N° d'ordre : 4107




Université des Sciences et Technologies de Lille

THÈSE

présentée et soutenue publiquement le 05 mars 2008

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Rabie BEN ATITALLAH

Modèles et simulation des systèmes sur puce multiprocesseurs - Estimation des performances et de la consommation d'énergie

Composition du jury

Président :	Gilles GONCALVES	Professeur	LGI2A Université dArtois
Rapporteurs :	Ahmed Amine JERRAYA	Directeur de Recherche	CEA-LETI
	Olivier SENTIEYS	Professeur	ENSSAT, Université de Rennes I
Examinateurs :	Sami YEHIA	Docteur	Thales Research and Technology
Directeurs :	Jean-Luc DEKEYSER	Professeur	LIFL, Université de Lille I
	Smail NIAR	MdC, HdR	LAMIH, Université de Valenciennes

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
Laboratoire d'Informatique Fondamentale de Lille — UPRESA 8022
U.F.R. d'I.E.E.A. – Bât. M3 – 59655 VILLENEUVE D'ASCQ CEDEX
Tél. : +33 (0)3 28 77 85 41 – Télécopie : +33 (0)3 28 77 85 37 – email : direction@lifl.fr

Table des matières

TABLE DES MATIÈRES v

Chapitre 1

Introduction

1.1 Contexte

Les systèmes embarqués sur puce (*SoC : System on Chip*) envahissent chaque jour un peu plus notre vie quotidienne et professionnelle. Il est difficile de trouver de nos jours un domaine où ces systèmes ne sont pas présents. Depuis plusieurs années, les systèmes embarqués sont utilisés dans divers domaines comme les télécommunications, l'avionique, l'automobile, la photo numérique, les appareils domestiques (fours, lave-vaisselles, etc.), les implants médicaux, etc. Alors, qu'environ 260 millions de processeurs sont vendus en 2004 pour équiper nos PC de bureau, plus de 14 milliards de processeurs (sous diverses formes : microprocesseur, microcontrôleur, DSP) sont vendus pour les systèmes embarqués. Par ailleurs, une augmentation annuelle de 16% du chiffre d'affaires est prévue pour le marché des systèmes embarqués d'ici fin 2009 (Future of Embedded Systems Technology from BCC Research Group).

Pour satisfaire les besoins des utilisateurs, les SoC d'aujourd'hui offrent des fonctionnalités de plus en plus complexes, telles que la visualisation vidéo, la connexion internet, le commerce électronique, etc. Malheureusement, ces nouvelles applications risquent de violer les contraintes imposées par ces systèmes embarqués. En effet, les SoC doivent être conçus de manière efficace afin d'assurer une certaine fiabilité et de minimiser le temps d'arrivée sur le marché (*time to market*). D'un point de vue commercial, ces deux facteurs favorisent la réussite du produit final. En outre, ces systèmes doivent être également développés et vérifiés de manière sûre pour minimiser le coût de fabrication, la consommation d'énergie et la taille du système.

Pour répondre à ces besoins et profiter de l'augmentation du nombre de transistors par puce, différents partenaires industriels et académiques multiplient leurs efforts pour faciliter le développement des systèmes sur puce. Le projet INRIA DaRT[1] dans lequel est réalisée cette thèse s'inscrit dans ce contexte. Un des principaux objectifs de ce projet est la mise en œuvre d'une méthodologie et d'un environnement de développement pour les systèmes sur puce hautes performances. Ces systèmes sont particulièrement adaptés aux applications de traitement de signal intensif telles que les applications de traitement d'images et de son. Ces traitements sont parallèles et répétitifs.

Dans ce cadre et afin de profiter de ce parallélisme, il est logique de penser à l'utilisation de systèmes multiprocesseurs (Multi-Processor SoC ou MPSoC). Ces derniers sont devenus une solution incontournable pour l'exécution d'un tel type d'application. Ce sont généralement des systèmes hétérogènes puisqu'ils peuvent contenir, entre autres, de la mémoire (Cache/SRAM, Flash), différents processeurs (processeur RISC, VLIW), des réseaux d'interconnexion ou NoC (pour network-on-Chip), des périphériques d'entrée et de sortie et éventuellement de la logique reconfigurable (FPGA). L'objectif du projet DART est de fournir un cadre unifié pour le développement de ces systèmes en partant de leur modélisation de haut niveau jusqu'à la génération du code permettant la conception matérielle et logicielle du SoC.

Notre contribution au projet DART est de proposer un environnement permettant l'évaluation des performances de ces systèmes. Cet environnement sera utile dans la phase de l'exploration de l'espace des solutions afin de réduire le nombre d'alternatives considérées et ainsi converger rapidement vers le couple Architecture/Application le plus adéquat. En effet, l'un des défis majeurs dans la conception des MPSoC d'aujourd'hui, est la réduction

[1] `http://www.inria.fr/rapportsactivite/RA2006/dart/dart_tf.html`

de la phase d'évaluation des différentes alternatives de conception. Ces alternatives sont représentées par un espace de solutions MPSoC très large.

C'est dans ce contexte que se situe notre thèse. Le travail réalisé se place plus exactement à l'intersection de trois axes de recherche (figure 1.1). Le premier concerne la co-simulation logicielle/matérielle des MPSoC à différents niveaux d'abstraction où nous nous intéressons en particulier aux niveaux transactionnels ou TLM (*Transaction Level Modeling*). Le deuxième axe concerne l'estimation des performances et de la consommation d'énergie. Cette estimation doit être fiable afin de localiser les meilleures solutions architecturales. Le troisième axe concerne la méthodologie de conception basée sur l'approche Ingénierie Dirigée par les Modèles (IDM). L'IDM offre un cadre conceptuel permettant la génération automatique du code du système à partir d'une modélisation de haut niveau.

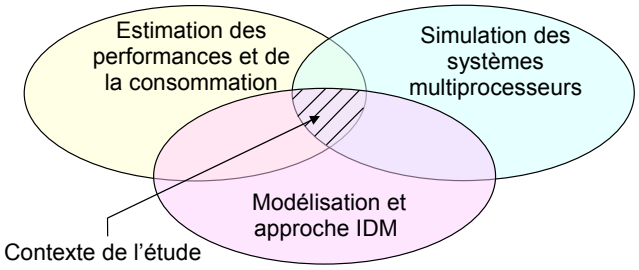


FIG. 1.1 – *Contexte de l'étude*

1.2 Problématique

L'augmentation de la complexité de conception des MPSoC nécessite des outils de développement puissants. Nous avons en effet besoin d'outils pour concevoir des systèmes sur puce de façon plus rapide et plus efficace en termes d'exploitation des ressources matérielles disponibles. Or, cet objectif ne peut être atteint qu'avec une étude des difficultés de conception afin de proposer des solutions pertinentes. Dans ce cadre, nous avons identifié trois principaux facteurs qui définissent la problématique de notre thèse :

La co-simulation des MPSoC

Aujourd'hui, les processus de conception des MPSoC tentent d'intégrer différents composants hétérogènes. Cette hétérogénéité est identifiée en termes de langages de description de composants, de niveaux d'abstraction utilisés pour modéliser les composants, de protocoles de description utilisés pour permettre la communication entre les composants, etc.

L'hétérogénéité des MPSoC crée par conséquent le besoin d'utiliser des outils divers et variés afin de réaliser la synchronisation entre les différents composants. Ceci complexifie encore plus la tâche de conception et rend la validation globale d'un MPSoC entier un objectif difficile à atteindre. Face à cet obstacle, les développeurs tentent la spécification du système entier avec un langage unique ou l'utilisation d'un seul simulateur pour réduire la complexité. Néanmoins, il en découle un manque de flexibilité dans la réutilisation des composants disponibles car il devient impossible d'intégrer dans la conception des modules qui sont développés dans un langage autre que celui qui a été choisi. Comme on peut le voir, les

choix du langage de description, du simulateur et du niveau d'abstraction doivent être faits en adéquation avec la complexité des MPSoC cibles.

L'estimation des performances et de la consommation d'énergie

La mise en œuvre de la co-simulation des systèmes MPSoC nécessite des outils d'estimation de performance et de consommation d'énergie. En effet, ces outils sont utiles pour une comparaison fiable des solutions architecturales. Au fur et à mesure que la modélisation se rapproche des niveaux les plus bas, l'espace de solutions architecturales se réduit en se basant sur des critères définis. Le défi dans chaque niveau d'abstraction est de développer des outils d'estimation de performance et de la consommation tout en maintenant un niveau minimum de précision afin de garantir la fiabilité de l'exploration.

La méthodologie de conception

Dans l'industrie, les soucis d'efficacité, de réutilisation, et de programmation sûre sont bien connus. Face à ces difficultés, et afin de comprendre et de décrire le fonctionnement d'un système, il est nécessaire d'utiliser des techniques de modélisation et des langages communs permettant l'expression, l'échange et la compréhension des principales fonctionnalités des systèmes étudiés. Dans ce domaine, l'*ingénierie dirigée par les modèles* (IDM) est considérée aujourd'hui comme l'une des approches les plus prometteuses dans le développement des systèmes. Cette technique offre un cadre méthodologique et technologique qui permet d'unifier différentes façons de faire dans un processus homogène. Son objectif est de favoriser l'étude séparée des différents aspects du système. Cette séparation permet d'augmenter la réutilisation et aide les membres de l'équipe de conception à partager et s'échanger leurs travaux indépendamment de leur domaine d'expertise. L'IDM présente une forme d'ingénierie générative permettant d'aboutir au code du système décrit au haut niveau. Dans le cadre de ce travail, nous allons tirer profit de l'IDM pour résoudre la complexité de conception des MPSoC.

1.3 Contributions

Notre contribution, à travers cette thèse, au domaine de la conception de MPSoC est de proposer des solutions permettant de lever les verrous technologiques présentés dans la section précédente. En résumé, nos contributions sont :

La co-simulation des MPSoC

1) Un environnement de co-simulation logicielle/matérielle pour les MPSoC a été proposé. Il est décrit au niveau transactionnel TLM et composé de trois sous-niveaux. Ces derniers offrent différents compromis entre l'accélération de la simulation et la précision.

2) Différents composants matériels (*IP : Intellectual Properties*) tels que le processeur, la mémoire cache, le réseau d'interconnexion, accélérateur matériel, etc. ont été conçus afin d'être implémentés dans différents niveaux d'abstraction. Afin de favoriser la réutilisation de ces IP dans le flot de conception, une bibliothèque de composants, GaspardLib, a été mise en place.

L'estimation des performances et de la consommation d'énergie

3) L'environnement de co-simulation des MPSoC a été enrichi par des modèles d'estimation de performance afin de rendre possible l'évaluation du temps d'exécution de l'application. Ce paramètre constitue un premier critère de sélection entre les différentes solutions architecturales possibles.

4) L'environnement de co-simulation intègre aussi des outils flexibles pour l'évaluation de la consommation d'énergie. Cette métrique constitue un deuxième critère de sélection des solutions architecturales dans une exploration de l'espace de solutions. Cette proposition a été concrétisée dans notre travail par le développement des modèles de consommation d'énergie à différents niveaux d'abstraction.

La méthodologie de conception

5) Un méta-modèle de déploiement a été défini afin d'autoriser la génération complète du code à partir des modèles de haut niveau. Ce méta-modèle permet également d'exprimer les spécifications de temps et de consommation d'énergie de chaque composant.

6) Une transformation dans l'objectif de génération de code a été développée afin de bénéficier d'une chaîne complète de compilation capable d'exploiter directement les modèles.

7) Les différentes contributions précédentes ont été mises en œuvre dans l'environnement Gaspard[2] développé par l'équipe, permettant en particulier de compléter son flot de conception.

1.4 Plan

Notre manuscrit est organisé selon le plan suivant :

Chapitre 2 : État de l'art Dans ce chapitre nous présentons le contexte de nos travaux qui abordent la co-simulation des systèmes sur puce, les techniques d'estimation de performance et de la consommation d'énergie et enfin les méthodologies de conception basées sur une approche modèles. A partir de cette étude, nous positionnons nos travaux et nous donnons les grandes lignes de nos contributions.

Chapitre 3 : Simulation et estimation de performance des MPSoC au niveau CABA Ce chapitre a pour but d'étudier la problématique de l'accroissement du temps de simulation des MPSoC au niveau CABA (Cycle Accurate Bit Accurate). Cette étude est réalisée en utilisant la bibliothèque de composants SoCLib. Nous analysons l'effort de développement nécessaire à ce niveau d'abstraction qui permet d'obtenir des estimations précises.

Chapitre 4 : Simulation et estimation de performance des MPSoC au niveau PVT Dans ce chapitre, une proposition de description des systèmes MPSoC au niveau transactionnel PVT est présentée. A ce niveau, nous proposons un environnement sous forme de trois sous-niveaux de simulation visant à accélérer la simulation des systèmes MPSoC. Pour évaluer l'intérêt de notre proposition dans la description des MPSoC, différents composants matériels ont été conçus dans les trois sous-niveaux. Enfin, à chacun de ces derniers, un modèle de temps a été développé permettant l'estimation des performances.

Chapitre 5 : Estimation de la consommation d'énergie dans les MPSoC Dans ce chapitre, l'objectif est d'enrichir les simulateurs MPSoC décrits aux niveaux CABA et PVT (pré-

[2] http://www2.lifl.fr/west/gaspard/index.html#g2

sentés dans les chapitres 3 et 4) par des modèles de consommation d'énergie. Nous proposons une méthodologie hybride d'estimation de la consommation basée sur des simulations de bas niveau d'une part, et des modèles analytiques d'autre part. Cette méthodologie offre un niveau de précision et de flexibilité intéressant. L'intégration du critère de la consommation dans notre travail a été réalisée de façon à obtenir un environnement flexible pouvant être utilisé dans une exploration rapide et précise des systèmes MPSoC.

Chapitre 6 : Ingénierie dirigée par les modèles pour la simulation des MPSoC Ce chapitre présente notre chaîne de compilation à base de transformations de modèles afin de générer automatiquement la description du système en SystemC. Nous proposons une évolution du méta-modèle Gaspard via l'introduction du paquetage *Deployment*. Ce dernier permet de faire le lien entre les composants élémentaires définis à haut niveau et les implémentations réelles correspondantes. De plus, des concepts liés à l'estimation de performance et de la consommation d'énergie sont introduits dans ce paquetage. Par la suite, deux transformations sont appliquées dans le but de se rapprocher de plus en plus du code de simulation. A partir de là, nous développons une transformation de type modèle-vers-texte. Cette dernière est capable de générer l'ensemble de fichiers compilables pour produire le simulateur exécutable.

Chapitre 7 : Étude de cas et validation expérimentale Une étude de cas d'une application de codage vidéo placée sur un MPSoC à base de plusieurs processeurs MIPS est présentée. Cette étude de cas permet non seulement de donner un aperçu sur le flot de conception des MPSoC dans l'environnement Gaspard mais valide aussi les différentes propositions faites précédemment.

Chapitre 8 : Conclusion Nous concluons cette thèse par le bilan des travaux effectués et nous détaillons les contributions apportées avant d'aborder quelques perspectives à nos travaux.

Chapitre 2

État de l'art

2.1 Introduction

La motivation principale de notre travail est la mise au point d'outils permettant d'un côté de réduire le temps de simulation afin d'obtenir les performances de différentes alternatives et de l'autre côté de simplifier la phase de conception des MPSoC. En effet, l'une des difficultés rencontrées par les concepteurs de ces systèmes est de trouver la meilleure configuration des différentes unités contenues dans un MPSoC. Cette tâche de conception, appelée exploration, vise à optimiser les performances tout en respectant les contraintes imposées par l'application et l'utilisateur. Cet objectif doit être atteint tout en garantissant un temps de conception et de test relativement court pour évaluer un nombre important d'alternatives. Les travaux entrepris dans le cadre de cette thèse visent plusieurs points :

- La co-simulation logicielle/matérielle des systèmes MPSoC à des niveaux d'abstraction élevés.
- L'estimation de performance du système offrant un premier critère de comparaison des solutions architecturales
- L'estimation de la consommation d'énergie
- L'utilisation d'une méthodologie de conception pour réduire la complexité et les temps de développement.

Cet ensemble d'objectifs, nous amène à diviser ce chapitre, sur l'état de l'art, en 4 sections. Une section est dévolue à chaque objectif. Ainsi, la section 2 introduit aux notions de co-simulation et niveaux d'abstraction. Un tour d'horizon des techniques de réduction du temps d'évaluation des performances pour les MPSoC sera présenté dans la section 3. Ensuite, nous survolons dans la section 4 les différentes méthodes existantes d'estimation de la consommation dans les systèmes sur puce. La section 5 synthétise quelques environnements de conception de MPSoC basés sur une approche dirigée par les modèles.

2.2 Le codesign et la co-simulation des systèmes sur puce

La complexité de conception des SoC augmente de plus en plus. Les raisons de cette augmentation sont :

- L'augmentation du niveau d'intégration des transistors sur une même puce.
- L'hétérogénéité des architectures proposées qui devient nécessaire pour respecter les contraintes imposées par l'application.
- La taille croissante des nouvelles applications que ces systèmes tentent de supporter.
- Un nombre important de contraintes à respecter telles que le *time to market*, le coût final du système et la fiabilité.

La conception conjointe logicielle/matérielle, ou le *Codesign*, tente d'apporter une solution efficace à ces problèmes. En 1983, Gajski et Kuhn [49] ont présenté le modèle Y (Y chart) pour décrire les étapes de conception pour les circuits VLSI (*Very Large Scale Integration*). Dans ce modèle, le système est décrit selon trois vues : structurelle (la description électronique), comportementale (la description fonctionnelle) et géométrique (le résultat physique). Aujourd'hui encore, plusieurs travaux s'inspirent de ce modèle Y pour organiser les phases de *codesign* dans un SoC. La figure 2.1 détaille cette organisation. Dans les premières phases de conception, les deux parties matérielle et logicielle sont développées parallèlement et séparément. De ce fait, la conception de la partie logicielle peut commencer avant que la conception de l'architecture ne soit complètement terminée, ce qui réduit le temps de concep-

tion. Pour une plus grande réduction du temps de simulation, les développeurs ont recours à la réutilisation de composants déjà existants. Ces derniers, connus sous le nom de blocs IP pour "Intellectual Property", peuvent provenir de développements internes à un groupe de concepteurs ou peuvent être acquis auprès d'un fournisseur tiers et intégrés au flot de conception.

Dès que les premières étapes de spécification sont franchises pour les deux parties, une phase d'association qui consiste à placer l'application sur les composants de l'architecture peut être réalisée. Les tâches de l'application seront placées sur les différents types de processeurs alors que les données et les instructions seront placées sur les différentes mémoires disponibles. Cette phase d'association est à la fois topologique (placement) et temporelle (ordonnancement). Une fois l'association terminée, la description du système est complète. Il est alors possible de vérifier les choix du concepteur sur les performances du système. En d'autres termes, nous vérifions l'adéquation de la capacité de calcul des processeurs avec les demandes des tâches ainsi que la localité des données par rapport aux processeurs qui les manipulent. Plusieurs critères doivent être observés. Le premier est le bon fonctionnement du système, c'est-à-dire s'assurer que le SoC répond toujours correctement aux entrées. Le deuxième critère est le temps d'exécution qui représente la durée nécessaire pour obtenir les sorties. En outre, des critères physiques peuvent être exigés afin de s'assurer que le comportement du SoC est compatible avec son usage, tels que la consommation électrique ou la température de fonctionnement.

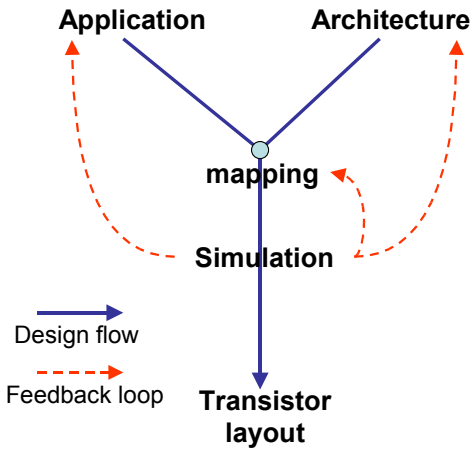


FIG. 2.1 – *Schéma d'organisation de la conception en Y*

L'approche la plus utilisée pour assurer ces critères est la simulation conjointe des deux parties logicielle et matérielle appelée aussi la co-simulation. Elle consiste à simuler l'ensemble du système et à fournir au développeur des informations sur le déroulement de l'application sur la plateforme matérielle (le temps d'exécution, la consommation, etc.). Ces informations permettent de localiser les points à optimiser/modifier dans le logiciel, dans le matériel ou dans l'association qui a été réalisée entre ces deux derniers.

Dans la littérature, le terme co-simulation possède un sens plus large que simuler le logiciel et le matériel ensemble. En effet, il regroupe différents aspects de simulation tels que multi-modèle de calcul, multi-langage et multi-niveau [95]. Au cours de cette thèse, nous nous limitons à la co-simulation logicielle/matérielle. Cependant, la problématique d'inter-

opérabilité entre différents langages et différents niveaux de simulation a été abordée dans la thèse de Lossan Bonde [26], ancien doctorant de l'équipe.

Plusieurs environnements de co-simulation logicielle/matérielle issus des universités ou d'industriels ont été proposés. La plupart de ces environnements permettent la co-simulation pour la validation des systèmes au niveau RTL. Ils offrent la possibilité de simuler en parallèle le logiciel et le matériel. La partie logicielle est exécutée en utilisant un ou plusieurs simulateurs au niveau du jeu d'instructions (ISS) et la partie matérielle est exécutée sur un simulateur matériel au niveau RTL. Comme exemple d'environnement permettant ce type de co-simulation, nous citons Mentor Graphics Seamless CVE [55] et CoWareN2C [38]. Certes ces outils offrent un bon niveau de précision mais ils sont incapables de s'adapter à la complexité des systèmes multiprocesseurs que nous visons. En l'occurrence, la co-simulation des deux parties logicielle et matérielle doit se faire dans des niveaux d'abstraction plus haut.

Généralement, un flot de conception intègre plusieurs niveaux d'abstraction pour la co-simulation du système. Ces niveaux offrent différent compromis entre la vitesse de simulation et la précision des résultats obtenus. Au fur et à mesure que la modélisation passe d'un niveau élevé à un autre plus bas, l'espace des solutions architecturales devient de plus en plus réduit. Ainsi, à la suite de nombreuses itérations entre la modification du système et la simulation, nous obtenons une solution de bonne qualité qui respecte jusqu'à un certain niveau les contraintes imposées.

2.2.1 Niveaux de simulation

L'utilisation de plusieurs niveaux de modélisation offre aux développeurs l'avantage de disposer d'un outil de simulation et d'estimation des performances dès les premières phases de conception. Même si la précision des estimations (temps d'exécution, consommation, etc.) obtenue est relativement faible, l'objectif est d'éliminer de l'espace de recherche les solutions architecturales les moins performantes. Au prix d'une augmentation des temps de simulation, le niveau de précision peut être augmenté par un raffinement dans la description du système. Plusieurs travaux de recherches proposent une classification des niveaux d'abstraction, en allant du niveau le plus haut vers le niveau le plus bas. Une multitude de définitions des niveaux de modélisation existe. Généralement, les points de différence concernent soit la terminologie soit le niveau de précision dans l'estimation des performances dans les parties calcul et communication. Dans notre travail, nous avons adopté la classification présentée par STMicroelectronics [51]. La figure 2.2 résume les niveaux d'abstraction que nous allons détailler dans la suite de cette section.

2.2.1.1 Algorithme

A ce niveau de la description, nous disposons seulement de la description de l'application sous forme algorithmique. Pour un certain nombre d'applications, l'algorithme, ou les algorithmes, sont connus et documentés. Ceci est le cas lorsque soit ces derniers ont déjà été utilisés dans la réalisation d'une précédente version du SoC soit ils font l'objet d'une norme telle que les algorithmes de codage ou décodage vidéo. A ce niveau de la description, l'architecture matérielle du système n'est pas définie et les algorithmes sont décrits dans des langages de haut niveau, comme Matlab ou C++. L'utilité de ce niveau est justifiée par la possibilité de réaliser une vérification fonctionnelle précoce de l'application via une exécution de celle-ci sur la machine hôte.

