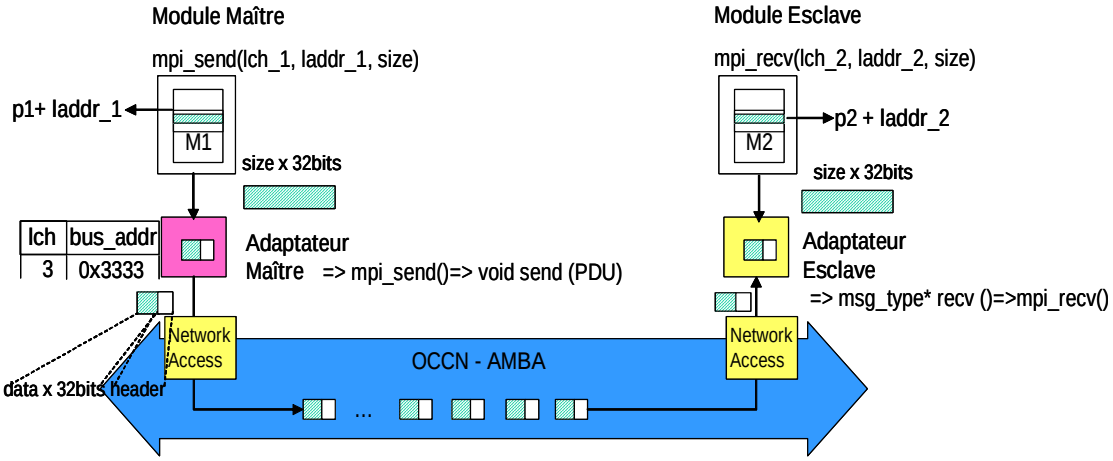


Figure 38. Génération des interfaces d'adaptation pour le sous-système d'interconnexion AMBA

L'exemple le transfert de données entre un module « maître » et un module « esclave » est illustré en Figure 38. L'adaptation des primitives **`mpi_send()/mpi_recv()`** est exprimée en pseudocode comme suit :

```
mpi_send(lch_1, laddr_1, size)
      1. copie du bloc de mémoire contenant le message, dans le
      buffer
      2. for(i=0; i<size; i++){          //segmentation
      3. construction(Header);//ajouter HBUSREQ, HBURST, HLOCK,...
      4. construction(Data);  //ajouter 32 bits de donnée du buffer
      5. send(Data+Header);
}
```

```
mpi_recv(lch_2, laddr_2, size)
      1. receive(), appelé dans un thread
      2. stocker Data dans le buffer
      3. si (réception finie)
            recv_completion_notify() ;
      4. wait(recv_completion) ;
      5. copier le buffer dans le bloc de mémoire du message
```

Les interfaces d'adaptation pour le sous-système d'interconnexion Octagon NoC

L'adaptation pour le sous-système d'interconnexion Octagon NoC fonctionne de la même manière comme pour le cas du bus AMBA. Les différences concernent le système d'adressage et la structure des paquets.

La phase de mapping (première phase, qui a place pendant la programmation du système) associe les modules avec les nœuds dans le réseau Octagon. Ainsi, chaque module aura un identificateur « **`ID`** », le numéro du routeur auquel il est connecté. Les identificateurs varient dans la gamme **`0-7`**, car dans l'application présentée, le réseau Octagon a 8 noeuds.

La deuxième phase (qui a place pendant l'adaptation) établit le schéma de translation d'adresses. Le paramètre « **`lch`** » est assigné avec une identification de destination « **`destID`** », qui sera placée dans l'en-tête du premier flit. De cette façon, le message peut être conduit via le réseau Octagon, à la destination correcte. Une table interne est employée à cet effet : par exemple, la valeur **`3`** du « **`lch`** » pourrait correspondre à un « **`destID`** » de **`1`**. Le contenu de cette table est généré au moment du mapping des modules de l'application sur les noeuds du réseau. Pendant la simulation, la table (programmé comme une structure) est accessible par des pointeurs vers différentes locations pour donner les identificateurs de modules et les canaux locaux.

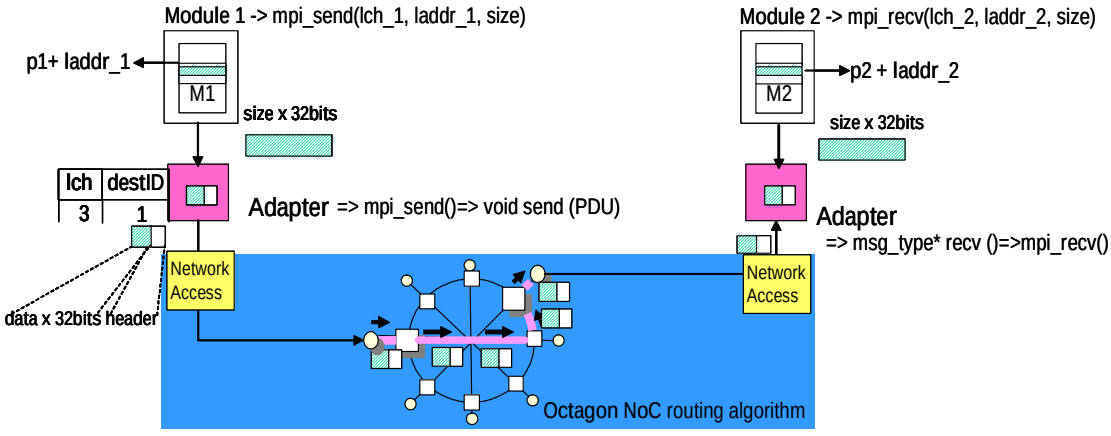


Figure 39. Génération des interfaces d'adaptation pour le sous-système d'interconnexion Octagon

La troisième phase (qui a place pendant l'adaptation) créée les paquets contenant :

- le corps du message – respectivement 32 bits d'information utile pour la communication inter-module ;

- un en-tête avec des informations utiles pour l'algorithme de routage : l'identification de destination « destID », l'identification de source « sourceID » et la longueur de message « size ».

L'exemple de la Figure 39 montre un transfert de données entre un module qui agit en tant qu'émetteur et un module ce qui agit en tant que récepteur. L'émetteur est mappé au nœud 6 (« **sourceID** » = 6), et le récepteur au nœud 1 du réseau (« **destID** » = 1). Le pseudocode des primitives **mpi_send()/mpi_recv()** est décrit ci-dessous. Dans ce cas, les interfaces d'adaptation qui relient les modules d'application au réseau d'interconnexion accomplissent une tâche plus simple comparée à l'exemple de la Figure 38 de la section précédente.

```
mpi_send(lch_1, laddr_1, size)
      1. copie du bloc de mémoire contenant le message, dans le
      buffer
      2. construction(Header);   //ajouter [destID, sourceID, size]
      3. Data = message;
      4. send(Data+Header);
```

```
mpi_recv(lch_2, laddr_2, size)
      1. receive(), appelé dans un thread
      2. stocker Data dans le buffer
      3. reconstituer le message original (enlever le Header)
      4. copier le buffer dans le bloc de mémoire du message
```

4 Expérimentations

Cette section présente l'application de la méthode d'intégration et de génération des interfaces d'adaptation dans le cadre du flot de conception ROSES.

Nous commencerons par présenter l'application utilisée – le codeur DivX – et ses contraintes de performances. Cela s'inscrit dans le travail du groupe, à côté des thèses qui ont exploré les solutions algorithmiques pour cette application ([Bona 06]) et des thèses qui ont visé la réalisation pratique de cette application, en matériel ([Pet 06]). Dans cette thèse, on a repris cette application et on a exploré ses performances avec différents réseaux d'interconnexion.

Les réseaux d'interconnexion ont été fournis ; leur modélisation et description paramétrique font l'objet des autres thèses dans le cadre du groupe SLS ([Pier]et [Han]). Les descriptions des sous-systèmes d'interconnexion sont précises, en termes d'information temporelle au niveau cycle.

Pour réaliser le système composé de l'application (DivX) et différents interconnexions, on décrira les interfaces d'adaptation : adaptation entre différents protocoles de communication (décrits en Chapitre IV.3.1 et Chapitre IV.3.2).

En cette section, nous commencerons par présenter l'application utilisée – le codeur DivX – et ses contraintes de performances. Ensuite, dans la deuxième sous-section, on va illustrer quelques exemples concrets d'architectures pour la réalisation de cette application et on évaluera leurs performances globales (en focalisant sur l'interconnexion). A la fin, dans la troisième sous-section, on comparera ces architectures en conformité avec des métriques de performances pré-établies.

4.1 Spécification de l'application – le codeur DivX, et la mise en place de l'expérimentation

Afin de préciser les demandes de la communication, qui doivent être examinées sur divers schémas d'interconnexion, nous utiliserons les spécifications d'une application en temps réel, l'encodeur DivX, illustrée dans la Figure 40.

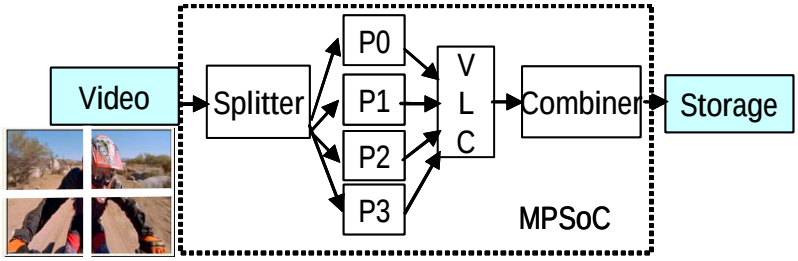


Figure 40. Schéma de l'encodeur DivX

La fonctionnalité de l'encodeur DivX est la suivante. Le module « diviseur » (***Splitter***) reçoit une suite des signaux vidéo non-comprimés (en format **`QCIF`**[13]), compatible au standard **`YUV 4 :1 :1`**[14]. Chaque trame vidéo est découpée en 4 sous-

[13] Quarter Common Intermediate Format
[14] Méthode de codage colorimétrique vidéo : Luminance (Y) - Chrominance et Saturation (U-V).

trames qui sont envoyés à 4 modules de codage parallèles (***P0 – P3***). Les données traitées par chacun de ces modules de codage sont alors expédiées à un module de codage de longueur variable (***VLC***[15]) qui effectue un certain codage additionnel. La trame entière sera reconstituée par le module « combinateur » (***Combiner***) avant d'être envoyée au module de stockage (***Storage***) [Han 04].

La mise en place de l'expérimentation est illustrée en Figure 41. L'application est spécifiée en SystemC, et avec des éléments de l'architecture virtuelle (VADeL) afin d'être représentée sur la plateforme d'évaluation. L'interconnexion, est aussi spécifiée au niveau TLM (en SystemC). L'application et l'interconnexion sont annotées avec les temps d'exécution évalués pour l'architecture ciblée.

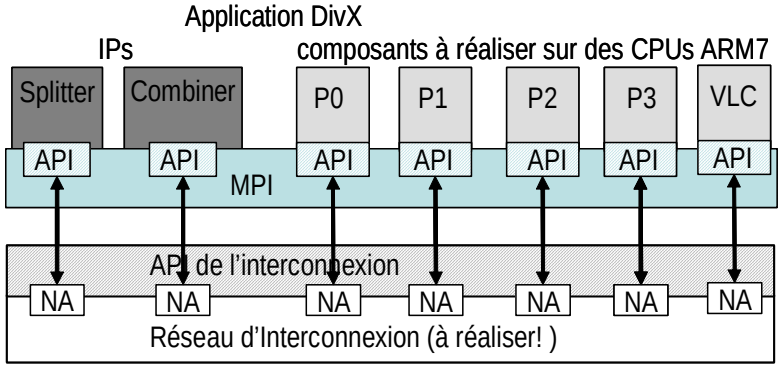


Figure 41. Plateforme MPSoC de l'application DivX pour l'exploration des architectures

L'application comprend en total 12000 lignes de code C. Le mapping proposé est le suivant :

- Les modules « diviseur » (***Splitter***) et « combinateur » (***Combiner***) seront implémentés sur des circuits spécifiques, en matériel (IPs).
- Tous les quatre modules de codage (***P0-P3***) et le module ***VLC*** sont implémentés sur des CPUs ARM7. Des annotations de temps seront introduites dans la description de ces composants, au niveau de chaque fonction et ils correspondent aux temps d'exécution sur le CPU ARM7.