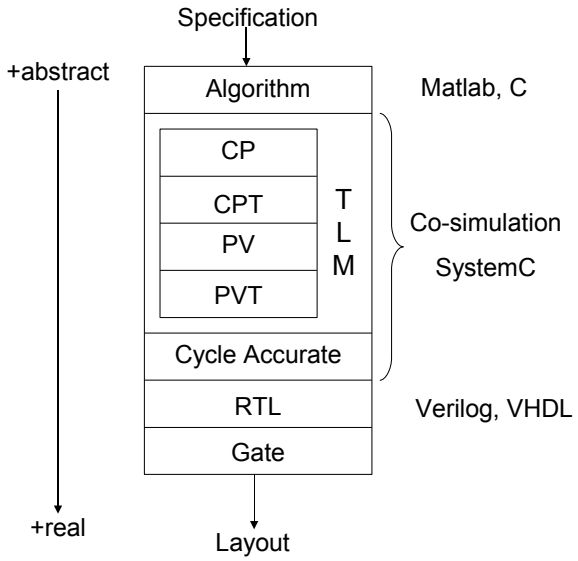


FIG. 2.2 – *Les différents niveaux d'abstraction pour la description d'un SoC*

2.2.1.2 Niveau transactionnel : TLM

Depuis les premières publications sur TLM (Transaction Level Modeling) en 2000 [50, 56], plusieurs définitions et classifications des différents niveaux de TLM ont été présentées dans la littérature [42] [31] [39] [48]. Toutes ces propositions ont en commun les points suivants :

- TLM est présenté comme une taxonomie de plusieurs sous-niveaux.
- Les aspects communications et calcul au niveau des composants sont séparés.
- Les transactions entre les modules sont simplifiées en utilisant des méthodes de communication appelées via des canaux (*Channels*) [50].

Parmi les classifications existantes, nous adoptons celle qui a été proposée par Donlin [42].

Le sous-niveau processus communicant (CP) : Dans le sous-niveau CP, l'application est découpée en tâches encapsulées dans des processus communicants. Dans CP il est possible donc de mesurer la quantité d'opérations de communication entre les tâches. Ce sous-niveau permet une première expression du parallélisme dans l'application sans se référer à l'architecture du système. Des annotations temporelles peuvent aussi être spécifiées au sous-niveau CP pour estimer le temps d'exécution. Dans ce cas, nous obtenons le sous-niveau CPT ce qui correspond à une modélisation sous forme de processus communicants temporisés.

Le sous-niveau Vue du Programmeur (PV) : C'est uniquement à ce niveau que l'architecture du système est prise en compte et où la co-simulation logicielle/matérielle devient possible. La partie matérielle est représentée sous forme de composants de différents types (processeurs, mémoires, réseau d'interconnexion, etc.). Des modules de communication sont utilisés comme mécanismes de transport des données et des politiques d'arbitrage entre les requêtes sont appliquées au niveau des ressources partagées. La description des composants doit être suffisamment précise pour que le concepteur puisse exécuter l'application de façon similaire au système final. Ainsi, le sous-niveau PV permet une première vérification du système entier (logiciel et matériel). Néanmoins, les propriétés non fonctionnelles telles que le

temps d'exécution ou la consommation d'énergie, sont soit omises soit approximées. Dans ce dernier cas, des annotations temporelles sont ajoutées à la description pour obtenir le sous-niveau PVT, qui correspond à un niveau "vue du programmeur temporisé". L'objectif dans PVT est d'atteindre des estimations de performance proches de celles obtenues par les bas niveaux de description.

2.2.1.3 Niveau Cycle Précis Bit Précis (CABA)

Au niveau CABA, le système est décrit de façon précise d'un point de vue temps d'exécution des opérations. Il permet de simuler le comportement des composants au cycle près. Au niveau calcul, une description de la micro-architecture interne du processeur (pipeline, prédiction de branchement, cache, etc.) est réalisée. Au niveau communication, un protocole de communication précis au bit près est adopté. Cette description détaillée et fine du système permet d'améliorer la précision de l'estimation des performances.

2.2.1.4 Niveau RTL

La modélisation au niveau RTL correspond à la description de l'implémentation physique du système sous forme de blocs élémentaires (bascules, multiplexeurs, UALs, registres, etc.) et à l'utilisation de la logique combinatoire pour relier les entrées/sorties de ces blocs. En général, des outils de synthèse standard sont utilisés pour obtenir le masque (*layout*) du système à partir du niveau RTL.

2.2.2 Langages de description

Comme nous venons de le voir, il est important de pouvoir modéliser un système complet, comprenant des parties logicielles et matérielles, à plusieurs niveaux d'abstraction. Pour faciliter cette approche et permettre un passage rapide entre les niveaux, il serait intéressant d'utiliser un même langage de description pour couvrir tout le flot de conception tout en restant exécutable à tous les niveaux d'abstraction. Malheureusement, ce langage n'existe pas encore [96]. Etant donné que le logiciel est typiquement développé en C/C++ et le matériel en langage HDL (Hardware Description Language), plusieurs initiatives visent à unifier le langage de description. Certaines initiatives tentent d'adapter C/C++ à la description matérielle, d'autres cherchent à étendre les HDL à la description système [83]. Une troisième voie appelle à la spécification de nouveaux langages.

Dans le cadre de cette thèse, nous nous intéressons à utiliser la co-simulation logicielle/matérielle dans la conception des MPSoC. Notre objectif est de réussir à évaluer les performances du système avant d'atteindre les phases finales de conception (au-delà de RTL). Le succès dans cet objectif nécessite tout d'abord le choix d'un langage bien adapté à la description de haut niveau, permettant ainsi l'exécution de notre système tout entier dans un temps raisonnable. Il sera intéressant aussi que le langage de description choisi permette de visualiser le comportement dynamique du système. Au cours de notre développement, le standard SystemC a été adopté comme langage de description des systèmes MPSoC. En effet, les spécificités de ce langage répondent le mieux à notre problématique de co-simulation comme nous le détaillerons dans ce qui suit.

2.2.2.1 SystemC et la librairie TLM

SystemC est une plateforme de modélisation composée de librairies de classes C++ et d'un noyau de simulation. Il introduit de nouveaux concepts (par rapport au C++ de base) afin de supporter la description du matériel et ses caractéristiques inhérentes comme la concurrence et l'aspect temporel. Ces nouveaux concepts sont implémentés par des classes C++ tels que les modules, les ports, les signaux, les FIFO, processus (threads et methods), etc. Ces classes permettent la modélisation d'un SoC à différents niveaux d'abstraction. L'initiative SystemC se développe dans le cadre de l'OSCI (Open Source systemC Initiative) [105] qui est chargée de diffuser, promouvoir et rédiger les spécifications de SystemC. Depuis décembre 2005, SystemC est standardisé auprès de l'IEEE sous le nom de IEEE 1666-2005.

Au cours de ces dernières années, SystemC a suscité un intérêt de plus en plus grand auprès des chercheurs et des industriels relativement à d'autres langages. La première raison de cet intérêt est le fait que SystemC soit basé sur le standard C++ largement utilisé pour le développement logiciel. Il devient donc possible d'utiliser les outils conventionnels de débogage qui sont généralement maîtrisés par les développeurs pour la vérification fonctionnelle. Par ailleurs, l'utilisation de SystemC pour modéliser les différents composants d'un système embarqué autorise l'utilisation d'un seul moteur de simulation. Ceci se traduit par un gain en temps de simulation et évite principalement l'utilisation des techniques de synchronisation entre différents moteurs de simulation. Les expériences réalisées par exemple chez NEC [139], montre que la conception des systèmes sur puces basée sur le standard C ou C++ offre des résultats intéressants en terme de gain en temps de développement et vérification.

La librairie TLM est parmi les principaux attraits de SystemC. Elle définit de nouveaux concepts permettant de supporter la description du matériel et du logiciel au niveau transactionnel. Nous identifions principalement les *fonctions de transport* décrites dans les modules de communication qui servent à convoyer les transactions entre les initiateurs et les cibles et vice versa. Ces transactions se présentent sous formes soit des requêtes de commande soit des requêtes de réponse. Afin de préciser le format de chaque type de requête ainsi que la liste des arguments nécessaires, un *protocole de communication* est défini. Ce dernier fourni un ensemble de *méthodes de communication* jouant le rôle d'API (Application Program Interface) pour l'utilisateur, facilitant ainsi la modélisation TLM. A titre d'exemple, pour initier une opération de lecture ou d'écriture, des appels fonctions de type *read()* ou *write()* sont utilisés. Ces fonctions sont transmises à travers les *ports* des composants connectés à des canaux de communication (appelés aussi *Channels*).

En résumé, SystemC possède des caractéristiques intéressantes pour la description de l'architecture à haut niveau. Cependant, ce langage n'est, pour le moment, pas synthétisable. Il n'est donc pas possible de l'utiliser au-delà du niveau RTL. Pour cette raison SystemC est utilisé en général pour les tâches de modélisation et de vérification rapide du système. De plus, comme il n'a été standardisé que récemment, son adoption par les grands consortiums de systèmes sur puce risque de demander un certain temps.

2.2.2.2 Autres langages de description

D'autres langages que SystemC peuvent être aussi utilisés pour une co-simulation logicielle/matérielle des systèmes sur puce monoprocesseur ou multiprocesseur. Nous citons à titre d'exemple, SpecC [129], VHDL et SystemVerilog [12]. SpecC est un langage de description et de spécification système basé sur le standard C, avec des extensions syntaxiques. Ces

extensions comprennent, par exemple, des types de données adéquats, des commandes pour supporter la concurrence et la description de machines à états. Il a été développé à l'Université de Californie à Irvine. Au-delà du langage lui-même, les travaux menés par l'équipe de D. Gajski introduisent des concepts intéressants pour la modélisation système ainsi que des techniques de raffinement [50]. Le développement de SpecC se fait dans le cadre du STOC (SpecC Technology Open Consortium). En comparaison avec SystemC, SpecC n'est pas un langage "open source" limitant ainsi les travaux de recherche qui développent des outils autour de ce langage. SpecC est plus utilisé par les consortiums de SoC asiatiques et très peu répondu en Europe.

VHDL est le langage de description matériel le plus répondu dans le monde. Il est destiné à décrire le comportement et/ou l'architecture d'un système électronique numérique. L'utilisation de ce langage dans le monde industriel ou académique est fortement liée à une phase avancée de la conception du système. VHDL permet une description du système au niveau RTL. L'intérêt d'une telle description réside dans son caractère exécutable pour vérifier le comportement réel du circuit. En outre, des outils de CAO existants permettent de passer directement d'une description en VHDL à un schéma en porte logique et par la suite au masque final. En comparaison avec SystemC, le langage VHDL ne permet pas la vérification de la conception à un haut niveau d'abstraction notamment TLM. Ainsi, dans un même environnement de co-simulation, ces deux langages peuvent être utilisés dans des phases de conception différentes suivant le niveau d'abstraction. Ils peuvent donc être considérés comme deux langages complémentaires.

Le langage SystemVerilog présente une autre approche que SystemC ou VHDL. En effet, il résulte de l'extension du langage standard de description matérielle Verilog. A ce dernier, il a été associé de nouveaux concepts afin que SystemVerilog puisse supporter la description de haut niveau et la vérification des modèles développés. C'est ce dernier point qui fait l'avantage par rapport à SystemC. En 2005, SystemVerilog est standardisé auprès de l'IEEE sous le nom de IEEE 1800-2005.

2.2.3 Synthèse

Au cours de cette thèse, nous mettons l'accent sur l'importance de la co-simulation logicielle/matérielle pour la validation comportementale des MPSoC. Face à l'obstacle des temps de simulation résultant des outils traditionnels au niveau RTL, de nouvelles approches doivent être adoptées. Aujourd'hui, l'utilisation du niveau CABA pour co-simuler les MPSoC est devenue une solution attractive par les chercheurs académiques et industriels [119, 19]. L'utilisation de SystemC comme environnement de co-simulation sera étudiée en détail dans notre travail, en particulier pour tirer partie du critère de précision offert par le niveau CABA. Cette étude fait l'objectif du chapitre suivant. Néanmoins, nous annonçons à l'avance que les temps de simulation à ce niveau ne sont pas encore satisfaisants. Pour cela, nous étudions dans la section suivante les techniques permettant d'accélérer l'évaluation des MPSoC à partir du niveau CABA.

N.B. Dans notre document nous avons utilisé le terme *simulation* pour exprimer une co-simulation logicielle/matérielle.

2.3 Techniques d'accélération de l'évaluation des MPSoC

Toute technique visant l'accélération de la simulation pour une évaluation rapide des performances des MPSoC doit tenir compte des facteurs suivants : la vitesse de simulation pour l'évaluation de chaque alternative, le niveau de précision souhaité et le niveau d'abstraction dans lequel est décrit le système. Notons, que dans notre thèse, nous nous intéressons principalement à l'aspect architecture. Nous supposons par conséquent que l'application est disponible et qu'elle est décrite dans un langage de programmation de haut niveau (C, C++, Java ou autre) pouvant être exécutée ou interprétée par le simulateur. Les techniques existantes et visant la réduction des temps de simulation au niveau CABA peuvent être divisées en 4 grandes catégories.

2.3.1 Technique d'accélération par échantillonnage

Cette première technique réduit le temps de simulation en considérant un nombre limité d'instructions de l'application. Ainsi, au lieu d'exécuter l'application dans sa totalité, quelques instructions seulement sont exécutées. Cette approche peut être abordée de 2 façons :

- En générant un programme synthétique ayant le même comportement que l'application sur la plateforme mais contenant moins d'instructions.
- En sélectionnant un nombre réduit d'intervalles d'instructions de l'application mais qui sont les plus représentatifs du comportement total de l'application.

Dans la première approche, appelée "simulation statistique", le programme est simulé afin de calculer des statistiques qui caractérisent l'exécution de l'application sur la plateforme matérielle. Habituellement, dans ces méthodes nous trouvons comme statistiques mesurées : les types et les occurrences des instructions exécutées (instructions UAL, lecture/écriture, branchement, flottant, etc.), le degré de localité de l'application et des références aux données (nombre de défauts dans les caches). Ces statistiques sont alors utilisées pour générer un programme synthétique ayant un nombre d'instructions plus réduit et permettant d'estimer rapidement le temps d'exécution [60, 98]. L'une des principales difficultés de cette approche est de tirer des statistiques sur l'inter-dépendance des tâches parallèles statiquement par un simple profilage des applications concurrentes.

La deuxième approche part du constat que dans la plupart des applications, une même partie du code est exécutée plusieurs fois. Pour réduire le temps de simulation, il suffit donc d'exécuter au niveau CABA, une seule fois les blocs d'instructions qui reviennent le plus souvent. Il est nécessaire de générer un profil de l'application au niveau des blocs de base exécutés et regrouper les intervalles d'exécution contenant les mêmes blocs de bases. Une fois, les intervalles les plus représentatifs définis, il suffit de sélectionner un échantillon de chaque groupe et d'exécuter les échantillons sur le simulateur CABA. Il a été montré que, plus grand est le nombre d'échantillons meilleures seront les estimations. Néanmoins, cette approche pose deux problèmes :

- La construction du contexte du programme avant le démarrage de chaque intervalle.
- La construction de la structure interne du processeur (contenus des caches, du pipeline, etc.).

Ces deux problèmes réduisent le facteur d'accélération potentiel de cette méthode par échantillonnage de l'application. Dans le cas des architectures multi-processeurs, la plupart des techniques existantes sont difficilement applicables du fait qu'il est impossible de connaître

à l'avance le recouvrement des codes parallèles dans les différents processeurs, rendant impossible la génération des intervalles [21, 22]. Même si récemment des solutions ont été proposées pour remédier à ces limites, il reste que leur application nécessite la réalisation d'un simulateur CABA afin d'estimer les performances [93, 23].

2.3.2 Technique d'accélération par augmentation de la granularité

La deuxième technique propose de réduire le temps de simulation en augmentant la granularité des événements considérés comme significatifs pour l'estimation des performances. En effet, habituellement, la simulation au niveau CABA des MPSoC est réalisée en commutant entre les différents contextes des processeurs à chaque cycle du simulateur. Cette méthode garantit la prise en compte des événements dès que ces derniers arrivent (cycle par cycle), ce qui offre une grande précision.

L'accélération de la simulation par l'augmentation de la granularité, utilise la spéculation sur les événements futurs. Ainsi, si l'analyse nous garantit que le prochain événement significatif pour un processeur se produira dans N cycles, la simulation pour ce processeur peut avancée de N cycles avant de commuter vers les contextes des autres processeurs. Cette diminution du nombre de changements de contexte par cycle permet par conséquent d'accélérer la simulation, puisque plusieurs instructions sont exécutées consécutivement sans changement de contexte [66]. Kim et al. [67] proposent une technique intéressante pour appliquer cette idée et permettre une co-simulation logicielle/matérielle rapide des MPSoC hétérogènes. L'idée ici est de combiner une simulation contrôlée par la trace et la synchronisation virtuelle. Dans une simulation contrôlée par la trace d'exécution (*trace driven simulation*), l'application est d'abord simulée pour enregistrer les événements considérés significatifs au niveau de chaque processeur pris individuellement. L'enregistrement se fait dans un fichier trace sur disque. Dans la deuxième étape, les fichiers traces des différents processeurs sont fusionnés afin d'aligner les événements et évaluer les performances. Cette étape nécessite l'utilisation d'un simulateur au niveau CABA qui lit les fichiers traces et avance événement par événement dans les fichiers. Un problème supplémentaire qui concerne l'espace mémoire exigé pour stocker toutes les traces est posé avec cette approche. Pour éviter ce problème, les auteurs proposent de construire les traces et les consommer à la volée. Pour cela, la trace est construite partie par partie au niveau des processeurs puis consommée par le simulateur. Une partie correspond aux événements qui se sont produits entre deux accès à la mémoire partagée.

2.3.3 Technique d'accélération par parallélisation de la simulation

Cette technique se base sur l'utilisation des machines SMP (Symmetric Multi-processor) permettant d'exécuter en parallèle un grand nombre de processus décrivant le système [16]. Ces processus sont décrits à travers des threads. L'ordonnancement de ces derniers est assuré par un contrôleur qui se charge de vérifier les threads prêts à être exécutés et de les allouer aux processeurs disponibles. Ce type de simulation, souvent appelé "multi-thread", nécessite trois types de synchronisation. Le premier type est assuré entre le contrôleur de la simulation et les processus afin de déclencher l'exécution au bon moment. Le deuxième type de synchronisation est défini entre les processeurs de la machine lors des accès simultanés à des ressources partagées. Le dernier type est spécifié par le concepteur dans les processus afin de garantir le comportement correct du composant. Les analyses formelles des per-

formances montrent que l'accélération est proportionnelle au nombre de processeurs de la machine tant que ce nombre est supérieur au nombre de processus du système. Dans le cas inverse, l'accélération peut atteindre au maximum la valeur du nombre de processeurs [16].

2.3.4 Technique d'accélération par abstraction du niveau de description

Enfin, dans la quatrième technique, l'accélération de la simulation des MPSoC est obtenue par l'utilisation de niveaux d'abstractions plus élevés que le niveau CABA. L'approche TLM, que nous avons présentée dans la section 2.2.1 fait partie de ces niveaux. En effet, l'accélération de la simulation par TLM est obtenue par la réduction des surcharges (ou overheads) associées à la synchronisation et la communication entre composants. Il faut noter, néanmoins que la plupart des travaux réalisés dans ce cadre se limitent à définir une classification des différents niveaux d'abstractions sans proposer d'outils permettant de réaliser une estimation des performances. A notre connaissance très peu de travaux se sont intéressés à l'implémentation de TLM dans le cas des MPSoC. Deux projets méritent d'être détaillés ici. Il s'agit de Pasrisha et al. [109] et Viaud al. [138].

Pasrisha et al. ont défini et implémenté un niveau abstraction au dessus du niveau RTL. Ce niveau nommé "*Cycle Count Accurate at Transaction Boundaries*" (CCATB) correspond au niveau PVT Event Accurate que nous proposons dans le chapitre 4. Au niveau CCATB, dans un souci de précision, des annotations sur les délais des opérations et le protocole de communication ont été ajoutées. Néanmoins, la mise en œuvre de CCATB nécessite la connaissance de l'état du système, comme le nombre de requêtes en lectures/écritures à un moment donné, qui n'est possible que dans le cas des architectures à bus partagé. Les niveaux d'accélération obtenus par CCATB sont assez limités.

Viaud et al. proposent aussi une méthode visant à réduire le temps de simulation en se basant sur la théorie des événements discrets parallèles (ou Parallel Discrete Event Simulation PDES). L'objectif est d'enrichir la description TLM par un certain nombre de mécanismes et de règles permettant de déduire l'horloge des processeurs sans pour autant avoir un fonctionnement cycle par cycle. Lorsque le processeur désire envoyer une requête vers la mémoire, celle-ci comprend en plus des informations habituelles, la valeur de l'horloge locale. Au niveau du réseau d'interconnexion et du module mémoire partagé cible, un paquet contenant l'horloge locale sur chaque canal d'entrée doit être reçu avant de faire avancer l'horloge locale du réseau ou de la mémoire. De cette façon, le comportement dynamique de l'application sur la plateforme matérielle peut être construit. Ainsi, il devient facile de retrouver les événements architecturaux, tels que les contentions dans le réseau d'interconnexion ou dans la mémoire partagée, nécessaire à l'estimation des performances de l'architecture. Cette approche nécessite néanmoins l'utilisation de message "nuls" afin de faire avancer les horloges ce qui peut entraîner une augmentation de la surcharge sur le simulateur. Par ailleurs, cette méthode semble difficile à généraliser dans le cas des MPSoC hiérarchiques ou distribués. Les valeurs des accélérations obtenues sont assez importantes, mais celles-ci n'ont pas été mesurées sur des applications réelles.

2.3.5 Synthèse

Parmi les techniques déjà citées, nous avons prêté une grande attention à la technique d'accélération par abstraction du niveau de description. En effet, le développement des

composants au niveau transactionnel peut se faire d'une façon plus rapide que celle au niveau CABA. Pour cela, cette technique permet d'obtenir une co-simulation du système dans les premières phases de conception en comparaison par rapport aux autres techniques. Par ailleurs, des modèles d'estimation de performance et de consommation d'énergie peuvent être intégrés dans le simulateur de haut niveau permettant ainsi une comparaison rapide des différentes alternatives architecturales. Enfin, cette technique offre la possibilité d'observer le comportement dynamique du système au cours de l'exécution de l'application.

2.4 Estimation de la consommation dans les systèmes sur puce

La réduction de la consommation d'énergie est devenue un objectif de premier plan dans la conception des SoC [20]. En effet, du fait des taux d'intégration et des fréquences d'horloges de plus en plus élevés, il devient nécessaire de concevoir des techniques pour réduire la consommation d'énergie. Dans le cas des systèmes embarqués, ces techniques sont intéressantes pour satisfaire les critères d'autonomie, de fiabilité et de coût. La solution à ces défis commence par le choix d'une technologie bien adaptée à la spécificité des systèmes embarqués en terme de consommation. Aujourd'hui, la technologie CMOS (Complementary Metal-Oxide Semiconductor) est la plus utilisée pour la fabrication de ces systèmes. En comparaison à d'autres technologies telles que la BiCMOS (Bipolar-CMOS) [52] ou la SOI (Silicon On Insulator) [25], la CMOS permet d'implanter l'électronique numérique et analogique au même niveau de performance. En outre, elle offre des meilleurs compromis entre les propriétés suivants : haute intégration, coût de fabrication et faible consommation électrique.

2.4.1 Consommation statique et Consommation dynamique

Depuis longtemps, la majeure partie de la consommation d'énergie a lieu pendant les transitions des nœuds logiques dans un circuit. Ainsi, cette source de consommation est appelée dynamique tandis que la deuxième source qui correspond à l'état de repos du circuit est appelée statique. Cette dernière était considérée pratiquement nulle. Néanmoins, cela n'est plus vrai pour les nouvelles technologies submicroniques où les courants de fuite deviennent de plus en plus importants.

En général, la consommation des circuits CMOS en terme de puissance est calculée par l'équation suivante [137] :

$$P = P_{statique} + P_{dynamique} \tag{2.1}$$

2.4.1.1 La consommation statique

La consommation statique dans un circuit est due principalement à trois types de courants de fuites dans le transistor [123, 137] :

- Le courant sous le seuil, I_{seuil} (subthreshold current) qui est considéré le composant le plus important de la consommation statique [123]. Ce courant circule entre le drain et la source du transistor quand il y a une différence de potentiel drain-source et le transistor se trouve à l'état bloquant (figure 2.3).
- Le courant à travers la grille dû à la réduction des dimensions du transistor et donc de l'épaisseur de l'oxyde de la grille, provoque la rupture de l'impédance entre la grille et

le substrat. Dans ce cas, la différence de potentiel grille-substrat donne lieu à un effet tunnel avec un transfert d'électrons (figure 2.3). Un nouveau courant de fuite apparaît entre la grille et le substrat, I_{grille} (gate leakage current), qui circule toujours quand il y a une tension grille-substrat. Ce courant peut se produire à tout moment quel que soit l'état du transistor (passant ou bloquant) [123].

– Le courant de fuites des jonctions, I_{diode} ou courant d'injection dans le substrat correspond aux fuites dans les diodes des jonctions PN (illustré dans la figure 2.3). Ce courant circule à travers ces jonctions continuellement, quel que soit l'état du transistor (passant ou bloquant) [123].

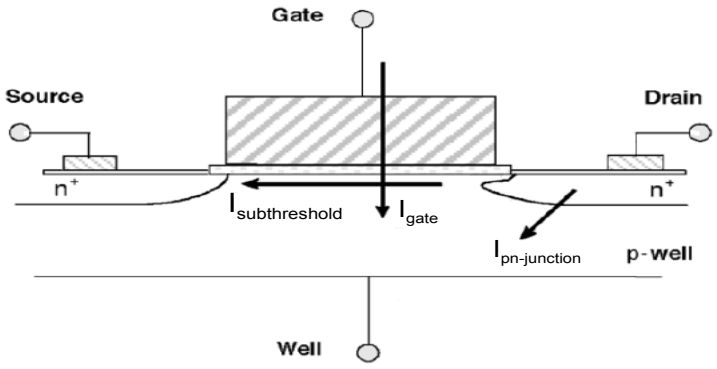


FIG. 2.3 – *Courants de fuites dans un transistor CMOS*

A température constante, la puissance statique d'un transistor dépend donc principalement de ces trois courants I_{seuil}, I_{diode} et I_{grille} et de la tension d'alimentation V_{dd} selon l'expression suivante [123] :

$$P_{statique} = I_{fuites} \times V_{dd} = (I_{seuil} + I_{diode} + I_{grille}) \times V_{dd} \quad (2.2)$$

Pour les technologies en dessus de 0.25μm, les courants de fuite étaient faibles ce qui implique que la consommation statique n'est pas prise en compte au cours de la conception des circuits. A l'opposé, dans les nouvelles technologies submicroniques ces courants occupent une grande importance pour déterminer les caractéristiques finales des circuits. Donc, il devient nécessaire de les prendre en compte. A titre d'exemple, pour le processeur Itanium d'Intel conçu avec une technologie 0.13μm, cette consommation occupe 14% de la dissipation totale du circuit [61].

La figure 2.4 (source :EETimes June 2002) donne des prévisions sur la contribution de la consommation statique en fonction de la technologie de fabrication. Selon cette source, à partir d'un processus de 70 nm l'augmentation prévue des valeurs des courants de fuite sera tellement considérable, que la consommation statique dépassera la consommation dynamique. Des nouvelles technologies devront être mises en place pour faire face à ce problème.

2.4.1.2 La consommation dynamique

La consommation dynamique se produit à chaque transition d'un nœud logique dans un circuit CMOS. Elle est due principalement à deux types de courants : le courant de court-circuit et celui de charge [115]. La puissance dynamique est exprimée avec l'équation sui-

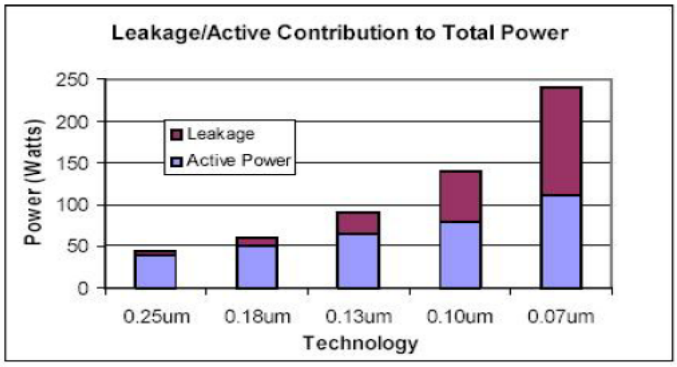


FIG. 2.4 – *Contribution de la consommation statique et dynamique dans un SoC*

vante :

$$P_{dynamique} = P_{court-circuit} + P_{charge} \tag{2.3}$$

Lors de la commutation d'une porte CMOS, il arrive un moment où des transistors de type PMOS et NMOS sont passants en même temps, ce qui crée un courant de court-circuit entre l'alimentation et la masse. La puissance de court-circuit qui en résulte est donnée par l'équation suivante :

$$P_{cc} = f \times I_{cc} \times V_{dd} \tag{2.4}$$

Où I_{cc} est la valeur moyenne du courant de court-circuit lors d'une transition, f est la fréquence des transitions et V_{dd} est la tension d'alimentation. Les études réalisées par Nose [97] nous montrent que la consommation de court-circuit peut augmenter avec l'évolution des technologies submicroniques et des tensions de seuil V_{th} très basses. Cela se produit dans les circuits fonctionnant à basse tension V_{dd} pour augmenter la vitesse de commutation. Dans ce cas, les courants de court-circuit augmentent considérablement et deviennent une source importante de consommation, environ 20% de la dissipation totale du circuit. Pour diminuer cette source, les circuits doivent être correctement conçus en utilisant des valeurs de tension V_{dd} et V_{th} bien calibrées.

La consommation de charge dynamique d'un circuit CMOS représente la puissance nécessaire pour charger la capacité C de la sortie de la porte logique à un niveau correspondant à la tension d'alimentation V_{dd}, puis pour la décharger à la masse. Cette partie est la composante la plus importante de la consommation. C'est elle que nous appelons souvent puissance dynamique. La puissance moyenne dissipée dans une porte logique est calculée de la façon suivante [115] :

$$P_{dynamique} = F \times C \times V_{dd}^2 \tag{2.5}$$

Avec $F = \frac{1}{T}$, T est le temps de réponse de la porte. Pour un système synchrone composé de plusieurs millions de portes, la puissance totale est déduite par l'équation suivante :

$$P_{dynamique} = \alpha \times N \times F \times C \times V_{dd}^2 \tag{2.6}$$

Où N est le nombre total de portes et α est l'activité du circuit exprimée en pourcentage de portes qui changent de niveau.

2.4.2 Outils d'estimation de la consommation

L'estimation de la consommation dans les systèmes sur puce nécessite des outils d'analyse capables d'évaluer les différentes sources statiques et dynamiques lors de l'exécution de l'application. Cette analyse peut être faite d'une façon plus ou moins fine, suivant le niveau d'abstraction auquel notre système est décrit. Dans la littérature, il existe une variété d'outils permettant l'estimation de la consommation à différents niveaux d'abstraction. Ces outils offrent différents compromis entre la précision de l'estimation et sa rapidité.

Dans cette thèse, l'objectif n'est pas de proposer des méthodes d'optimisation de la consommation de puissance mais plutôt de développer des outils d'estimation de la consommation pour les systèmes MPSoC à différents niveaux d'abstraction. Ces outils peuvent être utilisés dans un flot de conception de MPSoC pour réaliser une exploration architecturale et trouver l'alternative optimale en terme de performance et de consommation.

Dans la suite de cette section, nous présentons de façon synthétique les travaux d'estimation existants à plusieurs niveaux de conception. Nous positionnerons par la suite nos travaux et nos contributions par rapport aux autres travaux dans le domaine d'estimation de la consommation pour les MPSoC.

2.4.2.1 Niveau transistors

Les outils d'estimation à ce niveau de la modélisation permettent le calcul de la consommation prenant en compte les courants électriques circulant dans les transistors. La consommation dépend des caractéristiques physiques des transistors telles que les valeurs des capacités, la tension d'alimentation et de seuil, etc. Comme nous pouvons l'imaginer pour réaliser ces estimations, il est nécessaire de disposer de la description du SoC au niveau transistor. Le concepteur peut utiliser ces estimations pour choisir les cellules (transistors) qui répondent au mieux à ses objectifs [34]. Il est aussi possible d'utiliser ces estimations pour optimiser le placement/routage des cellules [18]. Parmi les outils existants à ce niveau, nous pouvons citer SPICE [104], PowerMill de Synopsys [131] ou Lsim Power Analyst de Mentor Graphics [54]. Même si les estimations obtenues à ce niveau sont très proches des valeurs du circuit réel, à l'exception des petites variations causées par l'environnement de fonctionnement telles que la température, le temps de simulation est très important. Cet inconvénient constitue un obstacle majeur pour ces outils. Dans le cas des MPSoC intégrant des centaines de millions de transistors, l'utilisation de cette approche pour évaluer la consommation totale de plusieurs centaines d'alternatives devient irréalisable.

2.4.2.2 Niveau portes logiques

La réalisation de l'estimation de la consommation de puissance à ce niveau est réalisée par l'utilisation d'une bibliothèque de portes logiques. Cette bibliothèque donne pour chaque porte ces caractéristiques en terme de consommation de puissance[3]. Cette approche permet une première réutilisation de blocs afin de réduire la complexité de conception. Le concepteur peut visualiser les courants au niveau des différentes portes logiques et ainsi contrôler la consommation du circuit. La fréquence d'horloge et les tensions d'alimentation des portes sont des exemples de paramètres qui peuvent être explorés à ce niveau. Pour ce

[3] nous dirons par la suite qu'une caractérisation des composants a été réalisée.

faire, des techniques telles que les horloges inhibées (clock gating) ou l'adaptation dynamique de la fréquence et de la tension (frequency/voltage scaling) sont appliquées. Comme exemple d'outils à ce niveau, nous trouverons, entre autres, PowerGate de Synopsys [132] et Diesel de Philips [113].

2.4.2.3 Niveau RTL (Register Transfer Level)

Afin d'accélérer la simulation du système, ce niveau de conception vise la réutilisation de blocs de granularité plus grossière que les deux niveaux précédents. La modélisation revient à décrire l'implémentation sous forme d'éléments logiques et séquentiels tels que les registres ou les bascules et à relier les entrées/sorties de ces éléments pour construire un composant du SoC. Plusieurs outils industrielles, tel que Petrol [79] de Philips, permettent l'évaluation de la consommation au niveau RTL. Les concepteurs emploient généralement la simulation avec des stimuli en entrée pour obtenir les valeurs exactes de la consommation des éléments, ce qui permet d'avoir une faible erreur d'estimation. Cette erreur peut varier entre 10 et 15% par rapport aux résultats obtenus sous SPICE [79, 124].

En résumé nous pouvons dire que la simulation d'un système MPSoC complet aux trois niveaux d'abstraction précédents souffre d'une certaine lenteur. Ce problème s'accentue avec l'utilisation d'un nombre important de processeurs. Nous sommes contraints dans ce cas de faire des estimations en composant le système sur plusieurs modules. En mesurant la consommation de cette façon, l'influence des interactions inter-blocs n'est pas prise en considération, ainsi que l'évolution de la consommation pendant le fonctionnement de l'application ne peut pas être visualisée. De ce fait, l'utilisation d'outils de modélisation au niveau transistor, portes logiques ou RTL n'est pas appropriée pour la réalisation de l'exploration des architectures MPSoC.

2.4.2.4 Niveau architectural

Pour contourner l'obstacle du temps de simulation, des niveaux d'abstraction plus élevés que ceux présentés précédemment doivent être utilisés pour l'estimation de la consommation de puissance. Plusieurs travaux de recherche proposent des solutions visant à abstraire la description de certains détails d'implémentation. Au prix d'une certaine perte de précision, l'estimation de la consommation de puissance pourra se faire en considérant des événements de niveaux de granularité plus élevés que la commutation dans un transistor ou le changement d'état d'une porte logique. Les événements identifient en réalité les activités pertinentes qui consomment une part importante d'énergie dans le niveau de modélisation considéré. Ces activités sont spécifiques à chaque type de composant (processeur, mémoires, etc.). Elles concernent généralement le type et le nombre d'instructions exécutées, les accès mémoires en lecture et en écriture, etc. L'estimation de la consommation du système est obtenue en deux étapes. La première consiste à obtenir les occurrences des activités pertinentes pendant ou après l'exécution de l'application. Dans la deuxième étape, les valeurs de ces occurrences sont transmises aux modèles de puissance pour calculer la consommation dans chaque composant du système. Pour implémenter cette méthodologie, plusieurs approches, offrant différents compromis entre la précision et rapidité, existent. Dans ce qui suit, nous détaillons les approches qui nous ont parues comme étant les plus importantes :

– **Analyse de la consommation par trace :** La première approche est proposée par l'outil AVALANCHE [77, 58]. Elle consiste à estimer la consommation en se basant sur une

analyse de la trace d'exécution de l'application. Cette dernière est exécutée en utilisant un simulateur de processeur décrit au niveau du jeu d'instructions (Instruction Set Simulator ou ISS). Les instructions exécutées servent à estimer la consommation du processeur. Les accès mémoire sont aussi détectés par un traceur de requêtes, qui envoie ces informations à l'outil Dinero (figure 2.5) pour les traiter. Ce dernier décide s'il s'agit d'un succès de cache ou bien d'un défaut de cache nécessitant l'accès à la mémoire de données ou d'instructions. Les occurrences de ces activités seront transmises à des modèles de puissance de chacun des composants pour calculer la consommation totale. Cette approche a été l'une des premières méthodes traitant le problème de l'estimation de la consommation des SoC au niveau système. Elle présente l'avantage de la précision des modèles de puissance des composants. Le modèle du processeur est composé de la consommation de toutes les instructions et des cycles d'attente dus aux défauts de cache. Le courant consommé par chaque instruction est mesuré de façon très précise avec la méthode proposée par Tiwari [135]. Nous présenterons de façon détaillée cette dernière méthode dans la section 2.4.2.5. Cette approche souffre cependant de quelques inconvénients. Le premier concerne les effets des interactions temporelles entre les composants qui ne sont pas évaluées. A titre d'exemple, les délais dus aux conflits lors des accès simultanés à la même cible (mémoire) ne sont pas pris en compte. Ce type d'événements à une importance majeure pour la précision dans l'estimation de la performance et de la consommation des MPSoC. L'absence de la modélisation des interconnexions dans cette approche présente une deuxième source d'erreur dans l'estimation.

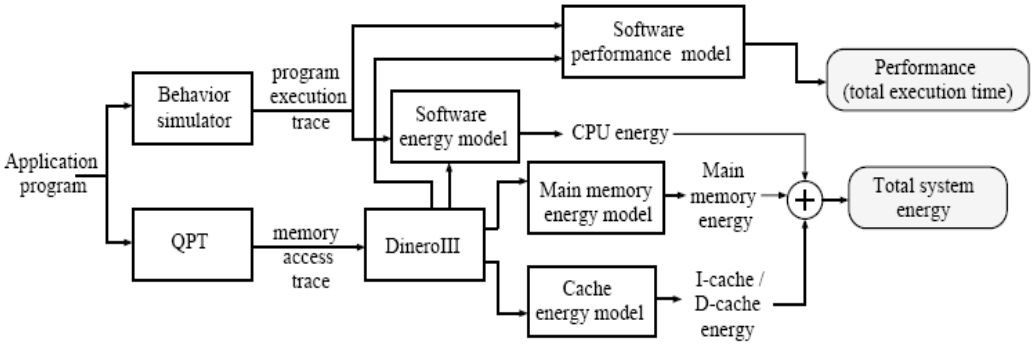


FIG. 2.5 – *Estimation de l'énergie avec l'outil Avalanche [77]*

– **Analyse de la consommation au cycle près :** Pour remédier à l'imprécision de l'approche précédente, plusieurs travaux de recherche proposent l'évaluation du système au niveau architectural en se basant sur une simulation cycle précis. L'approche consiste à modéliser le comportement des composants avec un langage de description comme C ou SystemC, et à assurer la synchronisation entre les différents composants du système. La description doit être suffisamment précise pour pouvoir simuler le comportement au cycle près des composants avec des données réelles. L'estimation des performances à ce niveau sera donnée par le simulateur d'architecture en nombre de cycles. Pour estimer la consommation, un modèle de consommation correspondant à chaque composant est intégré dans le simulateur d'architecture. Au cours de la simulation, la consommation est calculée à chaque cycle en se basant sur les occurrences des

activités pertinentes des composants. Parmi les outils décrits avec cette approche, nous citons Wattch [28] et SimplePower [142]. L'architecture du système est constituée principalement d'un processeur Superscalaire et d'une hiérarchie de mémoires. Le but de ces outils est de servir à optimiser la micro-architecture du processeur pour une application donnée. Cette optimisation concerne aussi la hiérarchie mémoire afin de trouver la meilleure configuration en performance et en consommation. Cette approche permet d'obtenir une précision acceptable et un temps de simulation raisonnable. Néanmoins, elle ne traite que des systèmes monoprocesseur. La plateforme MPARM [19], développée à l'Université de Bologne, présente un environnement de simulation des systèmes MPSoC au niveau CABA (le niveau architectural le plus précis). Elle intègre un modèle de puissance pour chaque composant, y compris le réseau d'interconnexion, permettant l'évaluation de la consommation au cycle près. Cette description offre les estimations les plus précises au prix d'une augmentation du temps de simulation comme il sera détaillé dans le chapitre suivant.

– **Analyse de la consommation par co-simulation :** L'outil Ptolemy [69] présente une troisième approche en se basant sur une co-estimation matérielle/logicielle à plusieurs niveaux d'abstraction. Cet outil permet la simulation des systèmes MPSoC dont les composants sont décrits à différents niveaux d'abstraction. Pour simuler la partie logicielle du système, un simulateur au niveau du jeu d'instructions (ISS) est utilisé. La partie matérielle est décrite et simulée au niveau RTL ou portes logiques. L'outil Ptolemy permet ainsi la simulation conjointe de tout le système par la prise en compte de la synchronisation et les transferts d'informations entre les différents simulateurs. La co-estimation de la consommation au niveau architectural est obtenue en utilisant plusieurs outils d'estimation correspondant aux différents composants. Ces outils fonctionnent de façon concurrente et synchronisée. Le principal avantage de cette approche est l'augmentation de la précision de l'estimation. Cela permet de prendre en compte les effets des interactions temporelles entre composants grâce à la synchronisation des simulateurs. Un autre avantage vient du fait que cette approche permet de suivre l'évolution de la consommation au cours du temps de tout le système. Par conséquent, elle permet de trouver le maximum de dissipation dans chaque composant en vue d'optimiser la consommation (figure 2.6). L'inconvénient majeur de cette approche est la lenteur de l'estimation provenant de la simulation de quelques composants au niveau RTL ou portes. Ce problème est amélioré en partie par les techniques d'accélération proposées, mais qui diminuent la précision de l'estimation. En outre, l'utilisation de la synthèse rapide des composants matériels dédiés peut introduire des erreurs dans les estimations.

2.4.2.5 Niveau fonctionnel ou algorithmique

Encore plus haut que le niveau architectural, nous trouvons le niveau fonctionnel. Dans ce cadre, Tiwari et al. [135] ont proposé la première méthode d'estimation de la consommation d'un programme. Cette méthode est souvent considérée comme référence dans l'estimation de la consommation des processeurs et elle est applicable théoriquement à tous types de processeurs (processeurs généralistes comme Pentium ou PowerPc, ou processeurs dédiés tels que les DSP). Sachant qu'un programme est une suite d'instructions, cette méthode consiste à cumuler l'énergie dissipée par chaque instruction. Pour évaluer les différentes instructions, il faut disposer d'abord d'un environnement expérimental réel contenant le pro-

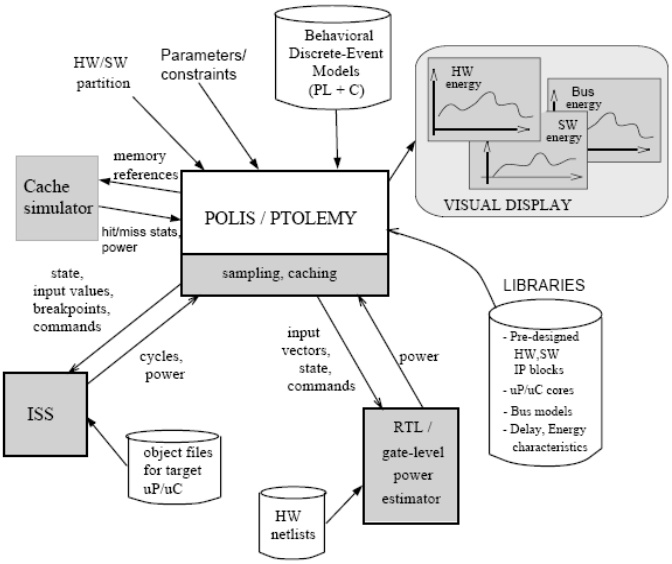


FIG. 2.6 – *Co-estimation de la consommation avec l'outil Ptolemy [69]*

cesseur à modéliser et exécuter l'instruction pour mesurer le courant moyen consommé. La précision de cette approche est améliorée en prenant compte aussi l'interaction entre deux instructions successives au niveau des valeurs des courants. L'approche de Tiwari est très précise mais nécessite beaucoup d'analyses et de mesures et par conséquent le temps de développement d'un modèle pour un processeur cible est considérable. L'outil Joule-Track [128] développé par Sinha et al. de l'Institut de Technologie du Massachusetts (USA) est un exemple d'environnement basé sur l'approche de Tiwari. L'outil permet l'estimation de la consommation à partir du code C pour les processeurs StrogARM SA-1100 et Hitachi SH-4. Sinha montre que pour un modèle de processeur simple, ne prenant en compte que la tension d'alimentation et la fréquence, cette approche peut donner des résultats relativement précis. Donc, elle peut être appliquée pour l'estimation de la consommation logicielle d'un processeur RISC simple dans un système embarqué. Dès que l'architecture du processeur devient plus complexe, cette approche n'est plus intéressante puisqu'elle génère des erreurs d'estimation importantes.

Plusieurs extensions ont été apportées aux travaux de Tiwari en vue de diminuer le temps de développement et de l'étendre pour des processeurs plus complexes. Nous citons principalement les travaux du laboratoire LESTER et leur outil SoftExplorer [125] qui a été conçu sur la base des travaux de Tiwari. La méthode d'estimation dans SoftExplorer est basée sur une analyse fonctionnelle de la consommation dans le processeur (Functional Level Power Analysis : FLPA) [71, 63]. Grâce à cette analyse, un nombre réduit de mesures est suffisant pour déterminer le modèle de consommation. Cette méthode prend en compte toutes les fonctions du processeur tel que les étages du pipeline, les unités de traitement, les mémoires internes ainsi que les communications du cache avec la mémoire. Cette méthode est illustrée dans la figure 2.7. FLPA est réalisée en 2 étapes : la définition du modèle et l'estimation de la consommation.

– La définition du modèle est basée sur l'analyse fonctionnelle du processeur. Cette étape sera faite une seule fois. Elle consiste à identifier d'abord les paramètres algo-

rithmiques pertinents agissant sur la consommation totale. Parmi ces paramètres nous citons le taux de parallélisme des instructions (ou instruction level parallelism ILP), le taux d'occupation des unités de traitement, le taux de défaut de cache du programme, etc. Le modèle de puissance du processeur sera composé d'un certain nombre de lois de consommation. Ces dernières sont calculées à partir d'un ensemble réduit de mesures expérimentales appliquées sur le processeur avec des applications élémentaires. Les lois seront ensuite décrites avec des fonctions mathématiques paramétrées en fonction des paramètres algorithmiques et d'autres paramètres de configuration tels que la fréquence, le placement des données en mémoire, etc.

– Le processus d'estimation est fait pour chaque application permettant de déduire la consommation totale. Pour ce faire, une analyse du code est faite pour extraire les paramètres algorithmiques par l'outil SoftExplorer. Ces paramètres sont par la suite insérés dans les équations du modèle de puissance pour obtenir la consommation de l'application.