[15] Variable Length Coding

- L'interconnexion qui a été fournie pour cette application est le réseau ***DMS***. Ensuite, dans nos expérimentations, on la substituera avec les autres réseaux disponibles dans les bibliothèques : le bus AMBA et le réseau-sur-puce Octagon.

Le cas de test est le codage d'un film, de **`125 trames`** à une résolution des **`352x288 pixels`**, à **`25 trames par seconde`** et à une fréquence de **`60 MHz`**. Le deadline d'exécution pour chaque trame étant de **`40 ms`**.

4.2 Différentes architectures d'interconnexions pour le codeur DivX

Cette section présente les différentes architectures obtenues par la substitution du réseau DMS de la spécification (qui nous a été fournie) pour le codeur DivX. Ainsi, on va développer encore deux architectures différentes : DivX communicant par un bus AMBA et DivX communicant par un réseau-sur-puce (Octagon).

Notre contribution consiste en le développement de la plateforme d'évaluation globale, contenant les adaptateurs flexibles et automatisables (qui peuvent être générés automatiquement, éventuellement pour d'autres configurations futures). En cette section, on détaille chaque « phase » d'adaptation (définies en Chapitre IV.3.3.2) pour le bus AMBA et pour le réseau Octagon.

4.2.1 Le mapping de l'application DivX sur le bus AMBA

Dans le cas où un bus AMBA est utilisé pour gérer la communication entre tous les composants de sur-puce, ***la première phase de l'adaptation*** est la définition des modules « maîtres » ou « esclaves ». En analysant le comportement fonctionnel de l'encodeur de DivX, on définie la dépendance « maître/esclave » suivante :

- le **`Splitter`** est « maître » en relation avec chaque module de codage (**`P0-P3`**);
- le **`Combiner`** est « maître » en relation avec le **`VLC`**;
- le **`VLC`** est « esclave » en relation avec le **`Combiner`**;
- chaque module de codage (**`P0-P3`**) est « esclave » en relation avec les **`Splitter`** et **`VLC`**.

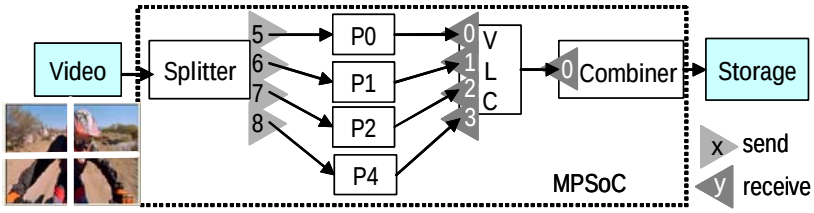


Figure 42. Le mapping de l'application DivX sur le bus AMBA

Table 9. La table de mapping d'adresses de l'application DivX sur le bus AMBA

NUMERO MASTER	CANAL LOCAL (LCH)	ADRESSE (EN HEXA)	DESTINATION
0 (SPLITTER)	5	1H	P0
	6	3335H	P1
	7	6668H	P2
	8	A001H	P3
1 (VLC)	0	1H	P0
	1	3335H	P1
	2	6668H	P2
	3	A001H	P3
2 (COMBINER)	0	D001H	VLC

La deuxième phase de l'adaptation est la translation des adresses. Un exemple de translation d'adresses réalisé en mappant l'identificateur de canal local « **`lch`** » à une adresse dans l'espace réservé pour le module « esclave » correspondant est illustré dans la Figure 42 et respectivement Table 9.

Premièrement, on déduit le système d'adressage du DMS à base des identificateurs « **`lch`** » des ports reliés à l'interface **`MSAP`**. Par exemple, le **`Splitter`** demandera une opération d'écriture à **`P0`**, en appelant la primitive **`mpi_send()`** avec la valeur de « **`lch`** » de **`5`**. Le **`VLC`** demandera une opération de lecture de **`P0`** en appelant la primitive **`mpi_recv()`** avec la valeur du « **`lch`** » de **`0`**. Les identificateurs « **`lch`** » liées aux opérations de lecture/écriture sont identifiés sur la Figure 42.

Seulement les identificateurs de canal « **`lch`** » utilisés par les modules « maîtres » doivent être translatés. Ceci parce que dans la communication avec le bus AMBA seulement les modules « maîtres » demandent les transferts de données, alors que les modules « esclaves » répondent seulement à ces demandes.

L'espace d'adresse du bus AMBA doit être partagée entre 5 modules « esclaves »: 4 processeurs et le VLC – qui est esclave en relation avec le module Combiner. Pour le modèle de simulation pour le bus AMBA, on a considère un espace d'adresse virtuelle de 16 bits, dans la gamme **`0H-FFFFH`** (pratiquement, il est défini dans un tableau fictif, pour la simulation). D'ici on fixe l'adresse en hexa correspondante à la destination.

La troisième phase sert à formater la donnée en paquets de 32 bits, chacun précédé par un en-tête contenant des signaux spécifiques au bus, comme l'on a montré en Chapitre IV.3.3.

Le modèle exécutable de l'encodeur DivX avec une interconnexion à base de bus AMBA est illustré dans la Figure 43. La communication entre les modules de l'encodeur est réalisée toujours en utilisant l'API de MSAP : « **`mpi_send/mpi_recv`** ». Par ces fonctions, l'application demande des services du bus AMBA, qui répond en offrant ces services via son propre API : « **`single_write()/single_read()`** ». Les différents adaptateurs sont illustrés : adaptateurs « maîtres », adaptateurs « esclaves », et aussi des interfaces d'adaptation qui accomplissent les deux tâches et qu'on a nommé adaptateurs « maîtres/esclaves ».

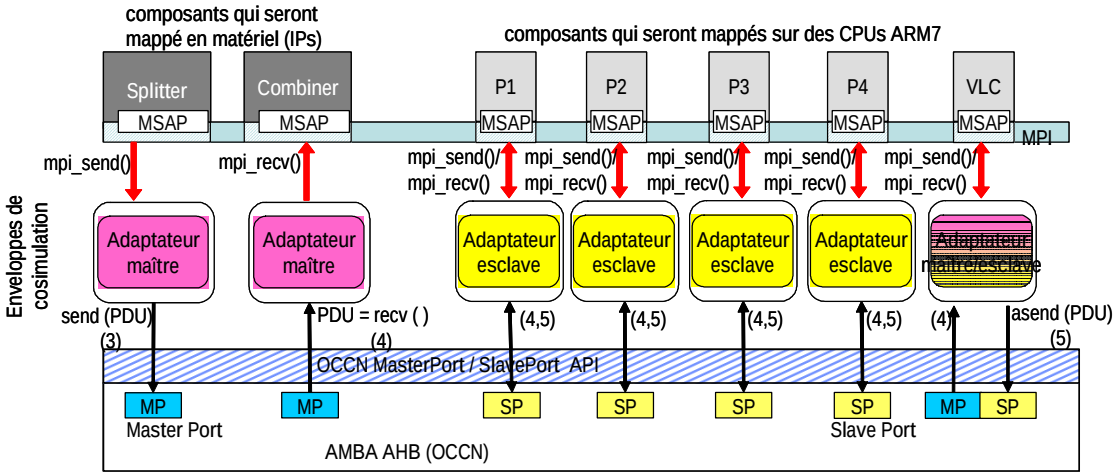


Figure 43. Le modèle exécutable MPSoC de l'application DivX, en ayant comme interconnexion le bus AMBA

Les adaptateurs sont programmés en SystemC, et ils comprennent environ 1000 lignes de code, chacun, en incluant les fonctions pour collectionner les statistiques utilisées pour l'évaluation des performances. Ils traduisent les primitives « **`mpi_send/mpi_recv`** » de l'application en des actions compatibles aux différents

« ports Maîtres » ou « ports Esclaves » auxquels ils sont connectés. Des interfaces différentes sont nécessaires pour les modules qui sont maîtres, esclaves et maîtres/esclaves, à cause de la non-homogénéité du protocole pour les primitives de « **`Read`**/**`Write`** » sur un port maître et esclave respectivement. La disposition des adaptateurs maîtres, esclaves et maître/esclaves est montrée dans la Figure 43.

4.2.2 Le mapping de l'application sur le réseau-sur-puce Octagon

Dans le cas des schémas d'interconnexion à base de DMS et bus AMBA, les modules de l'application DivX sont reliés d'une manière symétrique : le DMS utilise des interconnexions point-à-point pour les transferts de données alors que AMBA est un environnement partagé. A son tour, le réseau-sur-puce Octagon accepte plusieurs possibilités de mapping pour l'application, avec un impact différent sur la performance de l'exécution, selon le modèle de trafic.

Le mapping est ***la première phase*** de notre schéma d'adaptation, et il est réalisé comme suit. L'idéal serait de séparer les modules **`Splitter-P0-P1-P2-P3`**, **`P0-P1-P2-P3-VLC`** et **`VLC-Combiner`** par un seul hop[16]. Cette organisation n'étant pas possible, divers mappings doivent être examinés pour trouver un optimum. La Figure 44 montre le mapping utilisé dans notre exemple.

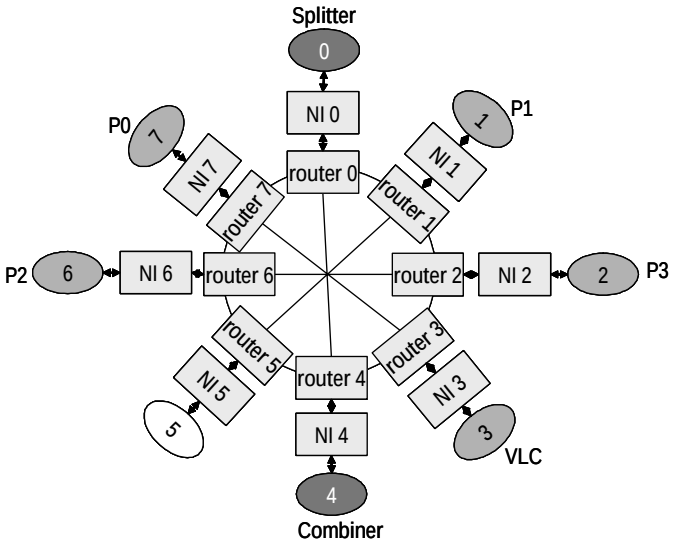


Figure 44. Le mapping des modules du DivX sur le réseau Octagon

[16] nombre de noeuds intermédiaires de routage

Apres le mapping, ***la deuxième phase de l'adaptation*** est la translation d'adresses entre DMS et Octagon. A partir du système d'adressage de DMS (Figure 45), le mapping interprète dans ce cas-ci l'identificateur « **`lch`** » liée à une opération d'envoi dans une identification de destination dans la gamme **`0-7`** (Table 10).

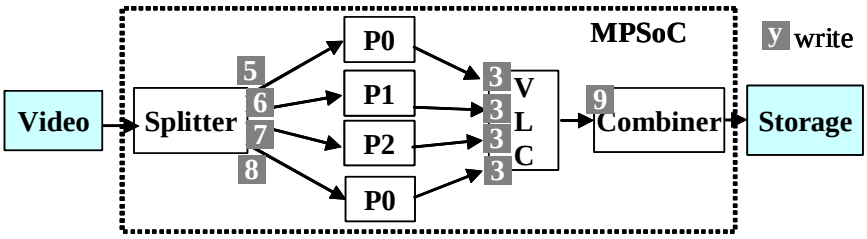


Figure 45. Le mapping de l'application DivX sur le réseau Octagon

Table 10. La table de mapping d'adresses de l'application DivX sur le réseau Octagon

MODULE EMETTEUR	CANAL LOCAL (LCH)	IDENTIFICATEUR DE DESTINATION (DESTID)	DESTINATION
SPLITTER	5	7	P0
	6	1	P1
	7	6	P2
	8	2	P3
P0	3	3	VLC
P1	3	3	
P2	3	3	
P3	3	3	
VLC	0	4	Combiner

La troisième phase sert à formater la donnée en paquets de 32 bits, chacun précédé par un en-tête contenant les identificateurs de la source et de la destination, ainsi que la taille du message, comme l'on a montré en Chapitre IV.3.3.