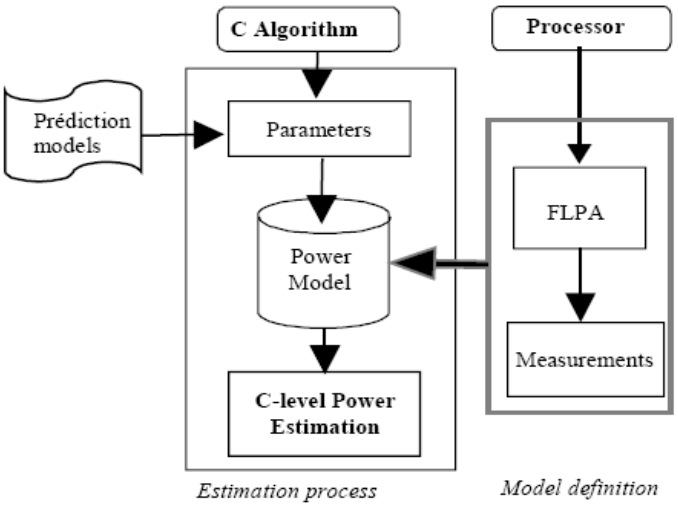


FIG. 2.7 – *L'analyse fonctionnelle de la consommation : FLPA [63]*

Cette approche a été appliquée sur plusieurs processeurs commerciaux tels que les DSP TI TMS320C6201 et TI TMS320C6701 ainsi que sur le processeur ARM7TDMI [64]. L'outil SoftExplorer dispose d'une interface graphique facilitant son utilisation. L'entrée de cet outil est le code de l'application en langage C ou en assembleur. L'outil fournit comme résultat la consommation totale ainsi que la répartition de la consommation sur les différents blocs du code de l'application. Le temps d'estimation est de quelques secondes. L'avantage majeur de la méthode FLPA est la réduction du temps de développement du modèle de puissance (de l'ordre d'un mois) pour des estimations de consommation très précises. Elle est applicable aussi bien sur des processeurs VLIW que sur des processeurs RISC. Néanmoins, elle est restreinte à des systèmes monoprocesseurs. Dans le contexte MPSoC, nous pensons que la méthode FLPA ne peut pas être directement appliquée. En effet, plusieurs autres paramètres sont mis en jeu tels que la cohérence des caches, les contentions sur le réseau d'interconnexion, etc. Par ailleurs, cette approche nécessite la disponibilité d'une plateforme intégrant le processeur cible à modéliser ainsi que les appareils de mesure nécessaires tels qu'un ampèremètre, un oscilloscope, etc.

2.4.2.6 Synthèse

Dans ce qui précède, nous avons présenté différentes méthodes et outils existants pour l'estimation de la consommation des systèmes sur puce à différents niveaux d'abstraction. Nous pouvons dire en guise de conclusion de cette section que, l'évaluation à des bas niveaux (au-dessous du niveau RTL) est précise mais coûteuse en terme de temps de simulation. L'estimation à des niveaux plus hauts est plus avantageuse puisqu'elle permet d'accélérer la phase d'exploration. Cependant, plus la modélisation s'éloigne du niveau RTL, plus il devient difficile de tenir compte des détails de l'architecture et plus grande devient l'erreur d'estimation. Dans notre travail, nous proposons une approche qui profite des avantages des différents niveaux. En effet, nous nous appuyions sur un nombre réduit d'expérimentations de bas niveau pour caractériser les activités pertinentes de nos composants en terme de consommation. Les coûts énergétiques qui en résultent seront ensuite utilisés dans le niveau architectural (CABA ensuite PVT). Ceci correspond à une méthode d'estimation hybride qui satisfait au critère de précision et de rapidité. D'un autre côté, nous avons constaté qu'une attention de plus en plus grande est accordée à la consommation statique. Cette dernière devient un facteur essentiel notamment dans les nouvelles technologies submicroniques. Pour cette raison, cette source a été prise en compte dans tous les modèles de consommation qui ont été développés dans notre travail.

La majorité des outils présentés sont destinés à l'estimation de la consommation dans des systèmes monoprocesseur. Dans la littérature, peu de plateformes MPSoC intègrent ce critère pour comparer différentes configurations architecturales. Celles qui existent sont décrites au niveau CABA offrant des estimations assez précises. De ce fait, cette approche sera utilisée dans notre environnement de simulation MPSoC. Néanmoins, les accélérations de simulation obtenues à ce niveau sont assez limitées et ne s'adaptent pas avec la complexité de ces systèmes. Face à ce défi qui sera détaillé dans le chapitre suivant, nous proposons d'aborder le critère de consommation à un niveau d'abstraction plus haut. Dans ce contexte, nous sommes parmi les premiers qui intègrent ce critère dans des systèmes MPSoC au niveau transactionnel. Nous allons démontrer que ce niveau est efficace pour l'estimation de la consommation de même que pour la vérification rapide du système.

L'estimation de l'énergie ou de la puissance se base principalement sur le développement de différents modèles de consommation. Dans les outils existants, la méthodologie de développement de ces modèles était négligée ou bien détaillée pour un composant spécifique tel que le processeur dans SoftExplorer. Nous considérons cette phase comme importante afin de permettre aux concepteurs d'adapter facilement ces modèles à des nouvelles architectures de composants. Dans notre travail, nous proposons une méthodologie pour le développement des modèles de consommation à différents niveaux d'abstraction. Cette méthodologie sera appliquée aux différents types de composants constituant le système MPSoC afin d'obtenir des modèles paramétrables et précis.

2.5 Environnements de conception basés sur une approche modèles

A ce point nous avons introduit la co-simulation logicielle/matérielle et son attrait pour accélérer l'évaluation des MPSoC. En outre, la prise en considération des critères de performance et de consommation d'énergie offre une meilleure fiabilité à l'exploration des solu-

tions architecturales. Maintenant, nous allons nous attacher à l'obtention automatique du système à co-simuler en favorisant autant que possible l'aspect de réutilisation de modèles. Nous avons donc besoin d'adopter une méthodologie de conception orientée modèles en permettant la représentation de la structure d'un composant indépendamment de son environnement d'utilisation. En effet, à partir d'une description de haut niveau du système, plusieurs implémentations peuvent être projetées telles qu'une simulation SystemC à un niveau d'abstraction donné, une description RTL du système, etc.

Une des lignes directrices suivies par nos travaux est le recours à l'Ingénierie Dirigée par les Modèles (IDM) pour organiser la conception des MPSoC et la génération automatique du système à co-simuler. L'IDM doit permettre de bénéficier à la fois d'une approche orientée modèle, permettant d'abstraire et de simplifier la représentation des systèmes, et d'une chaîne complète de compilation capable d'exploiter directement les modèles. Dans ce qui suit, nous allons introduire au début le principe et les concepts fondamentaux de l'IDM. Ensuite nous survolons quelques environnements de conception des MPSoC basés sur une approche modèle utilisant de façon implicite ou explicite les concepts de l'IDM.

2.5.1 Ingénierie dirigée par les modèles

Afin de pouvoir comprendre et agir sur le fonctionnement d'un système, il est nécessaire de disposer d'un modèle de ce dernier. Un modèle est une simplification et/ou une abstraction de la réalité. Modéliser consiste à identifier les caractéristiques intéressantes ou pertinentes d'un système afin de pouvoir l'étudier. Au nombre des évolutions dans le développement des logiciels, il faut citer l'apparition de plusieurs méthodes de modélisation : Merise [133], SSADM (Structured Systems Analysis and Design Methodology) [45], UML (Unified Modeling Language) [100], etc. Ces méthodes vont permettre d'appréhender le concept de modèle en informatique. Elles proposent des concepts et une notation permettant de décrire le système à concevoir. En général, à chaque étape du cycle de vie du système, un ensemble de documents généralement constitués de diagrammes permettent aux concepteurs, développeurs, utilisateurs et autres entités impliquées de partager leur perception du système. Ces méthodes de modélisation ont été parfois critiquées pour leur lourdeur et leur manque de souplesse face à l'évolution rapide du logiciel. Elles ont conduit à la notion de modèles contemplatifs qui servent essentiellement à communiquer et comprendre, mais reste passif par rapport à la production. Ainsi, après un demi-siècle de pratique et d'évolution, on constate aujourd'hui que le processus de production de logiciels est toujours centré sur le code.

L'Ingénierie Dirigée par les Modèles (IDM) vient pallier la déficience des méthodes traditionnelles de modélisation. Elle œuvre à fournir un cadre de développement logiciel dans lequel les modèles passent de l'état passif (contemplatif) à l'état actif (productif) et deviennent les éléments de première classe dans le processus de développement des logiciels.

2.5.1.1 Principe et concepts fondamentaux de l'IDM

Notons d'abord que le terme Ingénierie Dirigée par les Modèles (IDM) possède plusieurs synonymes tels que Model Driven Engineering (MDE), Model Driven Development (MDD), Model Driven Software Development (MDSD), etc. L'IDM est encore un domaine de recherche récent, dynamique et en pleine évolution. Cela se traduit par une pluralité d'idées, de concepts et de terminologies compétitives qui tendent à créer une confusion dans ce do-

maine. Dans cette section, nous tentons de dégager le principe et les concepts de base formant le socle de l'IDM et qui sont généralement acceptés par toute la communauté.

2.5.1.2 Principe de L'IDM

L'IDM représente une forme d'ingénierie générative, par laquelle tout ou partie d'une application est générée à partir de modèles dans un processus de conception. Cette approche offre un cadre méthodologique et technologique qui permet d'unifier différentes façons de faire dans un processus homogène. Il est ainsi possible d'utiliser la technologie la mieux adaptée pour chacune des étapes du développement, tout en ayant un processus de développement global et unifié. Le principe de l'IDM consiste à utiliser intensivement et systématiquement les modèles tout au long du processus de développement logiciel. Les modèles devront désormais non seulement être au cœur même du processus, mais également devenir des entités qui peuvent être interprétées par les machines. Un système peut être vu sous plusieurs angles ou points de vue. Les modèles offrent la possibilité d'exprimer chacun de ces points de vue indépendamment des autres. Cette séparation des différents aspects du système permet non seulement de se dégager de certains détails, mais permet également de réaliser un système complexe par petits blocs plus simples et facilement maîtrisables. Ainsi, on pourrait utiliser des modèles pour exprimer les concepts spécifiques à un domaine d'application et des modèles pour décrire des aspects technologiques. Chacun de ces modèles est exprimé dans la notation ou le formalisme les plus appropriés.

Dans le cadre de la conception des systèmes embarqués, l'approche IDM ouvre de nouvelles perspectives. Son principe de séparation des différents aspects du système, augmente la réutilisation et aide les membres de l'équipe de conception à partager et s'échanger leurs travaux indépendamment de leurs domaines d'expertise. Ce concept est particulièrement intéressant dans le domaine de conception des systèmes sur puce où les deux technologies logicielle et matérielle vont interagir pour la définition du comportement global du système.

2.5.1.3 Concepts fondamentaux de l'IDM

De façon générale, l'IDM peut être définie autour de trois concepts de base : les modèles, les méta-modèles et les transformations.

Modèle : Un modèle est une abstraction d'un système étudié, construite dans une intention particulière. Il doit pouvoir être utilisé pour répondre à des questions sur le système. Comme le montre cette définition, la notion de modèle va de pair avec la notion de système. En effet, un modèle est conçu pour représenter quelque chose que l'on désigne ici par le terme système. Il est important de noter qu'en soi la notion de modèle apportée par l'IDM n'a absolument rien de novateur. Comme l'a fait remarquer Favre [46], on peut considérer que la notion remonte à plusieurs millénaires. Cette notion était déjà présente en informatique dans les années 1970.

Méta-modèle : Un méta-modèle est un langage qui permet d'exprimer des modèles. Il définit les concepts ainsi que les relations entre concepts nécessaires à l'expression de modèles. Un modèle écrit dans un méta-modèle donné sera dit conforme à ce dernier. La relation entre modèle et méta-modèle (conforme à) peut être par analogie aux langages de programmation comparée à la relation entre une variable et son type ou un objet et sa classe (dans le cas des langages objets).

Transformation de modèles : Comme nous l'avons déjà indiqué, l'IDM œuvre à fournir un cadre de développement logiciel dans lequel les modèles passent de l'état passif (contemplatif) à l'état actif (productif). Un modèle est productif soit parce qu'il est directement exécutable par une machine, soit parce qu'il permet de produire d'une façon directe ou indirecte des modèles exécutables. Le second cas suppose la possibilité de pouvoir réaliser des opérations sur le modèle pour produire un autre modèle exécutable. Cette opération sur les modèles est connue dans l'IDM sous le concept de transformation de modèles.

Hubert Kadima [65] nous donne cette définition : Une transformation de modèles est la génération d'un ou de plusieurs modèles cibles à partir d'un ou de plusieurs modèles sources conformément à une définition de transformation. Cette définition est exprimée par un ensemble de règles de transformation qui décrivent globalement comment un modèle décrit dans un langage source peut être transformé en un modèle décrit dans un langage cible. La figure 2.8 illustre ce processus principal de l'IDM.

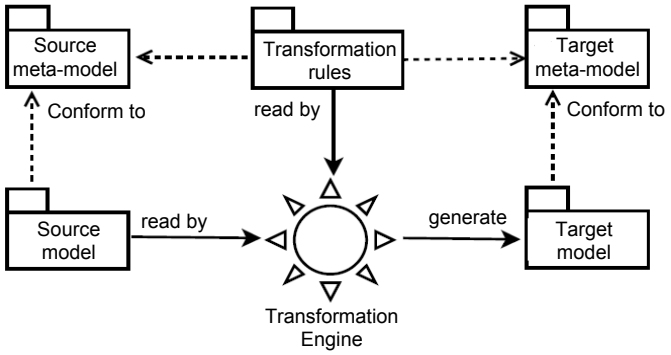


FIG. 2.8 – *Processus de transformation de modèles*

2.5.2 Gaspard

Gaspard (Graphical Array Specification for Parallel and Distributed Computing) [140] est un environnement basé sur le schéma en Y de Gajski. Il est orienté pour les applications de traitement de signal intensif. Ce type d'application est parmi ceux qui nécessitent le plus de puissance de calcul et se prête généralement bien à la parallélisation des traitements. Le traitement de signal est aussi parmi les types d'applications les plus utilisés dans les systèmes embarqués. Gaspard vise la génération de différents codes à partir d'un même placement d'une application sur une architecture à savoir :

- La génération du code VHDL correspondant à un accélérateur matériel capable d'exécuter l'application initialement modélisée à haut niveau.
- La génération de langages synchrones déclaratifs (tels que Lustre ou Signal) permettant de vérifier formellement la modélisation d'une application.
- La génération de langages procéduraux tels que Fortran/OpenMP offrant l'exécution concurrente de différents processus sur une architecture multiprocesseur (architectures à mémoire partagée dans l'état actuel de Gaspard).
- Enfin, la génération de code SystemC permettant la simulation du comportement d'un SoC à différents niveaux d'abstraction.

En effet, Gaspard regroupe des experts de différents domaines ayant comme point commun l'usage de l'IDM comme méthodologie de conception en allant de la modélisation en UML jusqu'à la génération de code. Notre travail dans Gaspard consiste à générer du code SystemC pour la co-simulation logicielle/matérielle d'un système sur puce multiprocesseur modélisé à haut niveau. Gaspard doit permettre au concepteur de modéliser indépendamment les différents aspects de notre système ; application, architecture et association. A partir de ces informations et en se basant sur les recommandations de l'IDM, plusieurs transformations de modèles doivent être développées afin de générer le code du MPSoC complet, à la fois matériel et logiciel. Par suite, ce code nous permettra des simulations à des niveaux de plus en plus précis. La figure 2.9 illustre cette organisation du développement, et fait le rapprochement avec le flot de conception en Y.

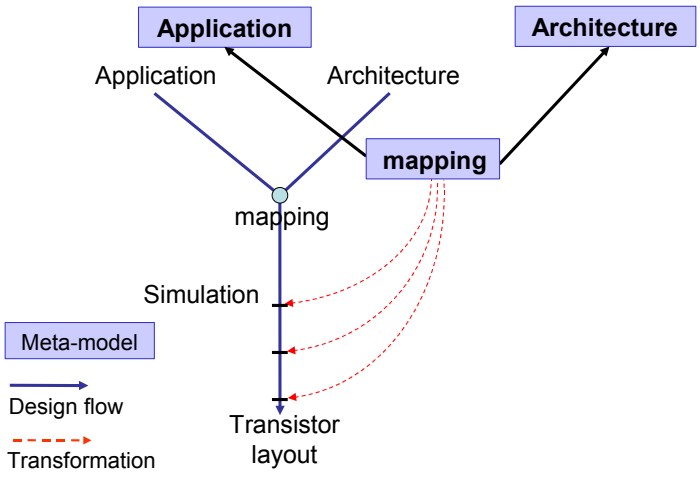


FIG. 2.9 – *Schéma du flot de conception en utilisant l'environnement Gaspard*

Au début de nos travaux de thèse, seul le méta-modèle permettant à l'utilisateur de spécifier un MPSoC était défini, les transformations étant encore uniquement des perspectives. Ce méta-modèle a pour origine les travaux d'Arnaud Cuccuru lors de son doctorat dans l'équipe. Il est en partie inspiré de standards OMG tels que SPT (Schedulability, Performance, and Time) [103] pour représenter l'architecture matérielle, SysML (System Modeling Language) [102] pour la mise en œuvre de l'association, et UML pour l'approche composant. Notons au passage que le méta-modèle Gaspard fait partie à la base du tout jeune standard MARTE (Modeling and Analysis of Real-Time and Embedded systems) [120]. Notre méta-modèle pour l'expression des MPSoC sera détaillé dans le chapitre 6. Gaspard est également basé sur le langage Array-OL (Array Oriented Language) [41, 27] comme modèle de calcul dédié aux applications de traitement de signal intensif. Comme nous allons le montrer dans les deux derniers chapitres de cette thèse, notre environnement est capable de générer du code SystemC présentant le système à co-simuler au niveau PVT. Ultérieurement, d'autres niveaux (CABA et RTL) seront intégrés dans Gaspard. Un autre attrait essentiel de notre environnement est l'intégration des outils d'estimation de performance et de consommation d'énergie.

2.5.3 Profil UML 2.0 pour SystemC

Différents chercheurs de l'Université de Milan, de l'Université de Catane et du consortium STMicroelectronics ont réuni les efforts pour réaliser le profil UML 2.0 pour SystemC [121]. Leur objectif est d'arriver à générer du code exécutable SystemC à partir d'une description de haut niveau. Différentes notations UML ont été associées aux concepts SystemC comme le montre la figure 2.10. Cette notation inclut la structure, les communications, les types de données, le comportement, la synchronisation, etc. La description d'un modèle de composant avec ce profil peut contenir les moindres détails sur le comportement. L'utilisateur peut modéliser le comportement de son composant avec une machine à états finis. Néanmoins pour des architectures complexes, ceci peut causer un temps de modélisation supérieur au temps de codage à la main. Un environnement [122] a été conçu en vue de la génération automatique du code à partir du modèle de haut niveau. Il contient trois transformations de modèles indépendantes, chacune ramène à un type de génération de code. La première transformation permet une génération squelette (*skeleton*) contenant seulement la structure statique du système. Ce type de génération est fréquemment adopté par les outils de transformation. La deuxième transformation offre une génération partielle intégrant une spécification plus complète du système. Le comportement peut être modélisé en utilisant une machine à états finis, un diagramme d'activités, etc. La dernière transformation aboutit à une génération totale du code SystemC. En comparant cet environnement avec Gaspard, ce dernier ne permet pas la génération du code d'un composant matériel ou logiciel mais plutôt favorise la réutilisation de ces composants à partir des bibliothèques.

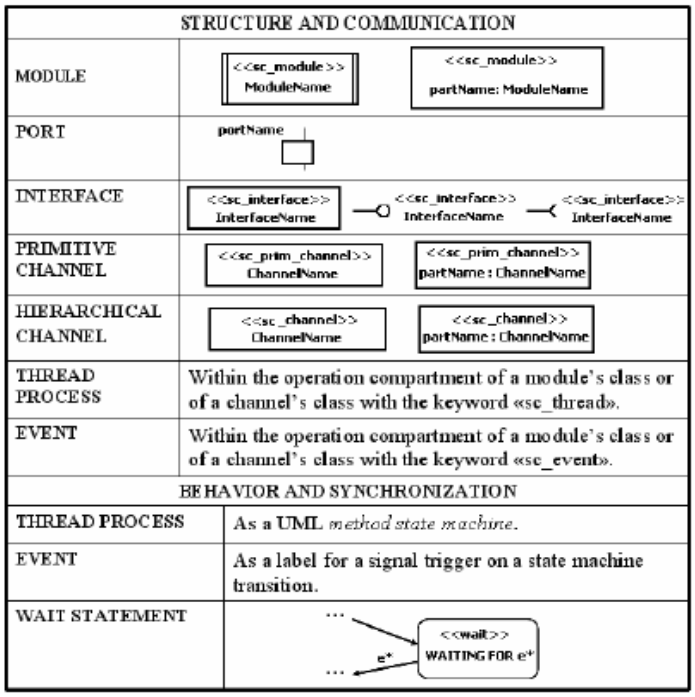


FIG. 2.10 – *Notation UML pour les concepts SystemC*

2.5.4 ROSES

L'environnement ROSES [33] permet une description hétérogène des systèmes MPSoC. ROSES englobe plusieurs modèles de description et plusieurs modèles de simulation à des niveaux d'abstraction différents et/ou décrits avec divers langages. Le cadre qui offre cette possibilité est l'utilisation du concept de l'architecture virtuelle. Cette dernière est constituée principalement de modules virtuels, ports virtuels et canaux virtuels. Le langage de description utilisé pour l'architecture virtuelle est VADeL permettant de décrire les systèmes d'une manière hétérogène. ROSES permet la co-simulation logicielle/matérielle avec plusieurs simulateurs en générant automatiquement les interfaces logicielles, matérielles et de co-simulation. Pour ce faire, 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 (figure 2.11). ROSES est un exemple d'environnement qui permet la génération du code de bas niveau (RTL) à partir d'une description de haut niveau. Les différents concepts de l'IDM sont utilisés de façon implicite dans ROSES. Ce dernier utilise un langage de description spécifique (textuel) tandis que la recommandation de l'IDM est d'utiliser un langage visuel tel que UML. Les outils de génération de code jouent le rôle des transformations de modèles permettant de passer d'une description abstraite à une autre plus détaillée.

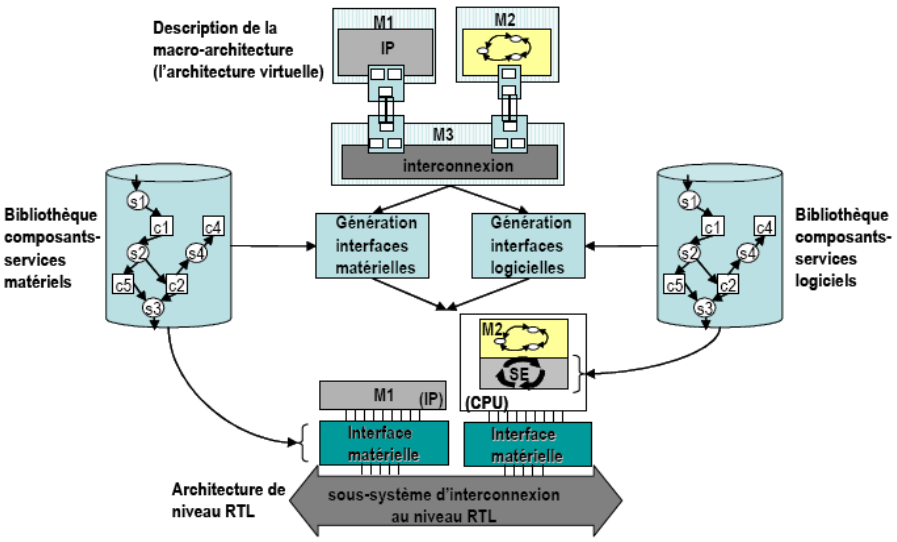


FIG. 2.11 – *Le flot de conception logiciel/matériel dans ROSES*

2.5.5 Le projet OMEGA

Le projet Européen OMEGA[4] s'inscrit dans le cadre d'une approche basée sur des modèles dédiés au développement des systèmes temps réel embarqués critiques. La modélisation est basée sur le langage UML. Pour exprimer les exigences spécifiques au domaine du temps réel, un profil UML a été défini [99]. Ce dernier représente un raffinement formel du profil UML standard Scheduling, Performance and Time (SPT). La boîte à outils IFx

[4]Projet Européen OMEGA (IST-33522). Voir aussi http ://www-omega.imag.fr

(figure 2.12) a été développée pour supporter l'utilisation de ce profil (dit profil OMEGA). Elle est utilisée en complément d'éditeurs UML compatibles avec le format d'échange XMI, tels que Rational Rose ou I-Logix Rhapsody. Le modèle UML est transformé ensuite en un modèle IF (langage spécifique), ce qui permet de simuler et de valider formellement des propriétés dynamiques des modèles. Pour que ce passage par IF soit transparent pour l'utilisateur, les fonctionnalités d'analyse et de vérifications ont été remontées au niveau de la spécification UML à travers une interface fournie par IFx. Ainsi, la connaissance de l'outil IF utilisé en dessous de l'interface n'est pas essentielle. L'inconvénient de cet environnement est qu'il ne supporte pas la description de l'architecture d'un système sur puce et reste restreint à un domaine spécifique d'application.

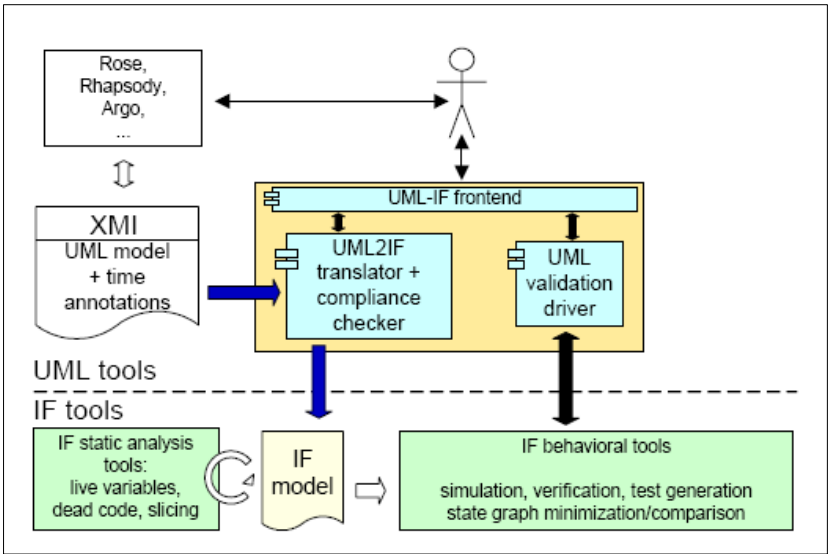


FIG. 2.12 – *Architecture de la boite à outils IFx*

2.5.6 Autres propositions

D'autres environnements, basés sur une approche modèles, existent dans la littérature tels que Milan [91], Ptolemy [40], Artemis [116], Metropolis [16], GRACE++ [53], etc. Ils offrent une sémantique de description respectant un méta-modèle de référence, ensuite proposent des outils permettant l'analyse, la simulation ou la synthèse du système. Les différences entre ces environnements concernent le langage de spécification de haut niveau, le modèle de calcul, le type d'application envisagée et le niveau d'abstraction du code généré.

2.6 Conclusion

Dans ce chapitre, nous avons présenté une analyse détaillée des aspects qui seront étudiés dans cette thèse à savoir : la co-simulation, l'accélération de la simulation, l'estimation de performance, l'estimation de la consommation d'énergie et la méthodologie de conception. Au début, il semble difficile d'aborder ces différents aspects ensemble. Chacun d'entre eux représente un domaine de recherche nécessitant une étude plus approfondie. Néanmoins, l'analyse dans ce chapitre a montré une forte dépendance entre ces aspects. En ef-

fet, l'estimation de performance ne peut pas être étudiée indépendamment du type de co-simulation réalisée, l'estimation de la consommation dépend de la description du système et enfin la méthodologie de conception regroupe les concepts et les outils développés par l'utilisateur. En conclusion tous ces aspects se réunissent ensemble pour réussir l'objectif final : résoudre la complexité de conception des MPSoC et réduire l'effort de développement. Dans le chapitre suivant, nous étudierons de façon plus détaillée la description des MPSoC au niveau CABA afin de retirer les critères de précision qui peuvent être remontés vers des niveaux d'abstraction plus haut.

Chapitre 3

Simulation et estimation de performance des MPSoC au niveau CABA

3.1 Introduction

Comme ceci a été évoqué dans le chapitre d'introduction de cette thèse, le temps de simulation nécessaire pour évaluer les différentes alternatives des systèmes MPSoC au niveau RTL devient l'obstacle majeur dans la phase de conception. Pour réduire ce temps de simulation, plusieurs travaux de recherche ont été consacrés à la modélisation des systèmes MPSoC au niveau CABA (Cycle Accurate Bit Accurate). A ce niveau, les modèles de composants offrent l'avantage de garder la précision au bit et au cycle près tout en permettant une simulation plus rapide. Habituellement pour passer du niveau RTL au niveau CABA, les détails architecturaux au niveau transfert de registres sont supprimés. Ce niveau nécessite néanmoins une description détaillée de la micro-architecture du système afin de simuler un comportement correct à chaque cycle. Comme nous le verrons dans ce chapitre, la simulation d'un système MPSoC au niveau CABA nécessite par conséquent un effort considérable pour le développement des composants matériels et pour interconnecter ces composants. Par ailleurs, le temps de simulation pour évaluer les performances de chaque alternative architecturale demeure important. Toutefois, une simulation préliminaire en CABA de certains composants s'avère nécessaire si l'on désire intégrer dans les niveaux de simulation à des niveaux d'abstraction plus élevés, des informations pertinentes sur les performances et la consommation d'énergie.

Ce chapitre a pour but essentiel d'étudier la problématique du temps de simulation au niveau CABA. Nous utiliserons la bibliothèque de composants SoCLib pour réaliser une étude de cas de la description d'un MPSoC au niveau CABA. Cette étude détaillée s'avère nécessaire pour deux principales raisons :

- La maîtrise de la modélisation au niveau CABA va nous aider à décrire des systèmes MPSoC à des niveaux d'abstraction plus élevés tout en gardant la possibilité d'obtenir des estimations de performance précises (chapitre 4).
- Cette maîtrise est aussi nécessaire pour développer des modèles d'estimation d'énergie au niveau de chaque composant pour l'évaluation de la consommation totale du système (chapitre 5).

Ce chapitre est structuré comme suit. La section 2 détaille la description au niveau CABA en se basant sur la théorie des "machines à états finis (FSM) communicantes synchrones". La section 3 présente les modèles des différents composants matériels que nous avons utilisés pour composer notre plateforme MPSoC au niveau CABA. La méthodologie d'estimation des performances à ce niveau est détaillée dans la section 4. Enfin, la section 5 présente une synthèse de ce chapitre et donne un aperçu sur la méthode d'intégration des composants CABA dans notre flot de conception Gaspard.

3.2 Description d'un système au niveau CABA à l'aide de FSM

La description d'un SoC au niveau CABA modélise le comportement du système à chaque cycle de façon similaire au niveau RTL. En effet, la modélisation au niveau CABA est basée sur la théorie des "machines à états finis (FSM) communicantes synchrones" (*Synchronous Communicating Finite State Machines*) [126, 73]. Une FSM est constituée d'états et de transitions entre états. Son comportement est conditionné par des variables d'entrée. L'automate passe d'un état à l'autre suivant les transitions décrites dans la FSM et en prenant en compte les valeurs des variables d'entrée. Une FSM possède par définition un nombre fini

d'états.

Dans une description comportant plusieurs FSM, la communication entre ces dernières est réalisée en partageant des variables. Dans un système embarqué monoprocesseur ou multiprocesseur, chaque composant est décrit avec une ou plusieurs FSM suivant les fonctionnalités réalisées. L'ensemble des signaux qui relient les différents composants représente les variables partagées pour l'échange de données. Les deux parties calcul et communication sont exécutées à des périodes de temps discrètes. Dans le cas d'une simulation CABA, une période de temps représente un cycle d'horloge du système global.

Ainsi, pour décrire le comportement d'un composant matériel, une ou plusieurs FSM sont spécifiées et exécutées en parallèle. La figure 3.1 représente une seule FSM qui contient trois fonctions : (*Transition, Moore et Mealy*) [126]. La fonction (*Transition*) calcule le prochain état du composant en se basant sur l'état actuel et sur les valeurs des signaux d'entrée. La fonction (*Mealy*) détermine les valeurs des signaux de sortie qui dépendent des signaux d'entrée et de l'état actuel du composant. Enfin, la fonction *Moore* calcule les valeurs des signaux de sortie qui dépendent seulement de l'état actuel du composant. Dans l'implémentation des FSM que nous avons expérimentées, les spécifications du simulateur SystemC, telles que l'exécution guidée par événement discret (*event driven*), ont été utilisées pour obtenir un simulateur précis au niveau cycle.

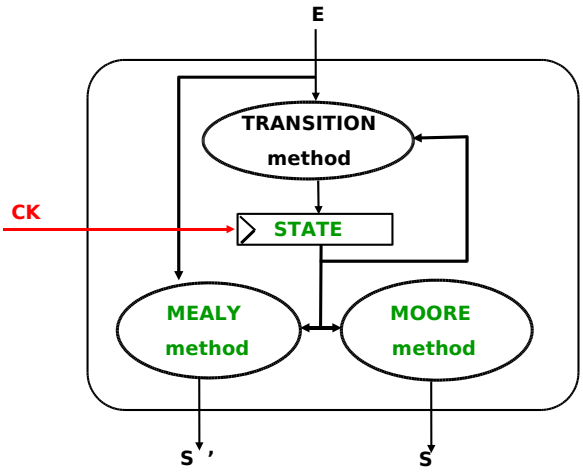


FIG. 3.1 – *modèle d'un composant [47]*

3.3 Modèles de composant au niveau CABA pour la conception des MPSoC

La plateforme SoCLib est une bibliothèque de composants matériels qui permet de réaliser un simulateur de MPSoC au niveau CABA. Les composants de SoCLib utilisent des FSM communicantes synchrones. A travers cette plateforme, une architecture MPSoC est obtenue par instanciation de composants matériels. Ces derniers sont connectés par des signaux et respectent le protocole de communication VCI (*Virtual Component Interface*) [29]. Dans cette section, nous allons commencer par décrire en détail ce protocole. Par la suite, les composants de SoCLib que nous avons utilisés pour concevoir notre plateforme d'expérimentation

au niveau CABA seront décrits. Comme de nouveaux composants ont été intégrés à la bibliothèque de SoCLib durant cette thèse, nous profiterons de cette section pour les décrire brièvement. Ces derniers permettent de concevoir des systèmes MPSoC hétérogènes. Rappelons que l'objectif de cette section est de montrer que la description détaillée au niveau CABA d'un MPSoC nécessite un effort de développement considérable et une connaissance détaillée du comportement des composants. Cette tâche est sûrement difficile néanmoins nécessaire car elle permettra par la suite la modélisation de l'architecture à des niveaux d'abstraction plus élevés (chapitre suivant).

3.3.1 Le protocole de communication VCI

Avec l'augmentation de la complexité des architectures, il devient crucial de pouvoir réutiliser des modèles de composants disponibles. Cette approche facilite la conception et réduit le temps de développement. Pour arriver à cet objectif, il est indispensable d'utiliser un standard d'interface entre les composants. Cette approche facilite en outre l'interconnexion des modules. Il existe sur le marché plusieurs standards de protocoles.

- le standard AMBA proposé par la société ARM [14]
- le standard coreconnect proposé par la société IBM [36]
- le standard VCI (Virtual Component Interface) proposé dans le cadre du consortium VSIA[5] (*Virtual Socket Interface Alliance*).

C'est ce dernier standard qui a été adopté dans la bibliothèque SoCLib. Le protocole VCI a été par conséquent utilisé dans cette thèse. Dans cette section, nous allons détailler le fonctionnement de VCI. Nous pensons que ces détails sont nécessaires pour comprendre la manière avec laquelle l'estimation de performance au niveau PVT sera réalisée (chapitre 4) ainsi que l'estimation de la consommation (chapitre 5). Dans cette thèse, nous supposons que les composants matériels utilisent le même protocole de communication. Une approche pour résoudre le problème d'interopérabilité entre des protocoles de communications différents est présentée dans la thèse de Lossan Bonde [26]. La solution proposée est basée sur l'utilisation d'une méthodologie dirigée par les modèles.

Norme VCI

La norme VCI définit une communication point-à-point asymétrique entre un *initiateur* et une *cible*. L'initiateur émet des requêtes (de lecture ou d'écriture, simples ou par paquets) et la cible y répond. La norme VCI se décline en trois versions plus ou moins élaborées selon la nature et le rôle des composants [57, 29]. Ces trois versions sont : la Basic VCI (*BVCI*), la Peripheral VCI (*PVCI*) et l'Advanced VCI (*AVCI*). BVCI définit les signaux de base pour permettre une communication entre les initiateurs et les cibles. PVCI, comme son nom l'indique, permet de connecter des périphériques d'entrées/sorties au réseau d'interconnexion. Enfin, AVCI permet à un même initiateur d'émettre plusieurs requêtes consécutives avant de recevoir les paquets réponses. C'est cette dernière version qui a été adoptée pour interconnecter les composants de SoCLib.

Les différents signaux dans *AVCI* sont résumés dans le tableau 3.1. Le nombre entre parenthèses correspond au nombre de bits du port, et b représente le nombre d'octets par donnée. Dans ce tableau $b \in \{1,2,4\}$, $e \in \{1,2,3\}$ *et* $n \in \{0..64\}$. Dans la plateforme SoCLib, nous avons $b = 4$, $e = 3$ *et* $n = 32$.

[5] http ://www.vsia.org

Nom (taille en bits)	Origine	Description
CMDACK (1)	Initiateur	Requête acceptée
CMDVAL (1)	Initiateur	Requête valide
ADDRESS (n)	Initiateur	Adresse
BE (b)	Initiateur	Byte Enable : octets concernés par la requête
CMD (2)	Initiateur	Type d'opération : lecture ou écriture (Word, Half word, Byte)
CONTIG (1)	Initiateur	Adresses contigues
WDATA (8b)	Initiateur	Donnée à écrire
EOP (1)	Initiateur	Marqueur de fin de paquet
CONS (1)	Initiateur	Adresses constantes
PLEN (8)	Initiateur	Longueur du paquet en octets
WRAP (1)	Initiateur	Adresses repliées
CFIXED (1)	Initiateur	Définition d'une chaîne
CLEN (8)	Initiateur	Nombre de paquets chaînés
SCRID (8)	Initiateur	Numéro d'initiateur
TRDID (8)	Initiateur	Numéro de thread
PKTID (8)	Initiateur	Numéro de paquet
RSPACK (1)	Cible	Réponse acceptée
RSPVAL (1)	Cible	Réponse valide
RDATA (8b)	Cible	Donnée lues
REOP (1)	Cible	Marqueur de fin de paquet
RERROR (e)	Cible	Signalisation des erreurs
RSCRID (8)	Cible	Numéro d'initiateur
RTRDID (8)	Cible	Numéro de thread
RPKTID (8)	Cible	Numéro de paquet

TAB. 3.1 – *Ports de la norme VCI*

L'interface *AVCI* est constituée de deux paquets de signaux : les signaux de commande (*CMD*) et les signaux pour acheminer la réponse (*RSP*). Ces deux paquets fonctionnent de façon asynchrone. Ainsi, il est possible d'envoyer plusieurs requêtes consécutives avant de recevoir les premières réponses. Ces dernières peuvent en plus arriver dans un ordre différent de celui des requêtes. Pour permettre cela, dans le protocole *AVCI*, de nouveaux signaux ont été ajoutés à BVCI. Il s'agit de SCRID, TRDID, PKTID, RSCRID, RTRDID et RPKTID. Les signaux CMDACK, CMDVAL, RSPACK et RSPVAL sont utilisés pour établir un protocole de communication de type *handshake*.

Le fonctionnement du protocole *AVCI* peut être décrit comme suit : un *initiateur* envoie un signal CMDVAL à une *cible* pour l'informer que des données valides sont prêtes à être émises. Ces données peuvent être mises dans un ou plusieurs paquets et chaque paquet peut contenir un ou plusieurs mots. La cible, à son tour, informe l'initiateur par le signal RSPACK de son état (libre ou occupée). Si la cible est libre, le cycle de transfert commence. Le nombre de cycles nécessaires pour terminer la transaction dépend du nombre de mots dans le paquet de données.

Le chronogramme de la figure 3.2 illustre des séquences de transfert où la cible traite des requêtes de différentes tailles. Comme nous pouvons le constater sur ce chronogramme,

quand l'initiateur a une requête à soumettre, il active le signal CMDVAL pour informer la cible. Le transfert est déclenché quand la cible active son signal RSPACK. Le signal EOP valide l'arrivée du dernier mot mettant fin au transfert. Le type d'opération lecture ou écriture est déterminé par le signal CMD. Pour implémenter le protocole de communication présenté, dans chaque interface de type VCI d'un composant initiateur ou cible nous avons besoin de deux FSM. La première pour gérer les signaux de commande (FSM VCI_CMD) et la seconde pour gérer les signaux de réponse (FSM VCI_RSP).

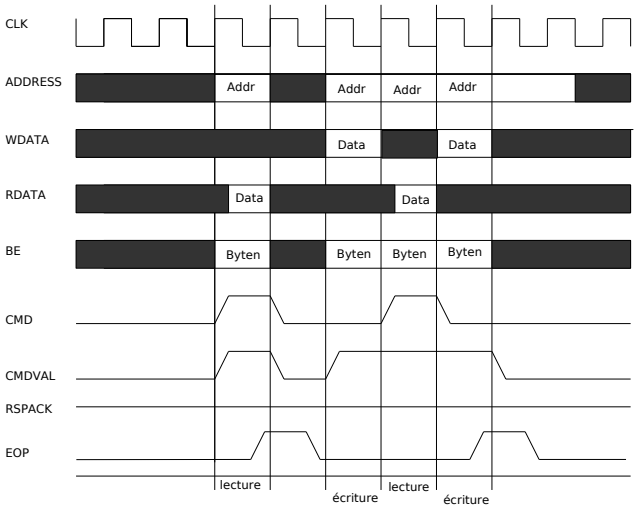


FIG. 3.2 – *Exemple de transactions en utilisant le protocole VCI*

3.3.2 Modèle du processeur MIPS R3000

Alors que les anciennes générations de systèmes embarqués étaient réalisées uniquement à base de circuits dédiés ASIC, donc sans processeurs, le nombre de processeurs dans les systèmes embarqués actuels augmente de plus en plus. Cette solution est attractive à plus d'un titre. En effet, l'utilisation d'unités programmables (les processeurs) permet de réduire le temps de conception, simplifie les tests du circuit et offre enfin un coût de fabrication intéressant. Ainsi, la part et le coût du logiciel dans les systèmes embarqués en général et dans les systèmes embarqués hautes performances en particulier, ne cesse d'augmenter. Néanmoins, l'utilisation des processeurs dans les systèmes embarqués pose le problème des respects des contraintes de temps et de consommation d'énergie. La solution que nous proposons dans le cadre de cette thèse consiste à utiliser une architecture multiprocesseur pilotée par une fréquence d'horloge relativement réduite d'un côté et d'adapter l'architecture du ou des processeurs aux spécificités de l'application de l'autre côté. Cette opération d'adaptation de l'architecture consiste en réalité à trouver la configuration (déterminer les paramètres) matérielle permettant d'obtenir un maximum de performances avec le minimum de consommation d'énergie.

Comme nous le verrons dans les chapitres suivants, notre solution n'est efficace que si nous disposons d'outils permettant d'évaluer les performances et la consommation des différentes configurations dans un temps réduit. Chaque configuration se caractérise par rap-

Type	Scalaire
Ordonnancement des instructions	Dans l'ordre
Étages de pipeline	5
Gestion du pipeline	solution matérielle
Prédiction du branchement	N'est pas implémentée
Latence d'exécution d'une instruction	1 cycle
Plage de Fréquence	100-200 MHz

TAB. 3.2 – *Caractéristiques du MIPS R3000*

port aux autres configurations par un certain nombre de paramètres comme : le nombre de processeurs, le nombre d'UAL (Unité Arithmétique et Logique) dans chaque processeur, la taille des caches, l'intégration ou non d'unités spécialisées (co-processeurs, FPGA), le type de NoC et les débits sur ce dernier, etc. La figure 3.3, donne un aperçu sur les processeurs les plus couramment utilisés dans la réalisation des systèmes sur puce. Comme nous pouvons le voir sur cette figure, depuis plusieurs années les processeurs ARM occupent une place importante.

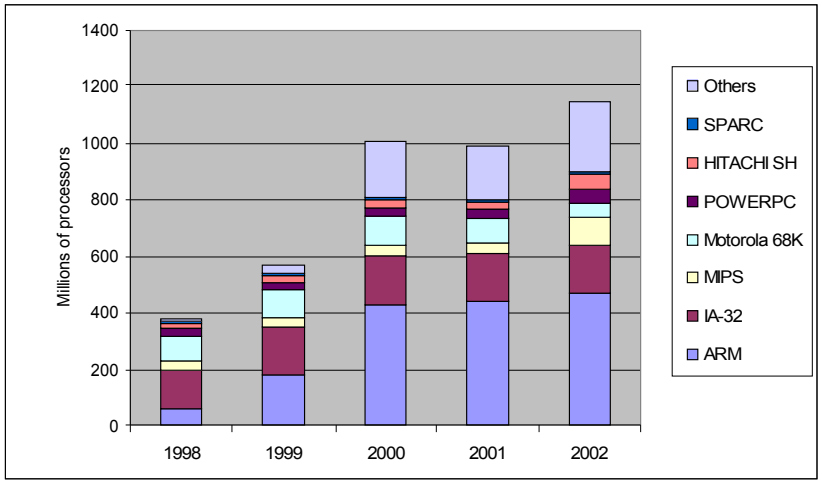


FIG. 3.3 – *Parts des fabricants dans le marché des processeurs embarqués [110]*

Le modèle de processeur qui a été utilisé est le MIPS R3000. Ce choix est justifié par l'existence d'un composant MIPS R3000 dans la bibliothèque SoCLib au moment du démarrage de la thèse. D'autres processeurs sont en cours de développement pour une future intégration à SoCLib. Le MIPS R3000 possède une architecture RISC avec les caractéristiques présentées dans le tableau 3.2. Dans cette étude, nous considérons que les mémoires caches d'instructions et de données sont des modules séparés du processeur qui seront présentées dans une section à part.

Le MIPS R3000 possède une architecture pipeline à 5 étages à savoir *Fetch Instruction* "I", *Decode Instruction* "D", *Execute Instruction* "E", *Memory Acess* "M" et *Write Back* "W". La figure 3.4 représente le chemin de données du processeur lors de l'exécution d'une opération d'addition. Une nouvelle instruction est chargée à chaque cycle dans l'étage "I", pour être décodée dans l'étage "D". Le résultat de l'instruction est par la suite calculé dans l'étage "E".

Tout accès à la mémoire de données est réalisé dans l'étage "M" et, finalement, l'enregistrement du résultat dans le registre destination se fait à l'étage "W".

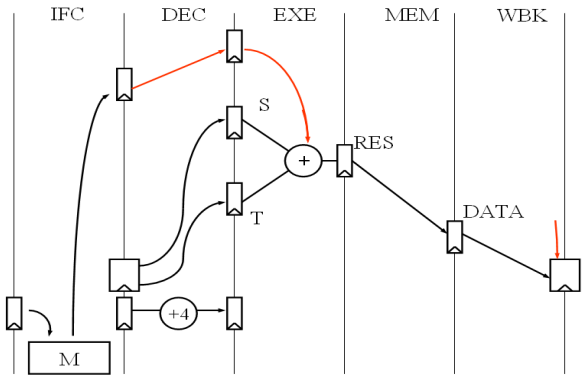


FIG. 3.4 – *Chemin de données pour une opération d'addition*

Le MIPS R3000 est connecté aux autres unités à travers 3 ports (figure 3.5) :

1. Un port contenant des Lignes d'interruption, utilisées pour le traitement des événements asynchrones. Ici, l'interruption correspond à une requête matérielle pour un traitement spécifique par le processeur (E/S, ordonnancement des tâches, etc.).
2. Un port avec le cache d'instructions qui est composé principalement des signaux d'adresses et de données et du signal *Miss* pour informer le processeur de l'occurrence d'un défaut de cache.
3. Un port avec le cache de données qui est composé principalement des signaux d'adresses et de données, du signal *Miss* pour informer le processeur d'un défaut de cache, du signal *Type* pour définir le type d'opération (lecture ou écriture) et enfin du signal *UNC* pour indiquer au cache que la donnée ne peut pas être chargée (non-cachable).