Le modèle exécutable de l'encodeur DivX avec l'interconnexion Octagon est illustré dans la Figure 46. Les demandes de services de l'application via son API **`mpi_send()/mpi_recv()`** sont traités dans les offres des services de la part du

réseau d'interconnexion, via l'API de ce dernier : **`send()/recv()`**. Des interfaces d'adaptation seront générées pour chaque module.

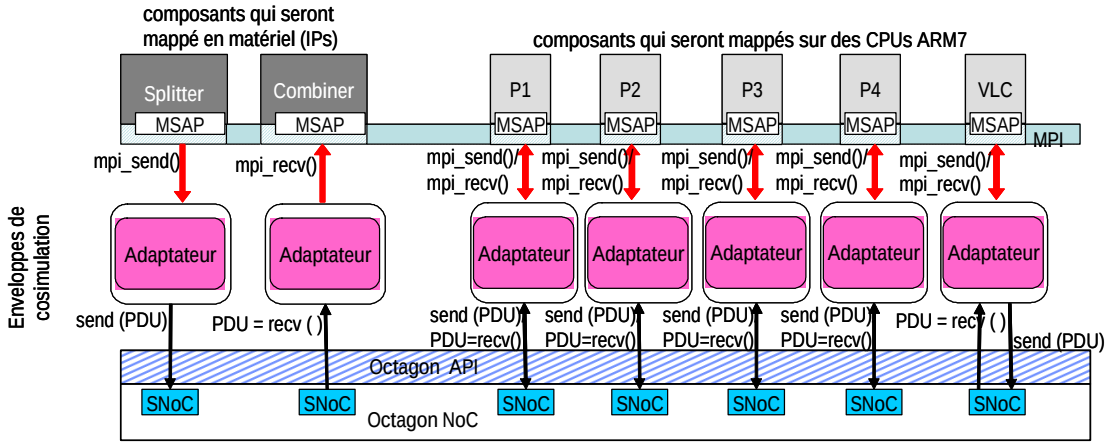


Figure 46. Le modèle exécutable de l'application DivX avec l'interconnexion le réseau Octagon.

Les adaptateurs sont programmés en SystemC, et ils comprennent environ 120 lignes de code, chacun, en incluant les fonctions pour collectionner les statistiques utilisées pour l'évaluation des performances. Ces adaptateurs sont plus simples, car il n'y a que deux fonctions d'accès à l'Octagon (**`send()`** et **`receive()`**), par rapport au bus AMBA où on peut avoir plusieurs types de « **`read`**/**`write`** » (**`single_write`**, **`single_write_leaving`**, **`incremental_write`**, **`burst_write`**, etc. ; dans nos expérimentations on n'a utilisé que les deux premiers).

Les interfaces d'adaptation sont homogènes pour le cas de mapping sur le réseau Octagon, car le réseau et le mapping sont symétriques. Par contre, on a introduit des dissymétries par la collection des statistiques que l'on a réalisé différemment pour les modules qui initient la communication et pour ceux qu'ils acceptent la communication.

4.3 Evaluation des performances et résultats obtenus

Cette section présente les résultats expérimentaux obtenus suite à l'étude comparative des trois architectures différentes pour la réalisation de l'application DivX. Les architectures ne diffèrent que par le sous-système d'interconnexion utilisé : DMS, le bus AMBA et le réseau-sur-puce Octagon.

Afin de faire des comparaisons de performance et des propositions d'optimisation, les indicateurs de performance étudiés sont ***le temps total d'exécution***, ***le débit moyen*** et

la latence des primitives **`Send()/Receive()`**. Chacune de ces indicateurs a été calculé pour les modules qui jouent le rôle d'initiateurs de communication: ***Splitter***, ***VLC*** et ***Combiner***. Ces modules génèrent du trafic sur le réseau d'interconnexion, en exécutant des opérations lecture/écriture.

4.3.1 Comparaison de temps d'exécution

L'application DivX comporte des fortes contraintes de temps réel. On considère que le code est déjà partitionné et mappé sur des processeurs, satisfaisant les contraintes de temps-réel ([Bona 06]). De la même manière, il faut montrer que l'interconnexion répond aussi aux contraintes de temps-réel (ce que l'on veut explorer dans cette thèse).

Dans nos expérimentations, on dispose de modèles d'exécution précis pour les sous-systèmes d'interconnexion (Chapitre IV.1.3). Les annotations temporelles sont également disponibles pour les tâches de l'application, afin de valider la fonctionnalité globale (Chapitre IV.1.2).

Table 11. Comparaison de temps d'exécution (en nombre de cycles d'horloge)

RESEAU D'INTERCONNEXION CONSIDEREE	DMS	AMBA BUS	OCTAGON NOC
TEMPS D'EXECUTION (CYCLES D'HORLOGE)	233,689,205	233,431,813	233,154,449

La Table 11 illustre ***les temps d'exécution*** pour les trois architectures considérées (l'exemple des sections 4.1, 4.2.1 et 4.2.2 du Chapitre IV). Les résultats obtenus sont très similaires. Cela est partiellement à cause de la description architecturale de l'application (format vidéo QCIF, mémoires double-bancs en lecture/écriture). D'un autre côté, c'est à cause de la synchronisation « instantanée » des processeurs (vers leur communication avec le VLC), et par la suite l'utilisation dans un cas « favorable » des enchaînements communications - calculs. Par conséquence, le temps passé dans la communication est très petit par rapport au temps dépensé dans les calculs d'environ 1‰. Le cas le plus défavorable serait où les processeurs finissent leurs calculs en même temps et ils doivent se synchroniser pour la communication

avec le VLC. Pour ce cas, il a été estimée ([Bona 06]) une communication qui prend dans le pire cas 2% du temps total d'exécution.

Même si les résultats sont très similaires, on ne peut pas utiliser chacune de ces configurations proposées. La raison est la violation des contraintes de temps réel imposées par l'application, comme il est montré dans Chapitre IV.4.3.3.

4.3.2 Comparaison des débits moyens

En utilisant la fonction **`StatInstantThroughput()`** de l'environnement OCCN, le débit est dérivé comme la moyenne cumulative calculée sur des fenêtres consécutives de temps. Le débit est calculé pour chaque appel des primitives d'émission/réception, en utilisant l'échantillonnage pour la période de transmission et la taille des messages transmis.

Le débit rapporte la valeur cumulative des échantillons transmis à la fenêtre de temps. Les valeurs impliquées sont :

$(t_c + \Delta t)$/no_échantillons – pour l'axe temporelle

$(N_{bc} + \Delta N_b)$/no_échantillons – pour l'axe du débit

Où t_c est le temps courant, Δt est le temps écoulé depuis le dernier échantillon, N_{bc} est le nombre de bytes actuellement transmis et ΔN_b est le nombre de bytes ajoutés à la valeur courante.

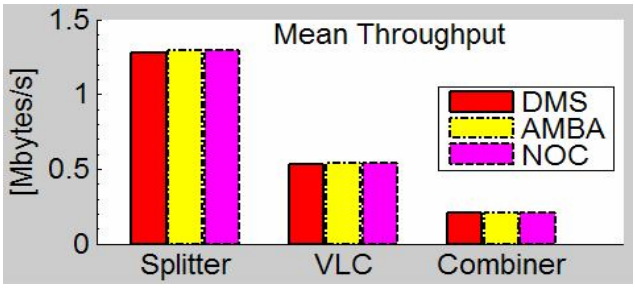


Figure 47. Comparaison des débits moyens pour le Splitter, VLC et Combiner, évalué pour les trois architectures : avec l'interconnexion DMS, AMBA et Octagon.

En notre cas, nous obtenons approximativement les mêmes débits pour les trois architectures (Figure 47). Aucune de ces valeurs n'atteint son maximum, indiquant qu'aucune des interconnexions ne sature.

Bien que les valeurs moyennes du débit soient très proches, on verra plus tard que ceci peut cacher la violation des contraintes de temps réel (Chapitre IV.4.3.3).

4.3.3 Comparaison des latences de communication

Finalement, la latence de communication peut prédire si l'interconnexion est adaptée à l'application. On a vu que si les latences excèdent l'intervalle de temps réel, les contraintes de temps réel imposées à l'application sont violées.

Dans notre étude, on évalue **la latence moyenne**. La latence moyenne est calculée comme la valeur moyenne du délai de communication évalué d'une extrémité à l'autre pour différentes sections critiques des primitives émetteurs-récepteurs. Elle est acquise en utilisant la fonction OCCN **`StatDelay()`**[Cop 03] [Copp 03].

Dans notre cas, ***l'intervalle de temps réel*** est le temps entre deux réceptions de trames consécutives :

`RT = 40 ms`.

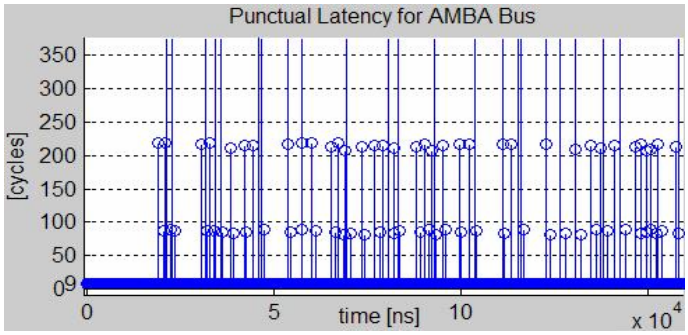
La latence du bus AMBA (Figure 48 (a)) montre que, en moyenne, le bus peut manipuler les demandes de l'application: la latence du transfert entre le **`Splitter`** et les modules de codage **`P0-P3`** est de **`9 cycles`**. Cependant, des conflits sont détectés sur le bus. Les valeurs très grands de la latence, illustrées dans la même figure, indiquent un fonctionnement singulier de l'application: les contraintes de temps réel ne sont pas satisfaites.

Cependant, ces valeurs n'ont pas pu pas être détectées par les valeurs du débit moyen. Elles sont « noyées » dans la moyenne, à cause de leur poids qui est faible. La moyenne est autour de 11 cycles.

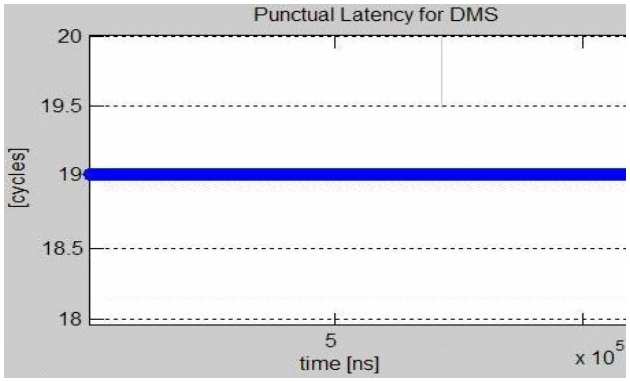
À cause des contraintes de conception, le flux de données provenant de l'Antenne n'est pas bufférisé et il ne peut pas être bloqué. Ce fait aurait impliqué une augmentation de la mémoire embarquée sur puce. Dans ce cas, les contraintes de conception en termes de surface sont violées. Par la suite, tous les calculs et transfert de données devraient être réalisés dans cet intervalle de 40 ms.

La latence par transaction (c.-à-d. MPI **`Send()/Receive()`**) pour le sous-système d'interconnexion DMS a été évaluée par exemple entre le **`Splitter`** et les modules

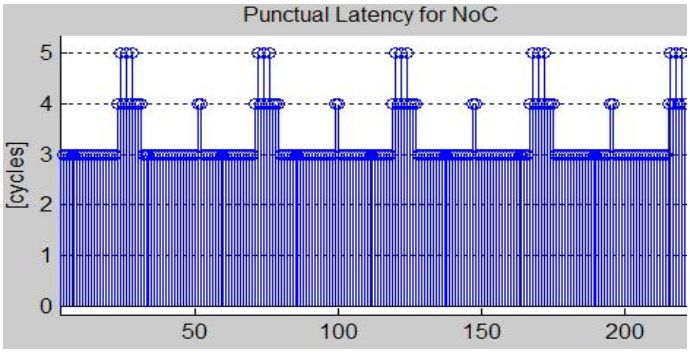
de codage **`P0-P3`**, à **`19 cycles`** constants tout au cours de la simulation. Cela est dû à la communication avec DMS qui utilise une interconnexion point-à-point, sans conflits.



(a)



(b)



(c)

Figure 48. Latence ponctuelle du (a) bus AMBA (b) réseau-sur-puce Octagon

La latence des primitives **`Send()/Receive()`** pour le réseau-sur-puce Octagon est illustrée dans la Figure 48 (b). Les valeurs varient entre **`3`** et **`5 cycles`**. Ils sont inférieures comparées aux latences obtenus avec autres interconnexions. Ceci est dû à la réalisation matérielle de type ASIC des routeurs du réseau. Un autre facteur important est la conception du réseau pour éviter la congestion.

5 Appréciation de la méthode d'évaluation de performances par composition

A part une évaluation globale du système MPSoC, parmi les trois objectifs majeurs fixés au début de cette thèse, il y avait aussi la précision et la vitesse de la méthode d'évaluation. Ainsi, dans notre approche nous essayons d'accomplir ces deux points majeurs. La précision de l'évaluation des performances par cosimulation dépend des profils d'exécution choisis et de leur synchronisation globale. Les résultats collectionnés suite à la cosimulation, dépendent de la précision des sondes de mesure sur les sections d'intérêt. La vitesse de l'évaluation, qui doit être augmentée, est une combinaison entre la vitesse de la cosimulation de système et le temps d'analyse des résultats d'évaluation.

La méthode d'évaluation de performances par intégration des différents modèles d'évaluation que l'on a proposée a deux qualités de base : l'extensibilité et la flexibilité :

- **L'extensibilité** se manifeste par la capacité d'intégrer plusieurs sous-systèmes différentes, sans demander un grand effort de conception. Ainsi, on peut générer plusieurs architectures différentes par la simple commutation parmi plusieurs réseaux d'interconnexion, ou plusieurs applications. L'extensibilité est une qualité inhérente, déterminé par l'abstraction du modèle. Aussi, elle ne doit pas impliquer l'augmentation de la complexité. La complexité est une fonction directement proportionnelle avec le nombre de simulateurs mis en parallèle, le taux de communication entre simulateurs et leur synchronisation.
- **La flexibilité** se retrouve dans une exploration de différents modèles architecturaux sans modifier l'application ou l'interconnexion. Seulement l'interface d'adaptation sera ajustée. Ainsi, l'ajout d'un nouveau réseau d'interconnexion dans les bibliothèques nécessitera simplement la spécification de l'interface d'adaptation correspondante entre son API et l'API de l'application. De même, l'ajout des nouveaux protocoles de

communication pour les applications impliquera uniquement leur translation dans l'interface, pour les réseaux existants.

Perspectives

La méthode proposée en ce chapitre pour l'évaluation des performances des MPSoCs, proposée dans ce chapitre, est basée sur la composition de différents modèles d'évaluation, pour les différents sous-systèmes est présentée d'une manière illustrative qui démontre la faisabilité de notre approche. Plusieurs améliorations au niveau local peuvent être réalisés, comme suit :

(a) l'automatisation complète de la conception. Cela implique la génération complètement automatique des interfaces d'adaptation et aussi l'extraction automatique des paramètres à partir du modèle de description. Nous avons proposé l'insertion les éléments des interfaces d'adaptation dans des bibliothèques de conception. De plus, il faut développer des procédures automatiques de paramétrisation et sélection optimisés des bibliothèques.

(b) la réalisation matérielle et/ou logicielle des interfaces d'adaptation. D'abord il faut établir la manière de réalisation : par évaluation des coûts en vitesse, flexibilité, et autres métriques significatives pour la conception. Ensuite il faut proposer la structure appropriée.

(c) l'enrichissement des bibliothèques des interfaces d'adaptation avec des autres APIs de communication pour l'application et pour l'interconnexion.

(d) l'extension de la méthode pour l'exploration des autres sous-systèmes que l'interconnexion. Cette extension est immédiate : il faut prévoir des interfaces d'adaptation et des méthodes pour collectionner des métriques à partir d'autres structures que l'interconnexion. Le cadre d'évaluation des performances (la méthode basée sur la cosimulation et la génération des interfaces d'adaptation) est flexible et modulaire, permettant ce type d'extensions.

(e) la connexion avec un processus d'optimisation. Cela est nécessaire pour modifier les paramètres critiques selon les directions indiqués par l'évaluation des performances.

(f) varier les métriques inspectées et les études de cas explorés. Dans nos expérimentations, on a inspecté des métriques illustratives. Les futures améliorations pourraient être de rajouter des métriques et d'inspecter plusieurs configurations, en vue de trouver les corrélations entre les valeurs des paramètres mesurés et les schémas considérés.

(g) la modélisation formelle du système. Les prédictions montrent que le but final des méthodologies d'évaluation de performances est de développer des approches analytiques pour pouvoir analyser le système à un niveau élevé [Itrs]. Ainsi, on pourra développer une méthode analytique pour améliorer de vitesse d'évaluation des performances. L'objectif est de trouver rapidement, à partir des phases préliminaires de la conception, les composants qui sont des goulots d'étranglement.

L'idée de base est d'employer la cosimulation pour évaluer une solution architecturale et annoter à posteriori un modèle analytique proposé. Ainsi, le modèle analytique est construit par caractérisation des composants et l'extraction des paramètres à partir du modèle de cosimulation. Plusieurs cycles d'annotation à posteriori sont nécessaires pour un modèle analytique correct. Ensuite, l'évaluation des performances à base d'un modèle analytique devient une étape autonome, qui prévoit rapidement les performances pour un nombre massif de nouvelles architectures conçues.

La théorie des files d'attente (QT[17]) est de grand intérêt pour les systèmes de calcul et les réseaux de transmission. C'est la méthode formelle qui regarde certains composants du système (par exemple processeurs, interconnexions, etc.) comme centres fournissant de services et des files d'attente à ces services. De plus, la théorie des files d'attente fournit l'ensemble d'approches et de formules pour résoudre analytiquement certains systèmes simples en obtenant leurs paramètres de performance. La performance est recherchée par l'étude des délais, des débits, des utilisations, etc., dans différents endroits du système ainsi modélisé [Kle 75] [Kle 76]. Cette théorie est une réalisation envisageable.

Les qualités désirés pour le modèle analytique sont : la généricité, l'extensibilité, la flexibilité, la facilité à utiliser et à le générer automatiquement. Dans notre approche proposée, on est capable de fournir ces qualités, en se basant sur les démarches

[17] Queuing Theory

suivantes : (1) la conception hiérarchique, conforme à la réalisation finale ; (2) la possibilité de fixer automatiquement les paramètres à partir de la description du système et des résultats de simulation de l'architecture réelle ; (3) la facilité de configuration de ce modèle, en vue de l'évaluation de ses performances et d'exploration des architectures.

(h) la composition des modèles de simulation avec des modèles analytiques. La méthode proposée basée sur la composition de différents modèles d'évaluation, pourra combiner également la simulation et des modèles analytiques. L'objectif qui dérive de notre méthode d'évaluation des performances par composition, est d'appliquer la stratégie d'évaluation la plus adaptée selon les besoins de chaque sous-système composant.

Par suite, l'objectif à long terme serait de développer un cadre générique pour l'exploration et l'optimisation de l'espace de conception, où différentes méthodes d'évaluation basées sur simulation ou analyse pourraient être appliquées aux différents sous-systèmes et à différents niveaux d'abstraction.

Conclusions

Notre contribution majeure, présentée en ce chapitre, est une solution flexible pour l'évaluation globale des performances pour les MPSoC. Elle est basée sur la composition de différents outils dans un environnement de cosimulation. Le modèle réunit tous les composants du système, représentés par leurs modèles d'exécution. Leur connexion est réalisée via des interfaces d'adaptation qui agissent comme des interfaces de cosimulation. La construction des interfaces d'adaptation est la deuxième contribution. La flexibilité de l'approche est donnée par les interfaces d'adaptation. Ils permettent des associations de différents modèles de sous-systèmes de différents réseaux d'interconnexion. L'applicabilité de notre approche dans l'exploration des architectures est évidente.

Cette méthode a été appliquée dans le cadre du flot de conception ROSES pour trouver l'interconnexion optimale pour l'application DivX. Elle permet la génération des différents modèles de simulation et de choisir celui qui correspond aux contraintes et qui a des meilleures performances.

Les perspectives exposent la possibilité d'incorporer un modèle analytique dans ce modèle global d'évaluation des performances. Un prototype pour le sous-système Octagon a été proposé, en prouvant la faisabilité de cette démarche.

Conclusions et perspectives

Conclusions

Les tendances actuelles de l'évaluation des performances pour les MPSoC vont vers l'assemblage des méthodes existantes pour les sous-systèmes individuels. Les difficultés rencontrées dans la conception d'une telle méthode sont : (1) considérer l'effet global de chaque sous-système individuel ; (2) décrire dans un environnement unifié les divers profils d'exécution des sous-systèmes, et leurs interactions ; (3) interconnecter tous ces sous-systèmes communiquant différemment ; (4) de répondre aux différents besoins dans l'évaluation de performances des concepteurs/utilisateurs et indiquer des directions d'optimisation ; (5) d'accélérer l'évaluation afin d'éviter une étape trop longue dans le flot de conception.

Dans ce travail nous avons abordé deux principaux problèmes de l'évaluation des performances : (a) la conception d'une méthode globale et flexible, capable de réunir différents méthodologies pour les sous-systèmes individuels et (b) la conception d'un modèle rapide et précis d'évaluation des performances pour le sous-système logiciel embarqué.

Le Chapitre I a présenté le cadre de conception ROSES qui nous a servi de contexte pour concevoir l'outil d'évaluation des performances. ROSES est une méthodologie au niveau système pour la conception des MPSoC, développée au sein du groupe SLS. Ce chapitre présente les éléments de base intervenant dans le flot de conception ROSES : le modèle de description des systèmes MPSoC, basé sur l'architecture virtuelle et le modèle de validation des systèmes, basé sur la cosimulation logicielle/matérielle. Ensuite, on présente le flot global de conception, basé sur ces deux éléments de base (description/validation), mais aussi sur la génération automatique des interfaces de cosimulation, interfaces logicielles et matérielles, à

partir des bibliothèques des interfaces. Toutes ces étapes interviendront dans notre modèle d'évaluation des performances par composition. Un autre point important de ce chapitre est la présentation détaillée des deux sous-systèmes « critiques » dans l'évaluation des performances : (1) le sous-système logiciel et (2) le sous-système d'interconnexion.