En plus de ces 3 ports, 2 signaux : "reset" pour la remise à zéro du processeur et le signal d'horloge sont prévus. L'interface du processeur MIPS R3000, reliée aux caches d'instructions et de données, est optimisée afin de pouvoir charger une donnée et une instruction par cycle. Lors d'un défaut de cache (*cache miss*), le processeur rentre dans le mode inactif (*idle*) jusqu'à ce qu'il reçoit l'instruction ou la donnée manquante. Pendant ce temps d'inactivité du processeur, la consommation d'énergie est réduite puisqu'aucune activité n'est réalisée.

Cette architecture du processeur MIPS R3000 est implémentée en utilisant un composant SystemC. Le comportement du composant est décrit avec une FSM qui implémente les 3 fonctions *Transition, Moore et Mealy*, déclarées en tant que processus SC_METHOD. En effet, ces 3 fonctions doivent d'être exécutées entièrement à chaque cycle d'horloge afin de calculer en premier lieu le nouvel état du composant et en deuxième lieu l'état des signaux de sorties. Pour cette raison la fonction de transition est sensible au front montant de l'horloge alors que les fonctions de génération Moore et Mealy sont sensibles au front descendant de l'horloge. En réalité cette démarche est valable pour tous les composants décrits au niveau CABA. Pour éviter la redondance dans les descriptions des autres composants de l'architecture, nous insistons sur un aspect particulier du composant. Le but final étant, s'il faut le rappeler, de quantifier l'effort nécessaire pour obtenir une description au cycle près pour ces composants.

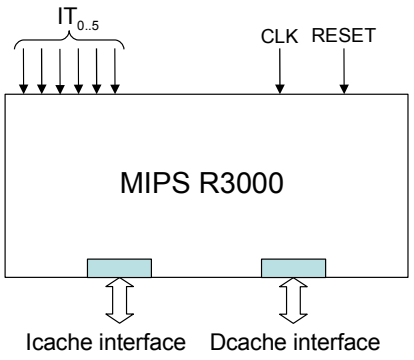


FIG. 3.5 – *Interface du MIPS R3000*

Dans chaque composant décrit au niveau CABA, nous avons besoin de déclarer au début du module composant :

1. Les noms des ports pour la communication avec les autres modules (étape 0)
2. Les noms des signaux pour modéliser des registres (étape 1)
3. Les noms des variables pour modéliser des constantes architecturales (étape 2)
4. Les fonctions décrivant le comportement (étape 3)

Le texte ci-après montre un exemple de déclaration du composant MIPS R3000 décrit au niveau CABA.

```
struct SOCLIB_MULTI_MIPS:sc_module {
    // Etape 0
    sc_in<bool>  CLK;
    sc_in<bool>  RESETN;
    // Interface avec cache d'instructions
    ICACHE_PROCESSOR_PORTS  ICACHE;
    // Interface avec cache de données
    DCACHE_PROCESSOR_PORTS  DCACHE;
    // Signaux d'interruption
    sc_in<bool>    IT_5;
    sc_in<bool>    IT_4;
    // Déclaration des registres (étape 1)
    sc_signal<int> GPR[32];
    sc_signal<int> PC;
    sc_signal<int> HI;
    sc_signal<int> LO;
    sc_signal<int> IR;
    // Constantes architecturales (étape 2)
    int  MSB_NUMBER; // nombre de bits MSB dans l'adresse
    ...
    SOCLIB_MULTI_MIPS(sc_module_name insname)
    {
        // Déclaration des fonctions pour décrire le comportement (étape 3)
```

```
        SC_METHOD (transition);
        sensitive_pos << CLK;
        SC_METHOD (genMealy);
        sensitive_neg << CLK;
        SC_METHOD (genMoore);
        sensitive_neg << CLK;
    }
}
```

3.3.3 Modèle du composant xcache

A cause de l'écart, de plus en plus important, entre les vitesses de fonctionnement des mémoires et celui des processeurs, les opérations d'accès mémoires sont devenues la cause principale de la chute des performances. Ce problème devient encore plus gênant dans le contexte des systèmes MPSoC où généralement il est nécessaire de transiter par un réseau d'interconnexion avant d'accéder à la mémoire. Cela entraîne une autre augmentation du temps d'accès. L'utilisation des systèmes à plusieurs niveaux de mémoires (ou hiérarchie mémoire) apparaît comme une solution intéressante. Cette hiérarchie comporte plusieurs niveaux. Au niveau le plus bas de la hiérarchie nous retrouvons les mémoires partagées, de taille significative mais avec des temps d'accès relativement importants. A l'opposé au niveau le plus haut, nous avons les registres en petit nombre et un temps d'accès très court. Les mémoires caches représentent un compromis entre ces deux niveaux extrêmes. Elles sont utilisées pour mémoriser les données et les instructions récemment accédées. Ce type de mémoire se caractérise par sa rapidité d'accès, de l'ordre d'un cycle processeur et par une capacité limitée de quelques dizaines de kilo-octets (Ko) [87].

Le composant cache (nommé xcache) de SoCLib est un composant générique construit sur une base de mémoire de type SRAM. Ce type de RAM présente l'avantage d'un temps d'accès réduit. Xcache contient en réalité deux mémoires caches : un cache pour les instructions et un cache pour les données. Ces deux derniers sont indépendants, mais partagent la même interface VCI (figure 3.6) pour l'accès au réseau d'interconnexion. Chaque cache est contrôlé par une FSM indépendante. Les champs VCI adresses et données sont codés sur 32 bits chacun. La taille de chaque cache est déterminée en fixant le nombre de lignes et le nombre de mots par ligne. Les deux caches instructions et données ont un degré d'associativité égale à 1 (*direct mapped cache*). La stratégie d'écriture dans le cache de données est l'écriture simultanée (*write through*).

La plateforme SoCLib utilise une solution logicielle simple pour résoudre le problème de cohérence des caches. En effet, à partir d'indications fournies par le programmeur, les données sont stockées dans deux segments (ou deux espaces d'adressage) différents de la mémoire. Ainsi, nous distinguons deux types ; des données locales ou privées à un processeur et les données partagées entre les processeurs. Les données locales peuvent être chargées dans le cache. Les données partagées entre les processeurs peuvent aussi être chargées dans les caches à condition que ces dernières soient lues uniquement. Dans le cas contraire, les données ne sont jamais mises dans le cache. Comme, nous pouvons l'imaginer cette solution simple ne permet pas d'optimiser l'utilisation des caches. En effet, dans les applications à traitement de données intensif, il est possible d'avoir une quantité importante de données partagées et modifiées par un ou plusieurs processeurs. Par conséquent, l'utilisation des caches est réduite et un trafic important sur le réseau d'interconnexion est généré. Il existe

dans la littérature plusieurs solutions pour résoudre ce problème [111]. En collaboration avec l'école nationale des ingénieurs de Sfax (ENIS), nous avons commencé le développement d'une solution matérielle plus performante en termes de performance et de consommation d'énergie [9]. Dans cette thèse, nous utiliserons uniquement la solution logicielle sans gestion de la cohérence.

En cas de défaut de cache (*Miss*), une mémoire tampon (*buffer*) permet de recevoir la ligne manquante. Ce tampon est aussi utilisé pour recevoir la donnée en cas de lecture non cachée. Enfin, le xcache possède une FIFO de requêtes (*request FIFO*) pour envoyer les demandes de lecture ou d'écriture vers les autres composants (mémoire partagée ou DMAC) via le réseau d'interconnexion. Ainsi, le contrôleur de l'interface VCI lit dans la FIFO et construit un paquet de commande lorsqu'il trouve plusieurs adresses appartenant à la même page. Par défaut, une page a une taille de 4 Ko. Notons que le processeur reste dans le mode inactif (*idle*) en attente de la donnée dans les cas de défaut de cache, lecture non cachée, écriture ou lorsque la FIFO est pleine.

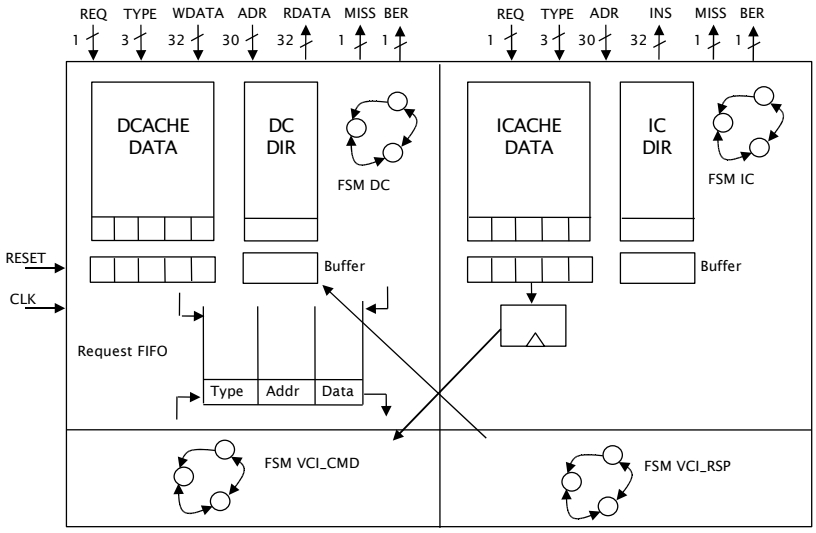


FIG. 3.6 – *Le composant xcache de SoCLib*

Pour implémenter le composant xcache au niveau CABA nous avons suivi la même démarche que celle appliquée au processeur. Ici, nous insistons sur l'effort de développement pour décrire le comportement des FSM de la xcache. Pour ce composant, nous avons besoin de 4 FSM à savoir :

1. FSM_DC pour modéliser le cache de données
2. FSM_IC pour modéliser le cache d'instructions
3. FSM_VCI_CMD pour modéliser l'interface de commande VCI
4. FSM_VCI_RSP pour modéliser l'interface de réponse VCI

Chaque FSM peut avoir plusieurs états. Là aussi, nous avons les 3 fonctions *Transition, Moore et Mealy*. La figure 3.7 montre l'exemple de la FSM qui décrit le cache de données (FSM_DC). Plusieurs états sont nécessaires pour assurer le bon fonctionnement du composant. Par exemple, l'état MISS_REQ correspond à une demande de donnée suite à un défaut de cache, l'état W_UPDT correspond au chargement d'une donnée dans le cache, etc.

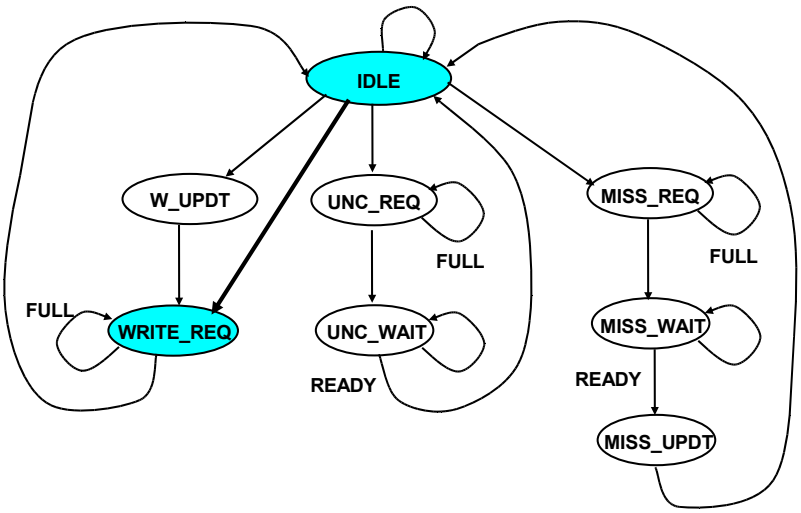


FIG. 3.7 – *FSM du cache de données*

Le texte ci-dessous montre un exemple de déclaration d'une FSM au niveau CABA.

```
// DCACHE_FSM STATES
switch (DCACHE_FSM) {

case DCACHE_INIT :
        DCACHE_TAG[DCACHE_CPT_INIT] = 0;
        DCACHE_CPT_INIT = DCACHE_CPT_INIT - 1;
        if (DCACHE_CPT_INIT == 0) { DCACHE_FSM = DCACHE_IDLE;}
break;

case DCACHE_IDLE :
if ((dcache_validreq == true) && (dcache_write == false) && (dcache_inval == false) &&
    (dcache_unc == false) && (dcache_hit == false))  { DCACHE_FSM = DCACHE_MISSREQ; }
else if ((dcache_validreq == true) && (dcache_write == true) &&
         (dcache_hit == false))  { DCACHE_FSM = DCACHE_WRITEREQ; }
else if ((dcache_validreq == true) && (dcache_write == true) &&
 (dcache_hit == true))  { DCACHE_FSM = DCACHE_WUPDT; }
else if ((dcache_validreq == true) && (dcache_inval == true) &&
 (dcache_hit == 1))  { DCACHE_FSM = DCACHE_INVAL; }
    ...
break;

case DCACHE_WUPDT :
    ...
    break;

case DCACHE_WRITEREQ :
    ...
```

```
break;

case DCACHE_MISSREQ :
    ...
break;

case DCACHE_MISSWAIT :
    ...
break;

    ...

} // end switch DCACHE_FSM
```

3.3.4 Modèle du réseau d'interconnexion

Le réseau d'interconnexion utilisé pour connecter les différents composants d'un MPSoC [24] joue un rôle d'une grande importance pour déterminer les performances du système. En effet, si les architectures MPSoC se présentent comme une alternative intéressante par rapport aux ASIC pour les prochaines générations de systèmes sur puce, il devient primordial de trouver un moyen de communication efficace entre les processeurs et les mémoires.

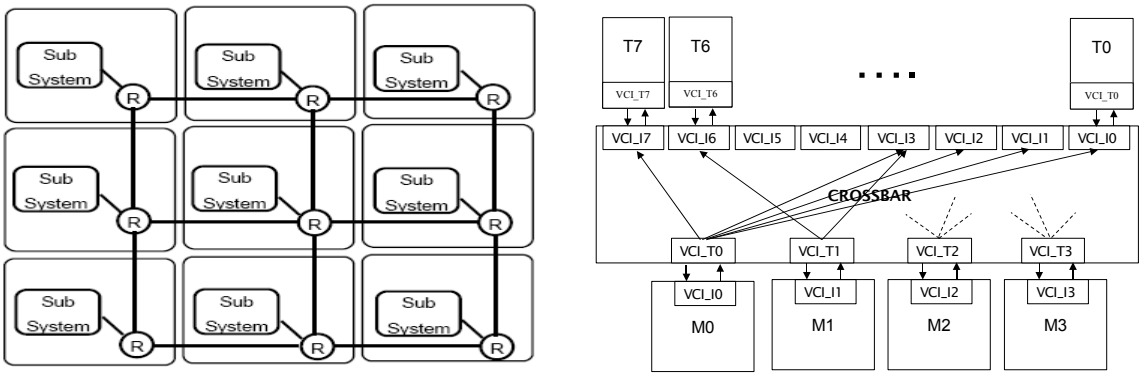
Différentes topologies de réseaux d'interconnexion existent. L'architecture en bus partagé est sans aucun doute la plus populaire. Néanmoins, ces dernières années des architectures de réseaux plus performantes telles que les grilles, les bus hiérarchiques, le crossbar et réseau multi-étages ont été proposées [24]. Dans le cadre de cette thèse, l'objectif n'est pas de proposer de nouvelles architectures de réseaux d'interconnexion, mais, là aussi, de concevoir des outils permettant d'accélérer la simulation pour explorer l'espace des différentes alternatives. Pour le réseau d'interconnexion, en plus des critères : délais de transmission, qui doivent être courts, et consommation d'énergie qui doit être faible, d'autres critères peuvent être déterminants dans le choix du réseau tels que :

- L'extensibilité du réseau pour pouvoir facilement intégrer un plus grand nombre d'éléments sur le réseau.
- La redondance des chemins entre 2 composants, afin de tolérer des pannes et, surtout, pour éviter les goulots d'étranglement dans le réseau.
- La reconfigurabilité dynamique du réseau afin de s'adapter aux besoins en communication de l'application.

La plateforme SoCLib que nous avons utilisée possède deux types de réseaux d'interconnexion, le crossbar et le DSPIN (*Distributed Scalable Integrated Network*) [106]. Le crossbar permet une liaison directe entre les initiateurs (M) et les cibles (T) (figure 3.8(b)).

Ce support de communication possède une large bande passante (*O(max(M,T))*) mais a comme inconvénient majeur la complexité de la conception en termes de points de connexion (*O(MxT)*). Cette complexité s'accentue avec l'augmentation du nombre de modules connectés. Pour cette raison, il est préconisé pour un nombre limité de composants interconnectés.

Le réseau DSPIN a une topologie en grille à deux dimensions (figure 3.8(a)) permettant de relier un nombre plus important de composants ou de sous-systèmes avec un coût rela-



(a) Architecture d'un système à mémoire distribuée (b) Architecture d'un système à mémoire partagée

FIG. 3.8 – différents types d'architectures MPSoC

tivement faible par rapport au crossbar. Le DSPIN permet ainsi une bande passante équivalente à celle du crossbar avec une complexité de conception réduite (*O(log(M+T))*). Néanmoins, ce dernier à un diamètre ($O(\sqrt{(M+T)})$), qui correspond au nombre maximal de routeurs à traverser, plus grand que celui du crossbar (*1*). Même si cela ne remet pas en cause la méthodologie proposée, dans cette thèse, nous nous intéressons aux systèmes multiprocesseurs à mémoires partagées. Nous nous sommes limités au crossbar qui est adapté pour les systèmes MPSoC étudiés. En effet, le nombre de composants que nous avons modélisé dans notre plateforme ne dépasse guère la dizaine de composants.

En plus du composant crossbar de SoCLib, nous avons développé le composant bus standard afin de réaliser des études comparatives. Ces deux composants sont génériques et par conséquent différents paramètres, tels que les latences de connexion entre les ports, peuvent être modifiés. Le protocole de communication utilisé par le crossbar est VCI pour la connexion avec les autres composants. Afin d'assurer un fonctionnement correct, lorsque le composant connecté est de type *initiateur*, c'est un port de type *target* dans le crossbar qui est utilisé et vice versa. Dans la figure 3.8(b), les ports target (resp. initiateur) du crossbar sont notés VCI_T_i (resp. VCI_I_i). Le fonctionnement de notre système MPSoC est totalement synchrone et de ce fait tous les composants sont cadencés par un seul signal d'horloge.

Du point de vue du fonctionnement du crossbar, à chaque cycle ce dernier réalise un test sur toutes les entrées du côté des initiateurs afin de récupérer les nouvelles requêtes de commande. Pour permettre à chaque initiateur de transmettre plusieurs requêtes successives, nous avons associé à chaque port d'entrée du réseau crossbar une FIFO pour stocker les demandes. La taille de cette FIFO est un paramètre à fixer par le concepteur. Pour résoudre le problème d'accès simultané à une même cible, le crossbar utilise la stratégie d'arbitrage par tourniquet (ou Round-Robin). Cette fonctionnalité est implémentée dans un composant SystemC au niveau CABA.

Pour montrer la complexité de la modélisation au niveau CABA de ce crossbar, c'est la génération des signaux de sorties des ports VCI à chaque cycle d'horloge qui est détaillée ci-après. L'état de sortie de chaque signal du port VCI est calculé dans la fonction Moore de la FSM qui décrit le composant. Le texte ci-après montre la boucle appliquée sur chaque port de type initiateur du réseau d'interconnexion afin de déterminer l'état des signaux de sortie. Dans cet exemple, nous nous intéressons aux paquets "réponses" qu'il faut transmettre des

ports VCI_I_i aux ports VCI_T_i.

```
void SOCLIB_VCI_GMN::genMoore()
{
// Boucle sur les ports de type initiateur
for (i=0 ; i<NB_INITIAT ; i++) {
 k     = T_ALLOC_VALUE[i]; //détermination du numéro de la cible
     if (RSP_FIFO_STATE[i][k] == not_empty) { //vérifier que le port n'est pas vide
        ptr   = RSP_FIFO_PTR[i][k]; //détermination du numéro d'ordre
        cmd   = RSP_FIFO_CMD[i][k][ptr]; //génération du mot de commande
        if (T_ALLOC_STATE[i] == true) { //vérifier que le port est prioritaire
            T_VCI[i].RSPVAL = true;     //réponse valide
            T_VCI[i].REOP = (bool) (cmd>>3  & 0x00000001);
            T_VCI[i].RTRDID = (vci_id_type) (cmd>>4  & 0x000000FF);
            T_VCI[i].RPKTID = (vci_id_type) (cmd>>12 & 0x000000FF);
            //lecture du numéro de l'initiateur
            T_VCI[i].RSCRID = (vci_id_type) (cmd>>20 & 0x000000FF);
            \\lecture de la donnée à envoyer
            T_VCI[i].RDATA = (vci_data_type) RSP_FIFO_DATA[i][k][ptr];
            ....
        }
     }
     else{
       //si port vide, alors bus au repos
       .....
     }
} // end loop initiators
```

3.3.5 Modèle de la mémoire de données et d'instructions

Les besoins en ressources mémoires des applications de traitement de données intensif deviennent de plus en plus importants. Ces dernières peuvent réduire les performances et/ou augmenter la quantité d'énergie électrique totale consommée. En effet, plus l'application est complexe, plus grands seront ces besoins en mémoire pour stocker les instructions et les données. Selon les prévisions de l'ITRS (International Technology Roadmap for Semiconductors) (figure 3.9), dans les prochaines générations de systèmes embarqués les unités de mémorisation (caches, scratchpads, buffers et autres) occuperont plus de 80% de la surface de la puce, alors que la surface de la logique réutilisable (les processeurs) et la surface des unités spécifiques (comme les ASIC) ne dépasseront pas les 5% chacun.

L'intégration des mémoires dans les systèmes embarqués peut se faire en utilisant différentes technologies : SRAM, DRAM, FRAM, EPROM, FLASH, etc. Pour chaque technologie plusieurs architectures et optimisations sont possibles dans la perspective d'augmenter les performances et réduire la consommation d'énergie [107]. Même si la part des mémoires embarquées non volatiles (ou mortes) devient de plus en plus grande pour stocker les applications dans les systèmes embarqués (tel que les mémoires FLASH), dans cette thèse nous nous sommes limités aux mémoires RAM embarquées, et plus spécialement les mémoires

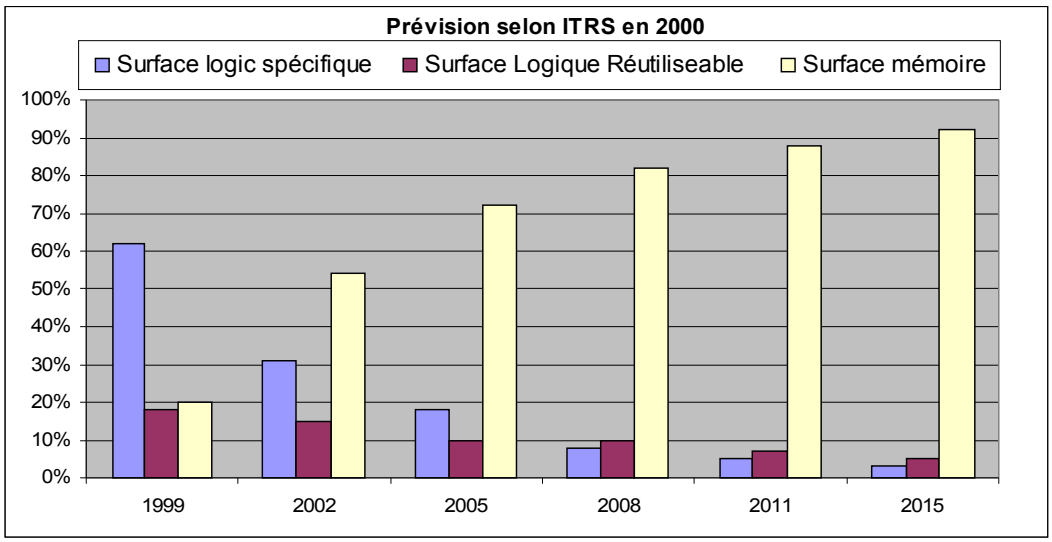


FIG. 3.9 – *Evolution des surfaces pour "logique spécifique", "logique réutilisable" et "mémoire" dans les SoC*

SRAM qui offrent une simplicité d'intégration et un temps d'accès plus court.