Le Chapitre II a présenté une étude synthétique sur l'évaluation des performances des différents sous-systèmes embarqués (matériels, CPU, logiciels et d'interconnexion), et des MPSoCs. La contribution majeure de ce chapitre est la définition d'un cadre de description, où l'on peut étudier méthodiquement différentes méthodologies et besoins en évaluation des performances pour les sous-systèmes composants du MPSoC. Premièrement, on définit les métriques utiles pour l'évaluation des performances, et intéressantes pour les éventuels processus d'optimisation réalisés après l'évaluation. Les paramètres de conception sont présentés ensuite ; ils sont les paramètres de la fonction que l'on a dénommé « métrique », et ils pourront être taillés dans un éventuel processus d'optimisation à base des résultats de performances. Ensuite, on présente les concepts structurels de base et les niveaux d'abstraction utilisés au cours de la conception des systèmes embarqués. Ce chapitre examine subséquemment la variation des concepts de base à travers les différents sous-systèmes composants du MPSoC : matériels, logiciels, CPU et l'interconnexion. Chaque sous-système sera présenté par une courte définition, ses niveaux d'abstraction, ses métriques et paramètres de conception. A la lumière de ces concepts, nous avons analysé l'état de l'art pour l'évaluation de performances des MPSoC. Cette étude a mis les fondations pour notre méthode globale et flexible d'évaluation des performances des MPSoCs par composition.

Le Chapitre III a présenté le développement d'un modèle de simulation pour le sous-système logiciel, incluant le système d'exploitation embarqué. Ses bénéfices majeurs sont (1) la définition d'un modèle indépendant de l'architecture cible, (2) l'annotation des temps d'exécution, spécifiques à l'architecture cible et (3) la prise en compte des interactions avec tous les autre sous-systèmes du MPSoC. Le modèle de simulation proposé pour le sous-système logiciel est à la fois rapide et précis. La vitesse de l'évaluation est donnée par l'exécution native utilisée, et la précision est donnée grâce à la cohérence avec la réalisation finale du SE et de l'inclusion de ce modèle dans un schéma de cosimulation logicielle/matérielle. Une autre contribution de ce chapitre est

la conception de l'interface de cosimulation TBFM qui considère les interactions avec les modules matériels, les interruptions matérielles, le traitement des entrées/sorties, tout en tenant compte de l'avancement du temps des sous-systèmes logiciels. L'outil que nous avons développé pour mettre en pratique cette méthode est nommé « ChronoSym ». Il offre la possibilité de mesurer les performances du sous-système logiciel, en incluant le SE, au niveau temps d'exécution.

Le Chapitre IV a présenté la conception de notre méthode d'évaluation des performances pour les MPSoCs. Notre première contribution de ce chapitre est la définition de la méthode basée sur la composition de différents outils d'évaluation des sous-systèmes singuliers, dans un environnement global de cosimulation. La deuxième contribution est la définition et la génération des interfaces flexibles qui interconnectent les différents modèles d'évaluation. La mise en pratique de cette méthode a visé surtout l'évaluation les performances d'une application avec différentes structures d'interconnexion. Une perspective intéressante de ce travail est de remplacer le modèle de simulation du réseau d'interconnexion par un modèle analytique.

Perspectives

Notre contribution dans ce travail est la conception d'un environnement générique et flexible pour l'évaluation des temps d'exécution des MPSoCs. En offrant une haute vitesse de simulation et une bonne précision, cet environnement permet la manipulation des grands systèmes MPSoC.

1. Il est important de poursuive cette étude en évaluant aussi des autres métriques comme la surface occupée sur puce et l'énergie consommée. Ces métriques pourront unifier notre méthode dans un cadre d'optimisation et exploration d'architectures, des choix de conception ou du partitionnement logiciel/matériel.

2. L'exploration des architectures a été testée pour les sous-systèmes d'interconnexion dans le Chapitre IV. Mais elle pourrait être étendue pour les autres sous-systèmes. Il y a deux aspects à regarder : (a) la variation de plusieurs sous-systèmes et (b) la modification de leurs paramètres. Par exemple l'outil **`ChronoSym`** pourrait explorer

les sous-systèmes logiciels, en variant l'architecture cible, ou explorer les paramètres des systèmes d'exploitation embarqués en variant la réalisation de différents services.

3. De plus, sa complexité et les aspects dynamiques rendent le sous-système logiciel embarqué difficile à concevoir, et encore plus difficile à analyser. Comme une perspective, il serait intéressant d'envisager la description des sous-systèmes logiciels comme des modèles mixtes : par exemple associant des graphes de flot de données avec la simulation et les statistiques.

4. L'autre sous-système critique considéré dans cette étude est le sous-système d'interconnexion. A part les délais de transmission et l'aspect de consommation d'énergie, l'intégrité des données et la fiabilité des communications sont d'un grand intérêt. Il serait intéressant d'introduire dans la spécification de ce système des aspects liés à la qualité des services. Cela est une conséquence de la nécessité de prédictibilité du trafic par le réseau embarqué. La qualité des services est très fortement couplée avec les aspects d'évaluation des performances.

5. Un autre aspect de grand intérêt pour les sous-systèmes d'interconnexion serait de développer son modèle analytique équivalent (voir les Perspectives du Chapitre IV). Ce modèle serait capable de prédire instantanément les performances de ce sous-système, en étant très utile dans les phases de sélection des interconnexions ou de choix des paramètres.

6. Une perspective attractive du point de vue de la vitesse et la précision de l'évaluation des performances serait de construire une plateforme d'émulation sur FPGA pour le système MPSoC. Cela offrirait plusieurs degrés de liberté et une bonne vitesse d'émulation, nécessaires pour tester différents cas de conception.

7. D'une manière générale, il serait intéressant de définir un environnement formel pour la conception concurrente logicielle/matérielle. L'augmentation du niveau d'abstraction de la conception serait la première étape de la mise en œuvre de ce concept. Aussi, des spécifications homogènes pour tous les sous-systèmes pourront abstraire la réalisation finale.

Cet environnement formel offrirait plusieurs degrés de liberté aux concepteurs et aux outils automatiques de conception. Le modèle suggéré a une grande flexibilité, une

bonne capacité de réutilisation des composants et de l'amélioration globale des performances. Ce modèle est une solution pour les problèmes majeurs de conception comme : les décisions et les compromis de conception, le partitionnement logiciel/matériel, l'exploration des architectures et les optimisations.

8. Le partitionnement logiciel/matériel est une étape avancée du flot de conception. Pour choisir le partitionnement, il est indispensable de bénéficier des évaluations les performances très tôt dans le flot de conception. Ainsi, les estimations de haut niveau sont nécessaires pour intervenir d'une manière efficace dans l'étape de partitionnement. Aussi, ces estimations de haut niveau doivent être corrélés avec les paramètres de l'architecture, qui sont extrait à un bas niveau.

Glossaire

A

AMBA AHB	**AMBA, Advanced High-Performance Bus** spécification (standard) de bus embarqué sur puce, qui couvre une stratégie d'interconnexion et la gestion des blocs fonctionnels constituants d'un SoC.
API	**Application Programming Interface** Ensemble de routines standard destinées à faciliter au programmeur le développement d'applications
ARM	**Advanced Risc Machine** Disponible à l'adresse : **http://www.arm.com/**
ASIC	**Application Specific Integrated Circuit** Circuit intégré développé Spécifiquement pour une Application

B

benchmark	un benchmark est un programme considéré comme point de référence pour une mesure
BFM	**Bus Functional Model** Interface pour la simulation, permettant de transformer les accès mémoire fonctionnels en des accès mémoire cycle-près
bridge	ponts de connexion
buffer	zone de mémoire tampon
burst	transmission en rafale

C

CA	**Channel Adapter** Composant élémentaire d'une interface de cosimulation
CASE	**Computer-Aided Software Engineering** Outils de développement logiciel assistés par l'ordinateur.
CPU	**Central Processing Unit** Partie principale d'un système processeur, réservée aux traitements
CPSR	**Current Process Status Register** le registre contenant le statut courrant
cross-compiling	compilation croisée

D

data abort	arrêt de transaction à cause d'une exception
DMS	**Distributed Memory Server** serveur de mémoire distribué
DSP	**Digital Signal Processor** Processeur spécialisé pour le calcul d'algorithmes de traitement de signal

E

earliest-deadline first	Algorithme d'ordonnancement à priorité dynamique. Une tâche est d'autant plus prioritaire que sa date d'échéance absolue est proche de la date courante.
EDA	**Electronic Design Automation** l'automatisation de la conception des systèmes électroniques
E/S	entrées/sorties, en Anglais I/O (input/output)

F

FIFO	**First In First Out** Classe ou protocole de communication qui assure que les premières données envoyées sont les premières reçues
FIQ	**Fast Interrupt Request** FIQ est une interruption de priorité plus élevé que IRQ, détenant plus de registres disponibles dans le mode FIQ (ceci réduit le surcoût de changement de contexte).
flit	**flow control digit** la plus petite subdivision d'un paquet qui peut être transporté dans le réseau
FPGA	**Field Programmable Gate Array** Réseau de portes logiques qui est destiné à être programmé par l'utilisateur, avant d'être utilisé pour une fonction particulière
FSM	**Finite State Machine** Modèle de calcul défini par en un ensemble d'états, un ensemble d'événements d'entrée, un ensemble d'événements de sortie et une fonction de transition entre les états

H

HAL	**Hardware Abstraction Layer** La couche basse de l'organisation du logiciel fournissant les pilotes et les contrôleurs pour la gestion de la communication
HDL	**Hardware Description Language** Catégorie des langages utilisée pour la spécification des systèmes matériels
HdS	**Hardware dependent Software** Logiciel dépendant du matériel sur lequel il s'exécute – coïncide avec la couche basse de l'organisation du logiciel,
HLL	**High-Level Language** langage de haut niveau
hop	nombre de noeuds intermédiaires de routage

I

interrupt disables	désactivation des interruptions
IP	**Intellectual Property** Élément (logiciel ou matériel) dont le fonctionnement est connu et documenté, mais dont la structure interne est inconnue
IPC	**Inter-Process Communication** Echange de données entre des processus s'exécutant sur une même machine où

	sur des machines différentes
IRQ	**Interrupt Request** requête d'interruption I/O
ISA	**Instruction Set Architecture** Niveau d'abstraction pour le logiciel simulant l'architecture du jeux des instructions, avec la précision du cycle d'horloge
ISS	**Instruction Set Simulator** Outil qui s'exécute sur la machine hôte et qui émule la fonctionnalité d'un processeur

K

kernel	noyau du SE

M

master/slave	protocole « maître/esclave »
message passing	communication par passage des messages
MIPS	**Mega Instructions per Second** Million d'Instructions par Seconde
MPI	**Message Passing Interface** MPI est une bibliothèque contenant l'interface d'échange de messages, proposé comme norme par des fournisseurs, concepteurs, et utilisateurs
MPSoC	**Multi-Processor System-on-Chip** système multiprocesseur monopuce
MSAP	**Memory Service Access Point** point d'accès à la mémoire

N

NA	**Network Access** point d'accès au réseau
NI	**Network Interface** interface de réseau
NoC	**Network-on-Chip** réseau-sur-puce

O

OCCN	**On-Chip Communication Network** réseau de communication embarquée sur puce

P

PA	**Processor Adapter** Composant élémentaire dans la structure de l'interface de cosimulation. Ce composant est spécifique au module à intégrer dans la cosimulation

PDF	**Probability Density Function** fonction de densité de probabilité
PDU	**Protocol Data Unit** l'unité de base d'un message
PCI	**Protocol Control Information** la partie contenant les signaux de contrôle d'un message
PE	**Processing Element** élément de calcul
prefetch abort	arrêt de recherche de l'instruction

Q

QCIF	**Quarter Common Intermediate Format** Formats de base standard de codage vidéo. QCIF est un format progressif pour animation ordinateur avec 180x144 pixels par images et 30 images par seconde. La résolution de QCIF équivaut au quart de la résolution de FCIF (Full Intermediate Format).
QT	**Queuing Theory** Théorie des files d'attente.