Dans un système MPSoC, il existe deux types d'architecture mémoire :

1. Mémoire globale partagée accessible par tous les processeurs. Dans ce cas les données sont entièrement stockées dans cette mémoire. Ce type d'architecture est utilisé dans les applications où les données sont fortement dépendantes [87].
2. Mémoire distribuée. Dans ce cas les données sont découpées et réparties dans les mémoires de chaque processeur. La mémoire associée à un processeur peut être accessible aux autres processeurs (mémoire distribuée partagée).

Le choix entre ces deux types d'architectures pour implémenter une application donnée n'est pas simple. En effet, il est contraint en premier lieu par la nature de l'application et en deuxième lieu par le type des composants utilisés (processeur, réseau de connexion, la bande passante des bus). Dans cette thèse, nous nous limitons aux architectures MPSoC à mémoire partagée.

Au niveau CABA, les différentes tâches à exécuter sont présentées sous forme d'un code binaire exécutable. On associe à chaque segment d'instructions et de données une adresse de base et une taille en octets définies par le concepteur du système (figure 3.10). Avant de commencer la simulation, les différents segments sont stockés dans des mémoires partagées d'instructions et de données comme le montre la figure 3.10. Les mémoires d'instructions et de données utilisées dans SoCLib sont de type SRAM (*Static Random Access Memory*) mono-port. Ce type de mémoire exige certes plus d'espace (4 fois plus grand) sur la puce par rapport à la technologie DRAM (*Dynamic Random Access Memory*) mais présente l'avantage de rapidité (cycles plus courts).

Notons que cette architecture du module mémoire peut être modifiée pour permettre des accès multiples à un même module et améliorer ainsi les performances. Le nombre de modules mémoire, la taille de chacun des modules et la latence des accès, peuvent être modifiés. Ces paramètres déterminent un espace de solutions pouvant être exploré. Le concepteur peut aussi modifier le placement des différents segments de données pour alléger les goulots d'étranglement dans le réseau d'interconnexion et améliorer les performances du système.

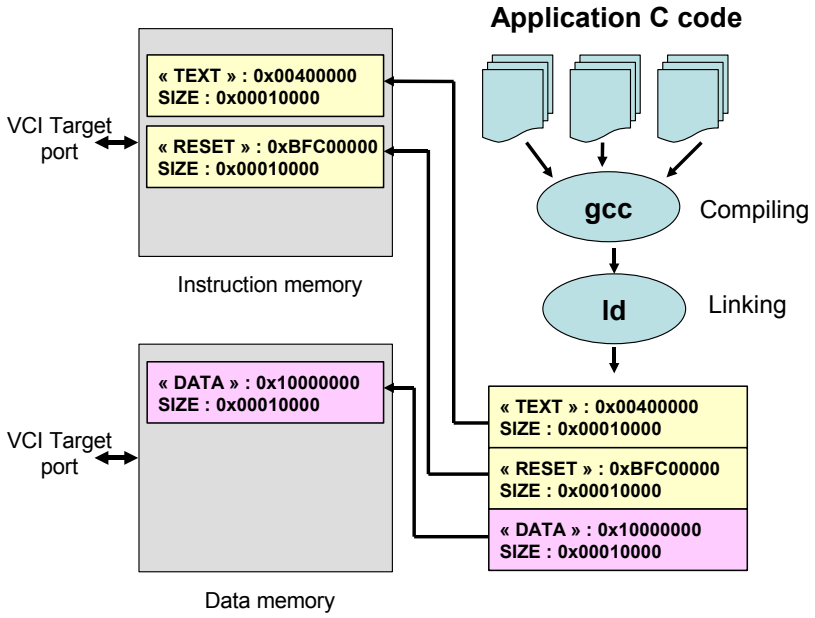


FIG. 3.10 – *Mémoires d'instructions et de données*

Un composant SystemC intégrant les fonctionnalités de la mémoire (de données ou d'instructions) a été conçu. Le texte ci-dessous montre l'implémentation de la fonction "Transition" pour calculer le nouvel état de la mémoire à chaque cycle d'horloge. A partir de l'état précèdent du composant (lecture, écriture ou inactif) et des valeurs des signaux d'entrées (comme CMDVAL de VCI), un nouvel état de la FSM est déterminé. Dans un souci de simplicité, nous ne détaillons que les transitions à partir de l'état "repos".

```
void transition()
{
case 0: //Etat inactif
if (VCI.CMDVAL == true) { // si requête valide
        // Identifier le type d'opération et l'adresse concernée
adrword = ((int)VCI.ADDRESS.read()) & 0xFFFFFFFC;
wdata   = (int)VCI.WDATA.read(); // données à écrire
be      = (int)VCI.BE.read(); //byte enable
cmd     = (int)VCI.CMD.read(); //opération: lecture ou écriture
rdata = 0;
rerror  = 1;
for (int i = 0 ; ((i < NB_SEG) && (rerror != 0)) ; i++) {
            // Identifier le segment de données
if ((adrword >= BASE[i]) && (adrword <  BASE[i] + SIZE[i])) {
                //Dans quel segment est la donnée?
                // fonction de lecture et écriture de données dans un segment
rdata  = rw_seg(RAM[i], (adrword - BASE[i]) >> 2, wdata, be, cmd);
rerror = 0;
} // end if
} // end for
```

```
        //préparation du paquet réponse:
FIFO_RDATA[0]  = rdata;
FIFO_RERROR[0] = rerror;
FIFO_RSCRID[0] = (int)VCI.SCRID.read();
FIFO_RTRDID[0] = (int)VCI.TRDID.read();
FIFO_RSCRID[0] = (int)VCI.SCRID.read();
FIFO_REOP[0] = (int)VCI.EOP.read();
FIFO_STATE = 1; // on va à l'état 1
} else {
FIFO_STATE = 0; // si non on reste à l'état repos
}
break;
case 1:
...
case 2:
...
break;
}
```

3.3.6 Modèle du contrôleur DMA

Dans les applications de traitement de données, une quantité importante de données est transférée entre les modules mémoires et les périphériques. Pour réduire les délais de ces transferts, nous avons conçu un contrôleur DMA (*Direct Memory Access controler*). Ce type de composant permet d'accélérer le transfert de données, ce qui libère les processeurs de ces tâches et évite le passage par des mémoires intermédiaires.

La figure 3.11 illustre l'architecture du composant développé. Il possède 3 ports VCI, le premier est de type *cible* (Slave_VCI) pour permettre aux processeurs de configurer le DMAC via les registres du DMA. Les deux autres ports sont de type *initiateur* (Read_VCI et Write_VCI) permettent un transfert de données simultané entre une adresse source et une adresse destination. Notre contrôleur DMA contient 5 registres. Les adresses source et destination sont spécifiées respectivement dans les registres *Source addr* et *Destination addr*. Le registre *Transfer size* détermine la taille du bloc à transférer. L'écriture dans ce registre fait démarrer l'opération de transfert de données. Le contrôleur DMA active son signal *IRQ* et initialise le registre *State* à 1 pour signaler la fin du transfert avec succès. L'écriture dans le registre *Reset* permet de désactiver le signal *IRQ* et initialise les autres registres pour un nouveau transfert. La fonctionnalité du contrôleur DMA est décrite avec un composant SystemC possédant 4 FSM (figure 3.11).

3.3.7 Modèle de l'accélérateur matériel TCD-2D

Aujourd'hui avec la complexité des applications embarquées, l'utilisation d'une approche totalement logicielle pour implémenter certains algorithmes exige un nombre de processeurs dépassant les capacités des circuits en termes de surface et de consommation. Pour résoudre ce problème, les tâches soumises à de fortes contraintes temporelles peuvent être implémentées de façon matérielle. Ceci permet d'obtenir un processeur dédié pour une fonctionnalité donnée, ce que nous appelons ici un "accélérateur matériel". Nous obtenons

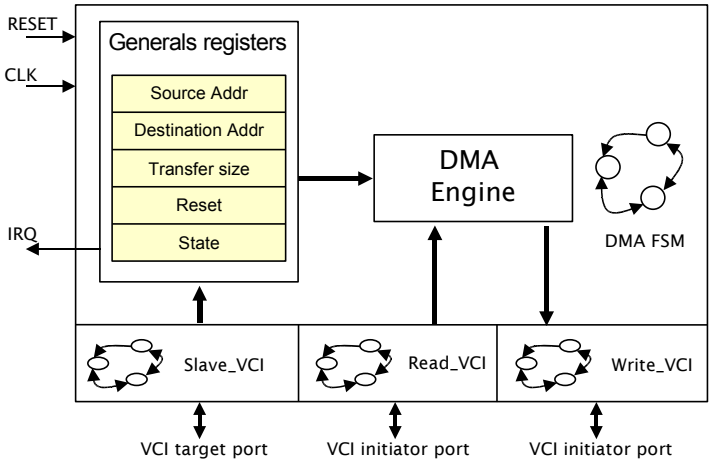


FIG. 3.11 – *Architecture du contrôleur DMA*

ainsi une architecture hétérogène pour implémenter les différentes tâches d'une application de façon logicielle ou matérielle.

Dans cette thèse nous nous intéressons à l'implémentation matérielle de la Transformée en Cosinus Discrète (TCD). La TCD est un algorithme très utilisé dans le domaine des applications vidéo. Elle permet la transformation d'un bloc d'image du domaine spatial au domaine fréquentiel [78]. La TCD bidimensionnelle (TCD-2D) est appliquée sur des blocs de 8x8 pixels et implémentée à travers deux TCD unidimensionnelles (TCD-1D). La première TCD-1D opère sur les 8 lignes tandis que la deuxième opère sur les 8 colonnes. Pour résoudre la complexité de calcul de la TCD et satisfaire des critères de temps réel, plusieurs architectures ont été proposées [78, 68, 117]. La base de comparaison entre ces implémentations est le nombre de multiplications (MUL) et d'additions (ADD) utilisés. L'algorithme le plus optimisé est celui de Loeffler [80] qui utilise 11 MUL et 29 ADD. Nous avons implémenté cet algorithme sur la carte de développement pour FPGA STRATIX II EP2S60 d'Altera [13] en utilisant le langage VHDL. L'objectif est d'établir une analyse temporelle ainsi qu'une évaluation de la consommation de puissance de l'accélérateur matériel développé. Cette analyse nous permettra d'estimer les performances à un niveau plus haut.

	TCD
ALUTs	1569 (3%)
E/S	195 (40%)
Blocs RAMs	0%
Blocs DSPs	22 (8%)
Fmax (MHz)	120

TAB. 3.3 – *Résultats d'implémentation de la TCD*

Une simulation temporelle en utilisant l'outil Quartus II d'Altera [82] (tableau 3.3) montre que la fréquence de fonctionnement de la TCD-2D est de l'ordre de 120 MHz. La solution matérielle de la TCD-2D développée permet une accélération du traitement par un facteur de 200 par rapport à la solution logicielle exécutée avec le processeur MIPS R3000. En effet, l'utilisation de la solution matérielle permet de réaliser une TCD-2D pour un bloc

d'image (8x8 pixels) en 720 cycles au lieu des 15000 cycles sur le MIPS R3000. Pour pouvoir intégrer l'accélérateur matériel TCD-2D dans l'environnement de simulation CABA, nous avons réécrit le composant en utilisant le langage SystemC. Une interface VCI de type *cible* a été ajoutée à ce composant pour le connecter au réseau d'interconnexion. Du point de vue fonctionnement, le contrôleur DMA décrit précédemment se charge de transférer un bloc d'image (8x8 pixels) d'un composant d'entrée/sortie (*Sensor*) à l'accélérateur matériel TCD-2D. Après l'exécution de la tâche TCD-2D, le bloc traité sera transféré par le contrôleur DMA vers la mémoire de données. Cette méthodologie peut être appliquée à divers algorithmes qui peuvent être implémentés sous forme matérielle en vue d'accélérer leur exécution comme la FFT (*Fast Fourier Transform*).

3.4 Estimation de performance au niveau CABA

En utilisant les composants décrits dans la section précédente, différentes architectures multiprocesseurs peuvent être modélisées, simulées et évaluées. Ces dernières peuvent contenir un nombre variable de processeurs, d'accélérateurs matériels, de périphériques d'entrées/sorties, etc. La figure 3.12 donne un exemple d'architecture MPSoC à base de ces composants. Cet exemple d'architecture convient en général aux applications de traitement de signal intensif nécessitant des moyens de calcul parallèles importants. Pour exécuter l'application, les différentes tâches sont placées, ensuite compilées vers les processeurs cibles. Les données manipulées sont aussi placées sur les bancs de la mémoire partagée SRAM. Pour garantir un ordonnancement correct des tâches entre les processeurs, des variables de synchronisation sont utilisées. Ces variables sont stockées en mémoire. Elles sont lues et modifiées par les différents processeurs. Pour cette raison, ces variables de synchronisation (ou sémaphores) ne doivent pas être copiées dans les caches. L'estimation des performances de l'application est donnée par le simulateur d'architecture en nombre de cycles. Pratiquement, à chaque processeur est associé un registre compteur de temps dans le composant "*Timer*". Le compteur de temps est incrémenté à chaque cycle d'horloge. Ainsi, pour estimer le temps d'exécution d'une tâche sur un processeur, il suffit de lire la valeur du registre avant et après l'exécution de la tâche. Par ailleurs, le temps d'exécution de toute l'application est donné par l'horloge globale du simulateur SystemC.

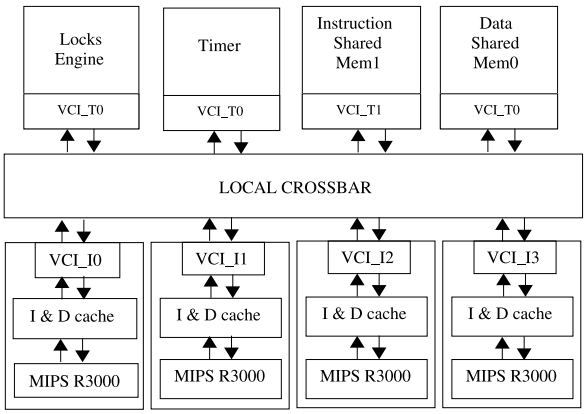


FIG. 3.12 – *Exemple d'architecture MPSoC*

Comme nous l'avons souligné, la description d'une architecture au niveau CABA permet d'obtenir des estimations de performances précises. Les résultats d'estimations obtenus à ce niveau permettront la conception d'outils en vue d'une exploration des alternatives architecturales. La figure 3.13 montre un exemple d'exploration d'architecture au niveau CABA. Dans cette figure, la taille du cache de données et d'instructions varie de 1Ko à 32Ko alors que le nombre de processeurs varie de 4 à 16. Les résultats sont rapportés pour l'application codeur H.263 qui sera détaillée dans le chapitre suivant. L'objectif ici est de monter qu'il est possible grâce à cet environnement de trouver la solution optimale en terme de performance. Ces résultats d'exploration seront discutés dans le chapitre 5. Malgré la précision de ces estimations, le problème majeur du niveau CABA est le temps de simulation nécessaire pour obtenir ces résultats. Le tableau 3.4 donne un ordre de grandeur sur les temps de simulation à ce niveau pour l'application codeur H.263 exécutée pour une séquence vidéo de type QCIF en utilisant une machine Pentium M (1.6 GHz). La taille des caches des processeurs est fixée à 4 Ko. L'évaluation du système nécessite ainsi plusieurs heures de simulation.

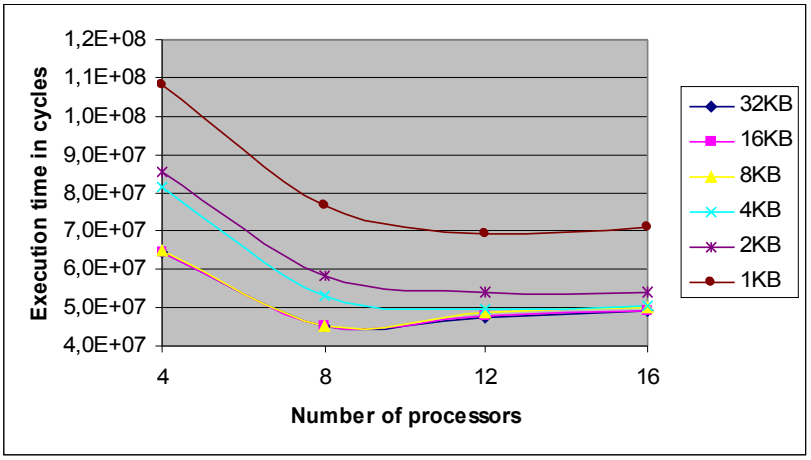


FIG. 3.13 – *Estimation de performance au niveau CABA pour l'application codeur H.263*

Nombre de processeurs	4	8	12	16
Temps de simulation	2h 37min	2h 57min	3h 52min	5h

TAB. 3.4 – *Temps de simulation en fonction du nombre de processeurs pour l'application H.263*

Des expérimentations sur d'autres simulateurs MPSoC au niveau CABA comme MPARM [19] ont donné des temps de simulation comparables. Cet inconvénient s'accentue avec l'augmentation du nombre de composants dans le système MPSoC à simuler et plus spécifiquement avec le nombre de processeurs. La figure 3.14 donne le temps de simulation, en (*Instructions simulées par sec*) pour l'application H.263 avec un nombre croissant de processeurs. Elle montre que la vitesse de simulation est inversement proportionnelle au nombre de processeurs. En effet, en augmentant le nombre de processeurs, la vitesse de simulation chute à cause du nombre croissant de changements de contexte de l'ordonnanceur SystemC d'une part et des opérations de communication entre les différents processus qui composent la plateforme d'autre part.

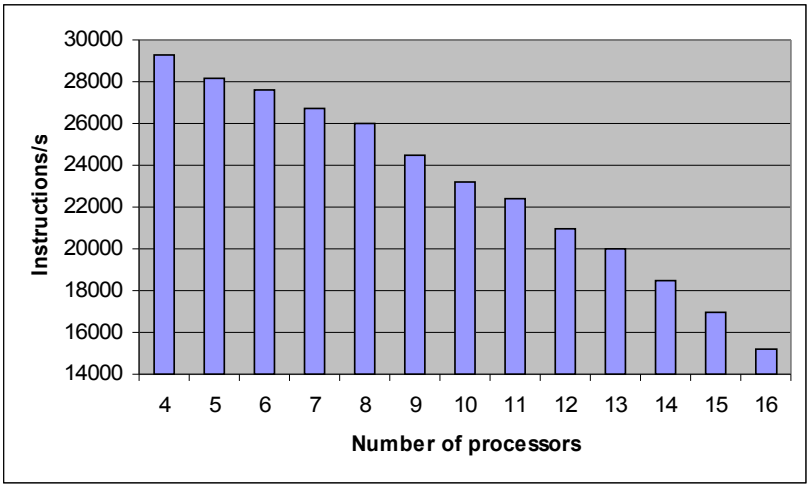


FIG. 3.14 – *Performance du simulateur SystemC*

3.5 Intégration dans Gaspard

Les composants matériels décrits au niveau CABA de diverses bibliothèques peuvent être intégrés dans notre environnement de conception des systèmes MPSoC Gaspard. Comme nous l'avons introduit au chapitre précédent, cet environnement vise la simulation du système à différents niveaux d'abstraction en particulier le niveau CABA. La simulation à ce niveau est obtenue à partir d'une modélisation de haut niveau et différentes transformations. La figure 3.15 détaille les phases de transformation appliquées sur la partie matérielle. Dans notre description nous allons insister sur le modèle d'architecture, le chapitre 6 fera le point sur les modèles d'application et d'association.

Notre flot de conception commence par une étape de modélisation de haut niveau du système MPSoC. Afin de profiter d'une description visuelle, nous utilisons le langage UML pour modéliser le système, il sera détaillé dans le chapitre 6. Cette modélisation décrit les composants de l'architecture (processeur, réseau d'interconnexion, etc.) sans se référer aux composants réels. En effet, à cette étape de conception, la description doit être assez générale, puisque plusieurs niveaux d'abstraction peuvent être ciblés. Le choix d'un composant particulier de l'architecture dépend de la disponibilité de ce composant au niveau d'abstraction ciblé. De ce fait, le concepteur a besoin d'une deuxième étape de déploiement qui associe à chaque composant du modèle de haut niveau un composant réel existant dans la bibliothèque. Ceci nous permet d'obtenir un modèle d'architecture déployé. Ce dernier peut intégrer plusieurs composants décrits au niveau CABA possédant des interfaces de communication différentes. Dans ce cas, il est nécessaire de réaliser une adaptation entre les interfaces. Pour ce faire, deux méthodes sont possibles. La première est d'utiliser des adaptateurs qui sont déjà développés et mis à disposition avec la bibliothèque de composant. La seconde approche est d'utiliser des outils de génération automatique d'adaptateurs à partir de la description de chaque interface de communication. SPIRIT [130] est un exemple de standard de description.

Cette phase de transformation nous permet d'obtenir un modèle d'architecture homogène au niveau CABA. A partir de ce modèle, nous appliquons une dernière transformation qui consiste à générer le code du système à simuler. Ce code comporte d'un côté l'instancia-

tion des composants à partir des bibliothèques avec les paramètres spécifiés par le concepteur et d'un autre côté la connexion entre les interfaces des composants. L'exécution du code généré permet la vérification fonctionnelle du système MPSoC et l'estimation des performances correspondantes. Le chapitre 6 présentera une méthodologie de conception dirigée par les modèles afin de réaliser l'ensemble de transformations depuis le modèle abstrait jusqu'à la génération de code.

3.6 Conclusion

Ce chapitre a montré que les méthodes classiques pour simuler et évaluer les performances des systèmes MPSoC au niveau CABA sont fastidieuses, en particulier pour des architectures embarquées complexes. Notre étude a été réalisée en utilisant la bibliothèque de composants SoCLib que nous avons enrichie pour rendre possible la simulation des systèmes MPSoC hétérogènes. A travers cette étude, nous avons montré que l'effort de développement nécessaire au niveau CABA est considérable pour obtenir des estimations précises. Par ailleurs, les temps de simulation obtenus à ce niveau ne sont pas suffisamment réduits pour permettre une exploration architecturale rapide des systèmes MPSoC.

Dans notre travail, nous nous sommes limités à des systèmes MPSoC à mémoires partagées et des architectures de processeurs simples tels que le processeur MIPS R3000. Le problème des temps de simulation s'accentue avec l'utilisation de processeurs plus complexes tels que les architectures superscalaires ou VLIW (*Very Large Instruction Word*) ou avec des architectures plus complexes telles que les architectures à mémoires distribuées.

Dans le chapitre suivant, nous allons présenter notre approche pour réduire le temps de simulation des systèmes MPSoC. Celle-ci permettra l'estimation des performances dans des délais relativement courts et pourra être intégrée dans un processus d'exploration de l'espace de solutions.