R

RAM	**Random Access Memory** Mémoire permettant aussi bien la lecture que l'écriture. Elle sert au stockage des données et des programmes qui sont en cours de traitement. Ce type de mémoire est le plus rapide. Elle est réinitialisée après chaque coupure de courant, ou reboot.
RISC	**Reduced Instruction Set Computer** Technologie de microprocesseurs ayant un jeu d'instructions câblées réduit au minimum et dont la plupart des instructions s'exécute en un cycle d'horloge.
ROM	**Read Only Memory** Mémoire uniquement accessible en lecture, impossible à modifier. Une ROM est programmée (son contenu est fixé) pendant sa phase de fabrication.
RT	**Real Time** temps réel
RTL	**Register Transfer Level** Niveau d'abstraction pour la spécification des systèmes : niveau de transfert des registres.

S

SAP	**Service Access Point** point d'accès aux services
SDU	**Service Data Unit** la partie contenant les données d'un message
SE	**Système d'Exploitation** en anglais OS – Operating System
SoC	**System on Chip**

	Système monopuce – circuit intégrant sur une même puce différents composants fonctionnels (ex. mémoires, processeurs)
SPSR	**Saved Process Status Register** registre de sauvegarde d'état
SLS	**System Level Synthesis** Groupe de recherché du laboratoire TIMA/INP Grenoble
SVC	**supervisor mode** mode superviseur
softwareI	**SoftWare Interrupt** interruption logicielle

T

TBFM	**Timed Bus Functional Model** Interface pour la simulation, permettant de transformer les accès mémoire fonctionnels en des accès mémoire avec précision temporelle
TG	**Traffic Generator** générateur de trafic
thread	fil d'exécution, avec les variantes : (1) single-threaded – en utilisant un seul fil d'exécution et (2) multi-threaded – en utilisant des multiples fils d'exécution.
TIPS	**Terra Instructions par Seconde**
TLM	**Transaction Level Model** modèle au niveau transactionnel ; avec le dérivé : **TLM-CA** (en Anglais, Transaction Level Model **Cycle-Accurate**) modèle au niveau transactionnel, précis au niveau du cycle d'horloge

U

undefined instruction	instruction non définie

V

VADeL	**Virtual Architecture Description Language** langage de description de l'architecture virtuelle
VCC	**Virtual Component Codesign**
VDSL	**Very high data rate Digital Subscriber Line** Standard de communication par paires torsadées
VHDL	**Very high-scale integrated Hardware Description Language** Langage de description de système électronique numérique, basé sur une extension de Ada
virtual hardware	matériel virtuel
VLC	**Variable Length Coding** module de codage de longueur variable, en DivX
VLIW	**Very-Long Instruction Word**

W

wormhole	protocole (de routage dans les réseaux) trou-de-ver

Y

YUV	Méthode de codage colorimétrique vidéo : Luminance (Y) - Chrominance et Saturation (U-V).

Références

[Aga 00] V. Agarwal, M.S. Hrishikesh, S.W. Keckler, and D. Burger, “Clock rate versus ipc: the end of the road for conventional microarchitectures”, The 27th Annual International Symposium on Computer Architecture, 2000, pp. 248–259.

[Agr 90] V.D.Agrawal and S.T. Chakradhar, “Performance estimation in a massively parallel system”, SC, 1990, pp. 306–313.

[Amba] AMBA Specification (Rev 2.0), ARM Limited 1999, available at http://www.arm.com/products/solutions/AMBA_Spec.html

[ARM7] ARM7 Technical Reference Manual (REV 4) ARM Limited, 17 April 2001, available at http://www.arm.com

[Bac 04] I. Bacivarov, A. Bouchhima, S. Yoo, A.A. Jerraya, “ChronoSym: a new approach for fast and accurate SoC cosimulation”, Int. J. Embedded Systems (2005), Vol. 1, Nos. 1/2, pp.103–111.

[Bag 01] A. Baghdadi, D. Lyonnard, N-E. Zergainoh, A.A. Jerraya, "An Efficient Architecture Model for Systematic Design of Application-Specific Multiprocessor SoC", Proceedings DATE 2001, March 2001.

[Bal 01] F. Balarin, “STARS of MPEG decoder: a case study in worst-case analysis of discrete event systems”, Proceedings of the International Workshop on hardware/Software Codesign, 2001, pp. 104–108.

[Ben 02] L. Benini, G. de Micheli: Networks on Chip: A new SoC paradigm, IEEE Computer, January 2002.

[Bis 05] P. Biswas, S. Banerjee, N. Dutt, P. Ienne, L. Pozzi, “ISEGEN: Generation of High-Quality Instruction Set Extensions by Iterative Improvement”, DATE 2005, Munich, March 2005.

[Bis 06] P. Biswas, S. Banerjee, N. Dutt, P. Ienne, L. Pozzi, “Performance and Energy Benefits of Instruction Set Extensions in an FPGA Soft Core”, Proceedings of the 19th International Conference on VLSI Design (VLSID’06), pages 651–56, Hyderabad, India, January 2006.

[Bju 01] P. Bjuréus, A. Jantsch, “Performance Analysis with Confidence Intervals for Embedded Software Processes”, ISSS’01, October 1-3, 2001, Montréal, Québec, Canada, Copyright 2001 ACM 1-58113-418-5/01/00010.

[Bju 02] P. Bjuréus, M. Millberg, A. Jantsch, “FPGA resource and timing estimation from Matlab execution traces”, international symposium on Hardware/software codesign, Estes Park, Colorado, Pages: 31 – 36, Year of Publication: 2002, ISBN:1-58113-542-4

[Bol 03] E. Bolotin I. Cidon, R. Ginosar and A. Kolodny, “QNoC: QoS architecture and design process for network on chip”, The Journal of Systems Architecture, Dec. 2003.

[Bon 06] M. Bonaciu, A. Bouchhima, 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, to appear.

[Bona 06] M. Bonaciu, “Flexible and Scalable Algorithm/Architecture Platform for MPSoC Design of High Definition Video Compression Algorithms”, Thèse de Doctorat INPG, Spécialité Microélectronique, Laboratoire TIMA, 2006.

[Bou 06] A. Bouchhima, “Modélisation du logiciel embarqué à différents niveaux d'abstraction en vue de la validation et la synthèse des systèmes monopuces”, Thèse de Doctorat INPG, Spécialité Microélectronique, Laboratoire TIMA, 2006.

[Bou 04] J. Y. Le Boudec, “Methods, Practice and Theory for the Performance Evaluation of Computer and Communication Systems”, EPFL, 2004. 299 pp., available at http://ica1www.epfl.ch/perfeval/lectureNotes.htm

[Bra 02] M. Bradley, K. Xie, “Hardware/Software Co-Verification with RTOS Application Code”, Mentor Graphics Inc., available at http://www.techonline.com/community/tech_topic/21082, TechOnLine Publication Date: Sep. 9, 2002

[Bra 04] C. Brandolese, W. Fornaciari, and F. Salice, “An area estimation methodology for FPGA based designs at SystemC-level”, DAC 2004, pp. 129–132, June 07-11, 2004, San Diego, CA, USA

[CarK] Carbon Kernel, available at http://www.carbonkernel.org/

[Cell] The Cell Architecture, available at http://www.research.ibm.com/cell/

[Ces 01] W. Cesario, G. Nicolescu, L. Gauthier, D. Lyonnard, A. A. Jerraya, "Colif: A Design Representation for Application-Specific Multiprocessor SOCs", IEEE Design & Test of Computers, Vol. 18 n° 5, Sept/Oct 2001

[Ces 02] W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A.A. Jerraya, M. Diaz-Nava, "Component-Based Design Approach for Multicore SoCs ", DAC'02, June 10-14 2002, New Orleans, USA, 2002

[Che 01] K. Chen, S. Malik, and D.I. August, “Retargetable static timing analysis for embedded software”, ISSS, 2001, 39–44.

[ConvSC] ConvergenSC, available at http://www.coware.com

[Cop 03] M. Coppola, S. Curaba, M. Grammatikakis, G. Maruccia, F. Papariello, “On-Chip Communication Network: User Manual v1.0.1”, ST Microelectronics, AST Grenoble Lab, France, October 2003, available at http://occn.sourceforge.net/occn_user_manual.html

[Copp 03] M. Coppola, “OCCN: A Methodology for NoC”, European SystemC Users Group Meeting, Stuttgart, Germany, November 2003.

[Des 00] D. Desmet, et al., “Operating System Based Software Generation for Systems-on-Chip”, Proc. DAC, 2000.

[Dey 97] S. Dey and S. Bommu, “Performance analysis of a system of communicating processes”, International Conference on Computer-Aided Design (ICCAD 1997), ACM and IEEE Computer Society, San Jose, CA, 1997, pp. 590–597.

[Dia 01] M. Diaz-Nava, G.S. Okvist, “The Zipper prototype: A Complete and Flexible VDSL Multi-carrier Solution”, ST Microelectronics Journal special issue xDSL, Sep. 2001.

[Dum 06] F. Dumitrascu, I. Bacivarov, L. Pieralisi, M. Bonaciu, A. Jerraya, "Flexible MPSoC Platform with Fast Interconnect Exploration for Optimal System Performance for a Specific Application", DATE'06, Munich, Germany, March

[ECos] eCos, available at http://sources.redhat.com/ecos/

[Esk 02] E. Eskenazi, A. Fioukov, and D.K. Hammer, “Performance prediction for software architectures”, Proceedings of the 3D Progress Workshop on Embedded Systems, 2002.

[Gaj 03] D. Gajski et al, “Transaction based design: Another Buzzword or the solution to a Design Problem", Design, Automation & Test in Europe, Munich, Germany, Mars 2003

[Gau 01] L. Gauthier, S. Yoo, A. A. Jerraya, “Automatic Generation and Targeting of Application Specific Operating Systems and Embedded Systems Software", Proceedings DATE 2001, March 2001, Munich, Germany

[Gaut 01] L. Gauthier, “Génération de système d’exploitation pour le ciblage de logiciel multitâche sur des architectures multiprocesseur hétérogènes dans le cadre des systèmes embarqués spécifiques”, Thèse de Doctorat INPG, Spécialité Microélectronique, Laboratoire TIMA, 2001.

[Gen 05] N. Genko, D. Atienza, G. De Micheli, J. M. Mendias, R. Hermida, and F. Catthoor, “A complete network-on-chip emulation framework”, in DATE 05, pages 246-251, 2005.

[Ger 03] A. Gerstlauer, H. Yu, D. Gajski, “RTOS Modeling for System Level Design”, Proc. DATE, March 2003.

[Gra 05] A. Grasset, F. Rousseau, A.A. Jerraya, “Automatic Generation of Component Wrappers by Composition of Hardware Library Elements Starting from Communication Service Specification", RSP 2005, Montréal, Canada, 8-10 June 2005

[Gras 05] A. Grasset, Synthese des interfaces de communication dans la conceprion des systemes monopuces : de la specification à la generation automatique”, Thèse de Doctorat INPG, Spécialité Microélectronique, Laboratoire TIMA, 2005.

[Gup 00] V. Gupta, P. Sharma, M. Balakrishnan, and S.Malik, “Processor evaluation in an embedded systems design environment”, 13th International Conference on VLSI Design (VLSI-2000), Calcutta, India, 2000, pp. 98–103.

[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”, Design Automation Conference, DAC'04, San Diego, USA, June 2004.

[Han] S. Han, “Automated design of flexible memory server for highly parrallel MPSoC”, Thèse de Doctorat INPG, Spécialité Microélectronique, Laboratoire TIMA, en cours.

[Hem 00] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Oberg, M. Millberg, and D. Lindqvist, "Network on chip: An architecture for billion transistor era", Proceeding of the IEEE NorChip Conference, November 2000, pp. 166-173.

[Hen 03] J. L. Hennessy and D. A. Patterson, “Computer Architecture---A Quantitative Approach”, Morgan Kaufmann Publishers, 3rd edition, 2003.

[Hen 05] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, “System level performance analysis — the SymTA/S approach”, IEE P–Comput. Dig. Tech., 2005.

[Her 00] A. Hergenhan and W. Rosenstiel, “Static timing analysis of embedded software on advanced processor architectures”, Proceedings of Design, Automation and Test in Europe, Paris, 2000, pp. 552–559.

[Hin 97] K. Hines and G. Borriello, “Optimizing communication in embedded system cosimulation”, Codes/CASHE 1997.

[Hor 04] M. Horowitz and B. Dally, “How Scaling Will Change Processor Architecture,” Proc. Int’l Solid State Circuits Conf. (ISSCC), pp. 132-133, Feb. 2004

[Itrs] International Technology Roadmap for Semiconductors http://public.itrs.net

[Itrs 01] ITRS, International Technology Roadmap for Semiconductors, Design, Edition 2001, available at http://public.itrs.net/Files/2001ITRS/Design.pdf

[Itrs 05] ITRS, International Technology Roadmap for Semiconductors, Design, Edition 2005, available at http://www.itrs.net/Common/2005ITRS/Design2005.pdf

[Iwa 03] H. Iwasaki, J. Naganuma, K. Nitta, K. Nakamura, T. Yoshitome, M. Ogura, Y. Nakajima, Y. Tashiro, T. Onishi, M. Ikeda, M. Endo. "Single-Chip MPEG-2 422P@HL CODEC LSI with Multi-Chip Configuration for Large Scale Processing beyond HDTV Level," p. 20002, Design, Automation and Test in Europe Conference and Exhibition (DATE'03 Designers' Forum), 2003.

[Jer 04] A. A. Jerraya, W. Wolf, “Multiprocessor Systems-on-Chips”, Morgan Kaufmann Publishers, ISBN 0-12-385251-X, September 2004.

[Jer 06] A. A. Jerraya, I. Bacivarov, “Performance Evaluation Methods for MPSoC Design”, Chapter 6 in “Electronic Design Automation for Integrated Circuits Handbook”, CRC Press, ISBN: 0849330963, publication date: 4/13/2006.

[Jon 03] H. Jones, “Analysis of the relationship between EDA Expenditures and Competitive Positioning of IC Vendors for 2003”, http://www.edac.org/resources_profitability.jsp

[Kim 04] S. Kim, S. Ha, "Exploring On-Chip Bus Architectures for Multitask Applications", Journal of Semiconductor Technology and Science (JSTS), Vol. 4 pp 286-292, December 2004

[Kin 00] P. King and R. Pooley, “Derivation of petri net performance models from UML specifications of communications software”, Proceedings of Computer Performance Evaluation Modelling Techniques and Tools: 11th International Conference, TOOLS 2000, Schaumburg, IL, 2000.

[Kle 75] L. Kleinrock. Queueing Systems – Volume 1: Theory, volume 1, John Wiley and Sons, New York, 1975.

[Kle 76] Leonard Kleinrock. Queueing Systems – Volume 2: Computer Applications, volume 2, John Wiley and Sons, New York, 1976.

[Lah 99] K. Lahiri, A. Raghunathan, and S. Dey, "Fast performance analysis of bus-based system-on-chip communication architectures", in Proc. Int. Conf. Computer-Aided Design, pp. 566--572, Nov. 1999.

[Lah 01] K. Lahiri, S. Dey, and A. Raghunathan, “Evaluation of the traffic-performance characteristics of system-on-chip communication architectures”, in VLSID '01, page 29, IEEE Computer Society, 2001.

[Laj 98] M. Lajolo, A. Raghunathan, S. Dey, L. Lavagno, and A. Sangiovanni-Vincentelli, “A case study on modeling shared memory access effects during performance analysis of hardware/software systems”, Proceedings of the 6th IEEE International Workshop on Hardware/software Codesign, Seattle,WA, 15–18, 1998, pp. 117–121.

[Laj 99] M. Lajolo, M. Lazarescu, and A. Sangiovanni-Vincentelli, “A compilation-based software estimation scheme for hardware/software cosimulation”, Proceedings of the 7th IEEE International Workshop on Hardware/software Codesign, Rome, Italy, 3–5, 1999, pp. 85–89.

[Laj 00] M. Lajolo, A. Raghunathan, S. Dey, and L. Lavagno, “Efficient power co-estimation techniques for systems-on-chip design”, Proceedings of Design Automation and Test in Europe, Paris, 2000.

[LexYacc] The Lex and Yacc Web Page, available at http://dinosaur.compilertools.net/

[Li 97] Y. Li and W.Wolf, A task-level hierarchical memory model for system synthesis of multiprocessors, Proceedings of the Design Automation Conference, 1997.

[Liu 02] Y. Liu, I. Gorton, A. Liu, N. Jiang, and S.P. Chen, “Designing a test suite for empirically-based middleware performance prediction”, The Proceedings of TOOLS Pacific 2002, Sydney, Australia, 2002.

[Log 04] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon, “Analyzing on-chip communication in a MPSoC environment”, Proceedings of the Design, Automation and Test in Europe Conference and Exhibition Volume II (DATE2004), Vol. 2, 2004

[Logh 04] M. Loghi, M. Poncino, and L. Benini, “Cycle-accurate power analysis for multiprocessor systems-on-a-chip”, ACM Great Lakes Symposium on VLSI, 2004, 410–406

[Lu 00] C Lu, J.A. Stankovic, T.F. Abdelzaher, G. Tao, S.H. Son, and M. Marley, “Performance specifications and metrics for adaptive real-time systems”, IEEE Real-Time Systems Symposium (RTSS 2000), Orlando, FL, 2000.

[Lyo 03] D. Lyonnard, “Approche d’assemblage systematique d’elements d’interface pour la generation d’architecture multiprocesseur”, Thèse de Doctorat INPG, Spécialité Microélectronique, Laboratoire TIMA, 2003.

[Mah 05] S. Mahadevan, F. Angiolini, M. Storgaard, R. G. Olsen, J. Sparso, and J. Madsen, “A network traffic generator model for fast network-on-chip simulation”, in DATE 05, pages 780-785, 2005.

[Mar 01] R. Marculescu, A. Nandi, L. Lavagno, and A. Sangiovanni-Vincentelli, “System-level power/performance analysis of portable multimedia systems communicating over wireless channels”, Proceedings of IEEE/ACM International Conference on Computer Aided Design, San Jose, CA, 2001.

[Max 04] A. Maxiaguine, S. Künzli, S. Chakraborty, and L. Thiele, “Rate analysis for streaming applications with on-chip buffer constraints”, ASP-DAC, Yokohama, Japan, 2004.

[McK 99] P.E. McKenney, “Practical performance estimation on shared-memory multiprocessors”, Parall. Distr. Comput. Syst., 1999.

[Mic 01] G. de Micheli, R. Ernst, W. Wolf, “Readings in Hardware/Software Co-Design”, Morgan Kaufmann; 1st edition (June 1, 2001), ISBN 1558607021

[ModSim] ModelSim, available at http://www.model.com/

[Mpi] The MPI Standard, available at http://www-unix.mcs.anl.gov/mpi/standard.html

[MPSoC 05] Proceedings of 5th International Forum on Application-Specific Multi-Processor SoC, 11 - 15 July 2005, available at http://tima.imag.fr/mpsoc/

[Mut 04] A. Muttreja, A. Raghunathan, S. Ravi, and N.K. Jha, “Automated energy/performance macromodeling of embedded software”, ACM/IEEE Design Automation Conference (DAC), 2004.

[Net 04] M. K. Nethi, J. H. Aylor, “Mixed level modelling and simulation of large scale hardware/software systems”, High performance scientific and engineering computing: hardware/software support 2004: 157 – 166, ISBN:1-4020-7580-4

[Nex] Philips Semiconductors, Nexperia cellular system solutions, available at http://www.semiconductors.philips.com/markets/communications/nexperia/system/

[Nic 01] G. Nicolescu, S. Yoo, A. A. Jerraya, "Mixed-Level Cosimulation for Fine Gradual Refinement of Communication in SoC Design", Proceedings DATE 2001, March 2001, Munich, Germany.

[Nic 02] G. Nicolescu, K. Svarstad, W. Cesario, L. Gauthier, D. Lyonnard, S. Yoo, P. Coste, A.A. Jerraya, “Desiderata pour la spécification et la conception des systèmes électroniques”, Technique et Science Informatiques, Mars 2002.

[Nic 03] G. Nicolescu, “Spécification et validation des systèmes hétérogènes embarqués”, Thèse de Doctorat INPG, Spécialité Microélectronique, Laboratoire TIMA, 2003.

[Noma] ST Microelectronics, Nomadik® Multimedia Processor, available at http://www.st.com/stonline/prodpres/dedicate/proc/proc.htm

[Ofe 00] D. Ofelt and J.L. Hennessy, “Efficient performance prediction for modern microprocessors”, SIGMETRICS, 2000, pp. 229–239.

[Pat 98] D. A. Patterson A. and J. L. Hennessy, “Computer organization and design, the hardware/software interface”, second edition, Morgan-Kaufmann, San Francisco, California, 1998, ISBN 155860 -491-X.

[Pes 04] S. G. Pestana, E. Rijpkema, A. Radulescu, K.G.W. Goossens, and O.P. Gangwal, “Cost-performance trade-offs in networks on chip: a simulation-based approach”, DATE, 2004, pp. 764–769.

[Pet 06] I. Petkov, Conception des sysetmes monopuces multiprocesseur : de la simulation, vers la realisation”, Thèse de Doctorat UJF, Spécialité Microélectronique, Laboratoire TIMA, 2006.

[Pier] L. Pieralisi, “SoC challenges ahead: Networks on Silicon”, Thèse de Doctorat INPG, Spécialité Microélectronique, Laboratoire TIMA – ST Microelectronics, en cours.

[Pol 03] F. Poletti, D. Bertozzi, L. Benini, and A. Bogliolo, “Performance analysis of arbitration policies for SoC communication architectures”, Kluwer J. Des. Autom. Embed. Syst., 8, 189–210, 2003. DE MODIF

[Rij 03] E. Rijpkema, K. G. W. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander. “Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip”. In Proceedings of Design Automation and Test Conference in Europe, Munich, Germany, March 2003.

[Row 94] J. A. Rowson, “Hardware/Software Co-Simulation”, Proc. DAC, 1994.

[Row 97] J.A. Rowson and A.L. Sangiovanni-Vincentelli, “Interface-based design”, Proceedings of the 34th Conference on Design Automation, Anaheim Convention Center, ACM Press, Anaheim, CA, ISBN 0-89791-920-3, 9–13, 1997, pp.J 178–183.

[Sar 05] A. Sarmento, L. Kriaa, A. Grasset, W. Youssef, A. Bouchhima, F. Rousseau, W. Cesario, A.A. Jerraya, "Service Dependency Graph, an Efficient Model for Hardware/Software Interfaces Modeling and Generation for SoC Design", International Conference on Hardware - Software Codesign and System Synthesis CODES-ISSS 2005, New York Metropolitan Area, USA, Sept. 18-21, 2005.

[Sarm 05] A. Sarmento, “Automatic Generation of Simulation Models for the validation of heterogeneous systems-on-chip”, Thèse de Doctorat UJF, Spécialité Microélectronique, Laboratoire TIMA, 2005.

[Sch 04] T. Schuele and K. Schneider, “Abstraction of assembler programs for symbolic worst case execution time analysis”, DAC, 2004, pp. 107–112

[Sch 05] A. Scherrer, A. Fraboulet, T. Risset, “Analysis and Synthesis of Cycle-Accurate On-Chip Traffic with Long-Range-Dependence”, Laboratoire de l’Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL no 5668, Research Report No 2005-53, November 2005, available at http://www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2005/RR2005-53.pdf

[SeamCVE] Seamless CVE, available at http://www.mentor.com

[Sel 93] B. Selic, "An Efficient Object-Oriented Variation of the Statecharts Formalism for Distributed Real-Time Systems," CHDL ’93: IFIP Conference on Hardware Description Languages and Their Applications, Ottawa, Canada, April 28-28, 1993.

[Sel 98] B. Selic, J. Rumbaugh, “Using UML for Modeling Complex Real-Time Systems”. March 1998. Rational software Whitepaper.

[Sem 00] L. Semeria and A. Ghosh, “Methodology for Hardware/Software Co-verification in C/C++”, Proc. Asia South Pacific Design Automation Conference (ASPDAC), Jan. 2000.

[Sparc] The SPARC Architecture Manual, Version 8, Revision SAV080SI9308, available at www.sparc.org/standards/V8.pdf

[Spi 98] B. Spitznagel and D. Garlan, “Architecture-based performance analysis”, Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering (SEKE 1998), 1998.

[Suz 96] K. Suzuki and A.L. Sangiovanni-Vincentelli, “Efficient software performance estimation methods for hardware/software codesign”, DAC, 1996, pp. 605–610.

[SStu] System Studio, available at http://www.synopsys.com

[SystemC] SystemC, available at http://www.systemc.org

[Tan 95] S. M. Tan, et. al., “Virtual Hardware for Operating System Development”, Technical rep., UIUC, Sep. 1995, available at http://choices.cs.uiuc.edu/uChoices/Papers/uChoices/vchoices/vchoices.pdf

[Thi 03] R. Thid, M. Millberg, and A. Jantsch, Evaluating NoC communication backbones with simulation”, in 21th IEEE Norchip Conference, Riga, November 2003.

[Tiw 94] V. Tiwari, S. Malik, and A.Wolfe, “Power analysis of embedded software: a first step towards software power minimization”, IEEE T. VLSI Syst., 2, 437–445, 1994.

[Unr 00] R. C. Unrau, “Development Techniques for Using Simulation to Remove Risk in Software/Hardware Integration January 2000, Red Hat, available at http://www.redhat.com/support/wpapers/cygnus/cygnus_risk/index.html#toc

[VCC] Virtual Component Codesign, Cadence Design Systems Inc. , available at http://www.cadence.com/products/vcc.html

[VSIA] Virtual Socket Interface Alliance, http://www.vsi.org

[VxSim] VxSim, Windriver Systems Inc., available at http://www.windriver.com/products/html/vxsim.html

[Wal 98] J. Walrath and R. Vemuri, "A performance modeling and analysis environment for reconfigurable computers", IPPS/SPDP Workshops, 1998, pp. 19–24.

[Wey 02] E.J. Weyuker and A. Avritzer, "A metric for predicting the performance of an application under a growing workload", IBM Syst. J., 41, 45–54, 2002.

[Wik 04] D. Wiklund, S. Sathe, and D. Liu, "Network on chip simulations for benchmarking", in IWSOC, pages 269-274, 2004.

[Wor 04] F. Worm, P. Ienne, P. Thiran, G. De Micheli, "On-Chip Self-Calibrating Communication Techniques: Robust to Electrical Parameter Variations", IEEE Design & Test of Computers, Vol. 21, No. 6, November-December 2004

[Wor 05] F. Worm, P. Ienne, P. Thiran, G. De Micheli "A Robust Self-Calibrating Transmission Scheme for On-Chip Networks", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 13, No. 1, January 2005

[Yal 05] S. Yaldiz, A. Demir, S. Tasiran, P. Ienne, Y. Leblebici, "Characterizing and Exploiting Task-Load Variability and Correlation for Energy Management in Multi-Core Systems", In Proceedings of the 3rd IEEE Workshop on Embedded Systems for Real Time Multimedia, pages 129–40, Jersey City, N.J., September 2005.

[Yos 04] H. Yoshida, K. De, and V. Boppana, "Accurate pre-layout estimation of standard cell characteristics", DAC, 2004, pp. 208–211.

[Yoo 01] S. Yoo, G. Nicolescu, D. Lyonnard, A. Baghdadi, A. A. Jerraya, "A Generic Wrapper Architecture for Multi-Processor SoC Cosimulation and Design", Proceedings CODES 2001, Copenhague, Danemark, April 2001

[Ziv 96] V. Zivojnovic, H. Meyr, "Compiled hardware/software Co-Simulation", Proc. DAC, 1996.

Publications

1 Publications Internationales – Chapitres de Livres

- A. Jerraya, I. Bacivarov, “Performance Evaluation Methods for MPSoC Design”, Chapter 6 in “Electronic Design Automation for Integrated Circuits Handbook”, CRC Press, 2006, ISBN 0-8493-7923-7

- S. Yoo, G. Nicolescu, I. Bacivarov, W. Youssef, A. Bouchhima, A.A. Jerraya, "Multi-Level Software Validation for NoC", Chapter 10 in "Networks on Chip", pp. 261-281 Kluwer Academic Publishers, 2003, ISBN 1-4020- 7392-5

2 Publications Internationales - Journaux

- I. Bacivarov, A. Bouchhima, S. Yoo, A.A. Jerraya, "ChronoSym – a New Approach for Fast and Accurate SoC Cosimulation", International Journal of Embedded Systems(IJES), Inderscience Publishers, Volume: 1 - Issue: 1/2, pp. 103-111, 2006, ISSN (Online): 1741-1076, ISSN (Print): 1741-1068

3 Publications Nationales - Journaux

- I. Bacivarov, C. Roman, D. Babus, “The application of Genetic Algorithms in optimal planning with constraints”, in Calitatea Journal, vol. 3, nr. 11-12, nov.-dec 2002, pp.65-72.

4 Publications Internationales - Conférences

- F. Dumitrascu, I. Bacivarov, L. Pieralisi, M. Bonaciu, A. Jerraya, "Flexible MPSoC Platform with Fast Interconnect Exploration for Optimal System Performance for a Specific Application", DATE'06, Munich, Germany, March 2006.

- A. Bouchhima, I. Bacivarov, W. Youssef, M. Bonaciu, A.A. Jerraya, "Using Abstract CPU Subsystem Simulation Model for High Level HW/SW Architecture Exploration", ASP-DAC 2005 proceedings, 18-21 January 2005, Shanghai, China, 2005.

- 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 ", DATE'03, Munich, Germany, March 2003.

- I. Bacivarov, S. Yoo, A.A. Jerraya, "Timed HW-SW Cosimulation Using Native Execution of OS and Application SW ", Seventh IEEE International High-Level Design Validation and Test workshop HLDVT'02, Cannes, France, October 2002.

- Iuliana Bacivarov, A. A. Jerraya, S. Yoo, “A fast and Accurate Validation Technique for Operating Systems in MPSOC Design”, in Proceedings of SPIE Advanced Topics in Optoelectronics, Microelectronics and Nanotechnologies, vol. 5227 ,2003, pp342-346

- Iuliana Bacivarov, A. A. Jerraya, S. Yoo, "A Fast and Accurate Validation Technique for Operating System in Multi - Processor System - on - Chip Design" in Proceedings of SPIE Advanced Topics in Optoelectronics, Microelectronics and Nanotechnologies, vol. 5227, 2003, pp 342-346

- I. Bacivarov, A. A. Jerraya, S. Yoo, “Quality Enhancement in SoC Design through Timed HW-SW Cosimulation” in Proceedings of the 8th International in Conference in Quality and Dependability-CCF2002, Sinaia, Romania, September 27-29, 2002, pp 137-141

5 Rapports Techniques

- Iuliana Bacivarov, "Estimation du temps d'exécution du logiciel embarqué en tenant compte des interfaces logicielles/matérielles", Master of Science Thesis, TIMA Laboratory, INPGrenoble, July 2003

- G. Majauskas, I. Bacivarov, G. Nicolescu, W. Cesario, L. Gauthier, D. Lyonnard, X. Chen, "Multiprocessor System-on-Chip Design of a VDSL Application using ROSES – User’s Tutorial", Internal Report Interne, TIMA Laboratory, INPGrenoble, July 2002

- Iuliana Bacivarov, "Fast Validation Techniques for an Application Specific, SoC Compliant, Operating System", Final Engineer Diploma Report, TIMA Laboratory, INPGrenoble, July 2002

- C. Koch, Iuliana Bacivarov, "Application du Flot du Conception pour la Réalisation d'un Décodeur WCDMA", Internal Report, TIMA Laboratory, INPGrenoble, September 2001

- Iuliana Bacivarov, Gabriela Nicolescu, Wander Cesario, Lovic Gauthier, Damien Lyonnard, "Hands on Tutorial Hardware-Software Architecture Design of Multi-Processor System-on-Chip", Internal Report, TIMA Laboratory, INPGrenoble, August 2001

RESUME

Les systèmes embarqués multiprocesseur monopuces (Multi-Processor System-on-Chip, MPSoC) visent l'intégration des sous-systèmes variés, matériels et logiciels, sur une seule puce. Ainsi, l'hétérogénéité et les contraintes imposées pour la mise sur le marché rendent l'analyse en vue de l'évaluation des performances et de l'optimisation de ces systèmes très complexes.

L'évaluation des performances est une étape clef dans n'importe quel flot de conception. En se basant sur les résultats de l'évaluation des performances, il est possible de prendre des décisions et de réaliser des compromis pour l'optimisation du système global.

La littérature prouve qu'une grande partie du temps de conception est passée dans l'évaluation des performances. De plus, les itérations dans le flot de conception deviennent prohibitives pour des systèmes complexes. Par conséquent, la réalisation des MPSoCs à rendement élevé est un défi. La solution est fortement liée à la disponibilité des méthodes rapides et précises pour l'évaluation des performances.

Dans cette thèse, le terme « performances » est limité aux performances des temps d'exécution pour la réalisation finale du système. L'aspect temporel est intensivement analysé pour la validation des systèmes temps-réel et l'optimisation des sous-ensembles d'interconnexion. Nous avons également considéré la vitesse de la méthode proposée d'évaluation des performances, car les temps d'évaluation peuvent devenir prohibitifs pour des systèmes MPSoC complexes.

Notre principale contribution est de définir une méthode globale d'évaluation des performances pour les systèmes MPSoC. La deuxième contribution est la construction d'un modèle rapide et précis pour l'évaluation des performances du logiciel embarqué. On a réalisé un modèle de haut niveau d'abstraction, afin d'avoir une vitesse élevée d'évaluation. De plus, on a inclus des annotations des temps d'exécution, afin d'avoir une bonne précision d'évaluation. La troisième contribution est l'exploration rapide et précise de l'espace architectural des sous-systèmes d'interconnexion, afin de pouvoir faire des choix efficaces dans la conception.

TITRE EN ANGLAIS

PERFORMANCE EVALUATION FOR HETEROGENEOUS MPSOC DESIGN

ABSTRACT

Multi-processor system-on-chip (MPSoC) is a concept that aims at integrating multiple subsystems on a single chip. Systems that put together complex HW and SW subsystems are difficult to analyze and even harder to optimize.

Performance evaluation is a key step in any design, allowing for decisions and trade-offs, in view of overall system optimization. The literature shows that a large part of the design time spent in performance evaluation, and iterations become prohibitive in complex designs. Therefore, the challenge of building high-performance MPSoCs is closely related to the availability of fast and accurate performance evaluation methods.

In our work, "performance" is restricted to time related performances of the final architecture. The timing aspect is intensively analyzed for the validation of real-time systems and the optimization of interconnect subsystems. We are also concerned with the speed of any proposed performance evaluation method, as evaluation times may become prohibitive for complex MPSoC designs.

Our main contribution is to define a global performance evaluation method for MPSoC. We also orient our research towards software performance modeling. Our second contribution is to define a high level software model, in order to have a high evaluation speed, and including timing annotations, in order to have good evaluation accuracy. The third contribution is to define a fast and accurate method for interconnect sub-system design space exploration, in order to make efficient design choices.

SPECIALITE MICROELECTRONIQUE

MOTS CLES

systèmes MPSoC, évaluation de performance, temps d'exécution, système d'exploitation embarqué, sous-système d'interconnexion embarqué, cosimulation logicielle/matérielle, génération automatique, interfaces d'adaptation

INTITULE ET ADRESSE DU LABORATOIRE

Laboratoire TIMA, Techniques d'Informatique et de la Microélectronique pour l'Architecture des ordinateurs, 46, av. Félix Viallet, 38000 GRENOBLE

ISBN/ISBNE: 2-84813-090-3