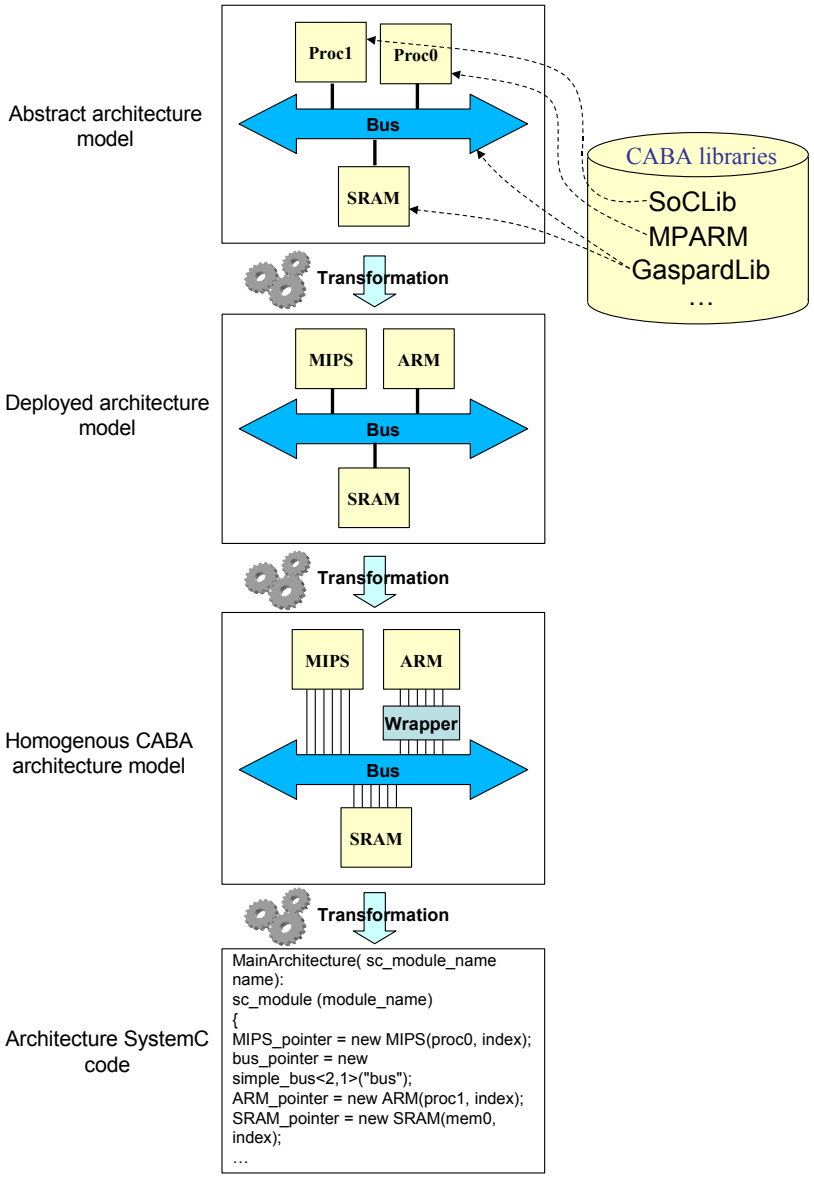


FIG. 3.15 – *La réutilisation des composants matériels depuis le modèle abstrait jusqu'à la génération de code*

Chapitre 4

Simulation et estimation de performance des MPSoC au niveau PVT

4.1 Introduction

Dans le chapitre précédent, la problématique du temps de simulation pour évaluer les performances des MPSoC au niveau CABA a été présentée. Nous avons ainsi montré que les temps de simulation obtenus à ce niveau ne sont pas satisfaisants pour une évaluation rapide de solutions architecturales. Cette lenteur de simulation est due principalement à la quantité importante de détails architecturaux que le simulateur au cycle précis doit prendre en compte à chaque intervalle de temps. Dans ce chapitre, nous proposons une solution qui consiste à décrire le système au niveau transactionnel (TLM), plus spécifiquement au niveau PVT. Y compris à ce niveau d'abstraction, la résolution de la complexité de conception des systèmes MPSoC nécessite une stratégie d'exploration multi-niveaux. A chaque niveau, un ensemble de solutions pourra être éliminé de l'espace de solutions architecturales. Pour cela, nous allons définir un environnement de simulation composé de 3 sous-niveaux. Ces derniers représentent en quelque sorte une décomposition du niveau PVT. Ainsi, dans notre plateforme le niveau PVT se décline en 3 sous-niveaux "pattern-accurate" noté PVT-PA, "transaction-accurate" noté PVT-TA, et enfin "event-accurate" noté PVT-EA. Ces 3 sous-niveaux sont présentés dans ce chapitre en partant du niveau d'abstraction le plus élevé (PA) vers le niveau d'abstraction le moins élevé (EA). A ce niveau, l'architecture est définie de façon beaucoup plus détaillée que dans PA. L'écart d'abstraction entre PA et EA nous a poussé à définir un niveau intermédiaire TA.

Dans ce chapitre notre évaluation est basée sur le critère d'estimation de performance (temps d'exécution de l'application) ce qui justifie notre développement d'un modèle de temps pour chaque sous-niveau. Pour implémenter une famille de MPSoC, nous avons modélisé plusieurs composants matériels tels que le processeur, la mémoire cache, le réseau d'interconnexion, accélérateur matériel, etc. Ces composants ont été décrits dans chacun des sous-niveaux. Les résultats expérimentaux montrent une accélération de simulation et une erreur relative à la précision de description de chaque sous-niveau.

Ce chapitre est structuré comme suit. La deuxième section présentera un justificatif de l'utilisation du niveau PVT pour la simulation et l'évaluation des systèmes MPSoC. La section 3 détaillera les trois sous-niveaux décrits au niveau PVT pour une exploration multi-niveaux de l'espace de solutions architecturales. La description d'un système MPSoC aux sous-niveaux PVT-PA, PVT-EA et PVT-TA sera détaillée respectivement dans les sections 4, 5 et 6. La section 7 présentera les modèles de différents composants matériels développés pour implémenter l'environnement PVT proposé. Des modèles d'estimation de temps d'exécution sont proposés dans la section 8 afin d'évaluer les performances dans les 3 sous-niveaux. L'efficacité de notre proposition sera illustrée à travers l'application codeur H.263 exécutée sur un exemple paramétré de MPSoC. La section 9 présentera les résultats de simulation qui comparent la description d'un système MPSoC décrit avec l'environnement PVT (avec ces 3 sous-niveaux) et le niveau CABA. Les critères de comparaison que nous avons choisis sont : le temps de simulation, la précision dans l'estimation des performances et l'effort à réaliser afin de développer les modèles dans chacun des 3 sous-niveaux de PVT. La section 10 présente une synthèse de ce chapitre.

4.2 Proposition et justification

Pour réduire le temps de simulation des systèmes embarqués et en particulier ceux intégrant plusieurs processeurs, il faut identifier les détails de la micro-architecture du niveau CABA qui peuvent être omis pour alléger la simulation. Parmi ces détails, nous avons identifié ceux qui sont reliés à la partie calcul (ou traitement) et ceux liés à la partie communication. Concernant la partie calcul, les machines d'états finis (Transition, Mealy et Moore) décrivant le composant représentent la source prépondérante dans les temps de simulation au niveau CABA. Pour la partie communication, nous avons remarqué que les machines d'états finis qui décrivent l'interface de chaque composant ralentissent aussi la simulation. Notre solution tend vers un compromis équitable entre précision et temps de simulation. Nous proposons à un niveau d'abstraction plus élevé la prise en compte d'indicateurs déduits d'une simulation au niveau CABA. Cette abstraction doit être suffisante pour pouvoir vérifier le comportement de notre système (application déployée sur l'architecture) et permettre aussi l'estimation du temps d'exécution d'une façon assez précise.

Notre étude de cas est construite autour d'une famille MPSoC sur laquelle une application sera déployée. A partir de ce déploiement sur cette architecture, nous évaluons les performances de notre système ce qui nous permet d'extraire la solution la plus adéquate. A chaque niveau d'abstraction correspond une précision d'évaluation de performance plus au moins coûteuse en temps de simulation. Ainsi, le niveau PVT permet d'extraire les solutions pertinentes dans un temps raisonnable tout en offrant un niveau de précision relativement intéressant. Dans nos trois sous-niveaux PVT, certains détails d'implémentation visibles au niveau CABA disparaissent au profit de fonctions d'évaluation basées sur des résultats de simulation au niveau CABA.

Dans ce chapitre, nous nous limitons au critère de performance (temps d'exécution) pour comparer les différentes solutions architecturales. Au chapitre suivant, la consommation de puissance sera introduite comme un second critère.

4.3 Description des MPSoC au niveau PVT

Pour réduire le temps de simulation des systèmes MPSoC, notre solution consiste à décrire le système au niveau PVT. La précision de la description à ce niveau pour les deux parties calcul et traitement dépend des objectifs attendus du système MPSoC à simuler. Dans notre travail nous avons fixé les objectifs suivants :

- La vérification fonctionnelle du système qui consiste à exécuter l'application cible sur une architecture définie et s'assurer de l'exactitude du résultat de simulation.
- Une analyse de performance du système sans trop pénaliser le temps de simulation. En effet, cette analyse consiste à définir un modèle de temps et à l'intégrer dans le niveau d'abstraction cible.
- La vérification du déploiement des tâches de l'application sur les processeurs et des données sur les bancs mémoires. L'estimation du temps d'exécution nous permet de savoir pour chaque tâche si celle-ci est exécutée dans les délais déterminés.
- La mesure des contentions sur le réseau d'interconnexion afin de vérifier l'adéquation du support de communication avec le système. Elle permet de détecter les goulots d'étranglement vers une cible particulière comme les bancs mémoires.

– L'analyse de la consommation d'énergie du système en vue de l'exploration architecturale.
– La diminution de l'effort de développement par rapport aux niveaux les plus bas. Dans notre étude, nous allons nous baser sur le critère *nombre de lignes de code* (LOC) développées pour comparer la modélisation à différents niveaux d'abstraction.

Le nombre important d'objectifs auquel s'ajoute le large espace de solutions architecturales nécessitent une stratégie d'exploration rapide. Les propositions faites dans ce chapitre doivent permettre une approche multi-niveaux ce qui faciliterait l'élimination d'alternatives à chaque niveau jusqu'à converger vers la solution la plus adéquate en bas niveau. Pour atteindre les objectifs précédemment définis, nous avons développé plusieurs sous-niveaux à l'intérieur du niveau PVT à savoir le PVT-PA (*PVT Pattern Accurate*), PVT-TA (*PVT Transaction Accurate*) et le PVT-EA (*PVT Event Accurate*). En effet, certains objectifs ne demandent pas un niveau de détail élevé. Une simulation rapide à un niveau d'abstraction élevé est dans ce cas suffisante.

Chaque sous-niveau propose un compromis accélération/précision différent. Les trois sous-niveaux ont en commun les caractéristiques du niveau PVT à savoir : une architecture matérielle définie et des outils de mesure des performances et de la consommation d'énergie. Néanmoins, chaque sous-niveau se distingue des autres sous-niveaux par les points suivants :

– le sous-niveau PVT-PA permet une exécution fonctionnelle de la partie calcul, donc il offre le facteur d'accélération de la simulation le plus élevé.
– Le sous-niveau PVT-TA utilise un simulateur au niveau instruction pour exécuter l'application. Il simule fonctionnellement le comportement de la micro-architecture et ne prend pas en considération les spécifications du protocole de communication. Le sous-niveau PVT-TA présente le meilleur compromis accélération de la simulation/précision.
– Le sous-niveau PVT-EA tient compte des délais entre les événements internes de l'architecture (exemple : exécution d'une instruction, accès en lecture mémoire, etc.) et implémente un protocole de communication ce qui lui permet d'atteindre le maximum de précision. Dans ce qui suit, nous allons détailler la description de ces trois sous-niveaux.

4.4 Le sous-niveau PVT Pattern Accurate (PVT-PA)

Nous avons défini deux objectifs pour le sous-niveau PVT-PA. En effet, ce sous-niveau doit d'un côté permettre au développeur la vérification fonctionnelle du système et de l'autre côté il doit rendre possible l'observation des contentions sur le réseau d'interconnexion et la récupération des informations temporelles correspondantes. Ce deuxième objectif représente la spécificité de PVT-PA par rapport au niveau PV qui n'inclut pas des spécifications temporelles. Les résultats de simulation à ce sous-niveau vont permettre au développeur de prendre des décisions sur l'adéquation entre le réseau d'interconnexion et le flux de communication de l'application. Il en est de même pour le déploiement des tâches sur les processeurs et le placement des données sur les bancs mémoires.

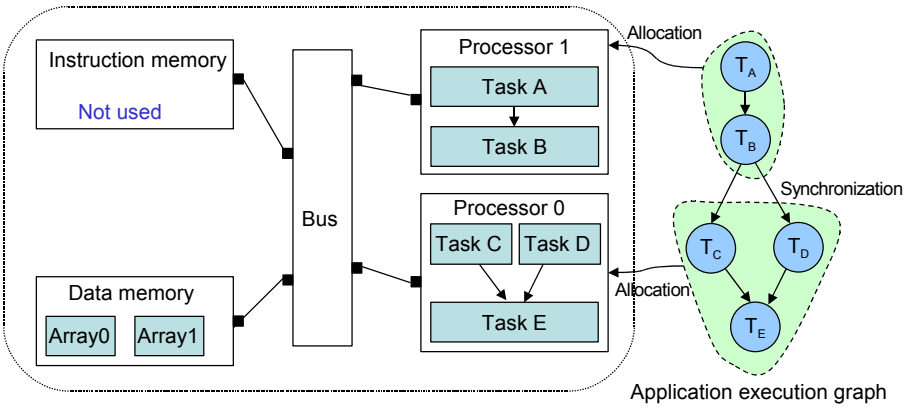


FIG. 4.1 – *Simulation logicielle/matérielle au sous-niveau PVT-PA*

4.4.1 Structure de données

La proposition d'un sous-niveau PVT-PA est justifiée par le domaine d'application, à savoir le traitement de données intensif. Dans ce contexte le nombre d'accès à la mémoire d'instructions est négligeable par rapport au nombre d'accès aux mémoires de données. Pour cela, la partie calcul est exécutée à un niveau d'abstraction assez élevé. A titre d'exemple, les processeurs ne sont pas décrits à travers des ISS (*Instruction Set Simulator*) mais réalisent plutôt l'ensemble de tâches de l'application. Ces tâches seront ensuite exécutées par la machine de simulation. Néanmoins, nous allons simuler les accès aux mémoires de données au niveau de l'architecture. La figure 4.1 illustre la simulation logicielle/matérielle au niveau PVT-PA.

La structure de données traitée par les tâches est nommée *Motif* (*Pattern*). Un motif peut être une variable simple ou bien un tableau à dimension variable. A titre d'exemple, pour une application de multiplication de deux matrices 32x32, le motif de données traité peut être un tableau à deux dimensions (32x32) ou bien un vecteur de dimension (32). Avant l'exécution d'une tâche, une phase de lecture des motifs d'entrées à partir de la mémoire est réalisée. De même après l'exécution de la tâche, un transfert des données résultats dans l'autre sens pour stocker les motifs de sorties est effectué. En comparaison au niveau CABA, le sous-niveau PVT-PA permet de simuler les mêmes tailles de transfert de données. D'où notre appellation PVT "précis au niveau motif" ou bien PVT Pattern Accurate (PVT-PA).

4.4.2 Synchronisation des tâches et ordonnancement

Pour assurer la cohérence des données lors de l'exécution de l'application, un mécanisme de synchronisation entre les tâches est nécessaire. Ce mécanisme est nécessaire pour vérifier la cohérence de données entre deux tâches successives exécutées sur des processeurs différents. Dans notre exemple de la figure 4.1, le mécanisme de synchronisation vérifie que la tâche C ou D du processeur 0 n'est exécutée qu'après la fin d'écriture du motif issue de la tâche B du processeur 1. Dans ce cas la tâche B est appelée producteur et les tâches C et D sont appelées consommateurs. Le mécanisme de synchronisation doit pouvoir gérer le cas de plusieurs producteurs et plusieurs consommateurs. C'est le cas si les tâches B, C ou D sont parallélisées sur plusieurs processeurs. Cette synchronisation des tâches n'est utile que lorsque les tâches sont placées sur des processeurs différents. Autrement, lorsque les tâches

sont exécutées sur le même processeur, il est possible de les ordonnancer statiquement selon un ordre respectant les dépendances de données.

Le mécanisme que nous avons implémenté peut être décrit comme suit : au début tous les processeurs de type consommateur sont bloqués. Pour chaque tâche de type producteur, lorsqu'elle est terminée elle indique la disponibilité du motif qu'elle doit fournir. Lorsque toutes les tâches producteurs ont indiqué la disponibilité de leurs motifs, chacune des connexions de sortie reçoit l'autorisation de débloquer les lecteurs. Une fois qu'une tâche consommateur a toutes ses connexions débloquées, elle commence à s'exécuter. Pendant ce temps-là, les tâches qui ont produit les motifs sont bloquées afin de ne pas récrire par-dessus les motifs qui sont en cours de lecture. Les variables de synchronisation sont déclarées au niveau du simulateur SystemC sans exécuter réellement les accès mémoires correspondants. Ceci nous permet d'accélérer la simulation au sous-niveau PVT-PA.

Pour traiter l'ordonnancement des tâches d'un même processeur, c'est le cas des tâches C et D du processeur 0, plusieurs possibilités existent. La solution la plus simple est d'effectuer un ordonnancement statique séquentiel en respectant le graphe de dépendance entre les tâches. Cette solution est simple à implémenter, mais ne permet pas d'obtenir le maximum de performance de l'application. En effet, nous ne pouvons pas savoir à l'avance quel motif de la tâche sera disponible en premier. Ceci ne dépend pas seulement du temps d'exécution de la tâche, mais dépend aussi des contentions sur le réseau d'interconnexion. Donc avec un ordonnancement statique, nous pouvons nous retrouver dans le cas où une première tâche attend qu'un motif soit disponible alors que nous avons déjà une deuxième tâche qui peut être exécutée. Pour remédier à cet inconvénient, la solution consiste à réaliser un ordonnancement dynamique. Seule cette dernière solution, a été implémentée comme ordonnanceur de notre simulateur. En résumé, la partie calcul au sous-niveau PVT-PA est structurée comme suit :

- Réaliser une synchronisation au début de la tâche avec les autres tâches exécutées sur des processeurs différents.
- Lire les motifs d'entrées à partir de la mémoire pour l'exécution de chaque tâche.
- Exécuter une ou plusieurs tâches.
- Écrire des motifs de sorties dans la mémoire après l'exécution de chaque tâche.
- Réaliser une synchronisation à la fin de la tâche avec les autres tâches exécutées sur des processeurs différents.

Au sous-niveau PVT-PA, cette description est intégrée dans le composant processeur en utilisant le processus SC_THREAD de SystemC. Ce choix est dû principalement à la nécessité d'utiliser des instructions *wait()* dans notre description pour différentes raisons. En premier lieu, nous utilisons les instructions *wait()* lorsque toutes les tâches d'un processeur sont en attente des motifs d'entrées. A ce moment, l'ordonnanceur a besoin de vérifier s'il y a une tâche prête à être exécutée à chaque période de temps, comme nous allons le décrire dans la section 4.8.1. En second lieu, nous avons besoin d'utiliser les instructions *wait()* pour les intégrer dans le modèle de temps pour l'estimation de performance.

Le texte ci-dessous est un exemple de thread pour implémenter la fonctionnalité de la partie calcul décrite au sous-niveau PVT-PA. Cet exemple concerne la description du processeur 0 de la figure 4.1.

```
void MIPSCore::run()
{
//Tâche_C
```

```
 void task_C (struct tlmpa_task* a)
 {
// Déclaration des tableaux correspondant aux motifs de la tâche.
   float Tab_in[16]; float Tab_out[16];
// Synchronisation avec les autres tâches exécutées sur d'autres processeurs
   sync_1.startReading(B);
// Lecture du motif d'entrée de la tâche C
   for ( int i=0 ; i<=15 ; i++ ){
   Tab_in[i]=Port.read(addr+i, data, t);
   Wait(Transaction_delay,NS)
   }
// Exécution de la Tâche_C
   Tâche_C(Tab_in, Tab_out );
   Wait(Task_delay,NS)
// Ecriture du motif de sortie de la tâche C
   for ( int i=0 ; i<=15 ; i++){
   Port.write(addr+i, Tab_out[i], t);
   Wait(Transaction_delay,NS)
   }
// Synchronisation avec les autres tâches exécutées sur d'autres processeurs
   sync_1.finishWriting(B);
 ...}

  void task_D (struct tlmpa_task* a)
  {
  ...
  }
  void task_E (struct tlmpa_task* a)
  {
  ...
  }
}
```

4.4.3 Les communications

Au sous-niveau PVT-PA, la communication entre les composants est décrite à un niveau plus haut que le niveau CABA. Les transactions sont réalisées à travers des canaux de communication (appelés aussi *Channels*) au lieu des signaux comme ceci est réalisé au niveau CABA. Les canaux de communication implémentent une ou plusieurs interfaces. Chaque interface est définie par un ensemble de méthodes *read()* et *write()*. Pour le chargement (*load*) et le stockage (*store*) des données, les initiateurs (processeurs, contrôleur DMA, etc.) instancient des appels de fonctions *read()* et *write()* qui seront transmis via le port. A titre d'exemple dans le texte ci-dessus nous utilisons les deux fonctions suivantes : *Port.read(addr+i, data, t)* et *Port.write(addr+i, Tab_out[i], t)* où le premier paramètre présente l'adresse de la donnée, le deuxième paramètre présente la variable qui reçoit la donnée et le dernier paramètre présente le temps mis par la transaction. Au niveau des cibles (mémoire, accélérateur matériel, etc.), les transactions seront récupérées pour exécuter la méthode correspondante et envoyer

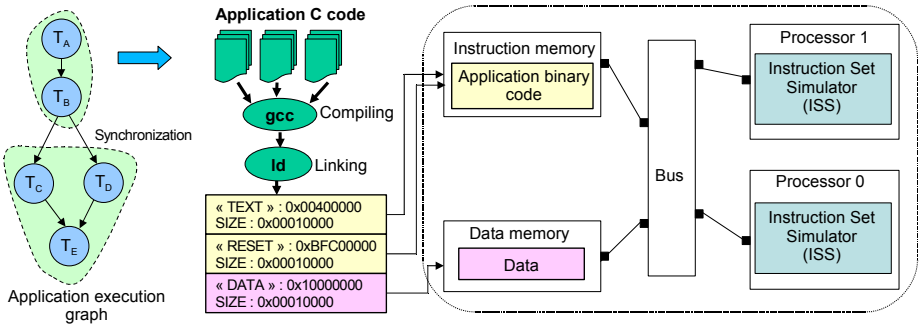


FIG. 4.2 – *Simulation logicielle/matérielle au sous-niveau PVT-TA*

la réponse à l'initiateur. A ce sous-niveau, un modèle de temps a été défini pour estimer le temps d'exécution de l'application. Ce modèle sera détaillé dans la section 4.8.1.

En résumé, le sous-niveau PVT-PA présente l'avantage d'un gain important en terme d'accélération de la simulation. Ceci est dû principalement au fait que nous avons remplacé l'implémentation de l'architecture des processeurs du système par le code de l'application. Cela permet de réaliser une simulation logicielle/matérielle même si le composant processeur n'est pas disponible. Des résultats préliminaires peuvent être obtenus assez tôt dans le processus de développement ce qui peut réduire le temps de développement. En plus, lorsque nous exécutons les tâches de l'application directement sur la machine de simulation, il est possible d'utiliser les outils conventionnels de débogage qui sont généralement maîtrisés par les développeurs pour la vérification fonctionnelle. En contre-partie de ces avantages, nous pensons que le transfert de données au niveau des motifs ne reflète pas ce qui se passe réellement dans le réseau d'interconnexion au niveau des transactions ce qui altère l'estimation de performance au sous-niveau PVT-PA. Le choix d'un motif à granularité fine peut être une solution pour remédier à cette source d'erreur. Mais ceci nous ramène à une décomposition fine des tâches de l'application ce qui va surcharger la simulation et contredit le premier objectif défini pour le sous-niveau PVT-PA.

4.5 Le sous-niveau PVT Transaction Accurate (PVT-TA)

L'objectif de ce sous-niveau est d'obtenir des estimations de performance plus précises que celles obtenues par le sous-niveau PVT-PA. Les solutions architecturales retenues du sous-niveau PVT-PA par le développeur peuvent être explorées une deuxième fois au sous-niveau PVT-TA afin d'obtenir des estimations plus précises par la prise en compte du flux d'instructions. Par rapport au sous-niveau PVT-PA, notre idée majeure est d'utiliser des simulateurs de processeurs au niveau des instructions (ISS) pour l'exécution de l'application afin d'augmenter la précision d'estimation de performance. Les tâches de l'application sont compilées afin d'obtenir un code binaire qui sera ensuite exécuté à partir de la mémoire d'instructions. La figure 4.2 illustre la simulation logicielle/matérielle au niveau PVT-TA. Avant de commencer la simulation, nous initialisons les différents segments de données et d'instructions dans les mémoires appropriées. En plus, pour chaque processeur nous définissons l'adresse du début de programme de chaque tâche à exécuter.

4.5.1 Structure de données

Au niveau PVT-TA, l'ISS qui simule la fonctionnalité d'un processeur sera exécuté sur la machine de simulation. Il modélise le processeur cible au niveau du jeu d'instructions. Pendant la simulation, l'application sera exécutée instruction par instruction en suivant ces 4 étapes : la lecture du code de l'opération à partir de la mémoire d'instructions, le décodage, la lecture des opérandes à partir de la mémoire de données et l'exécution de l'instruction. Pour supporter les ISS au sous-niveau PVT-TA dans un environnement de simulation MPSoC, cette fonctionnalité est décrite dans un module SystemC en utilisant le processus SC_THREAD. Ce choix est dû principalement à la nécessité d'utiliser des instructions *wait()* dans notre description pour deux raisons. La première est liée à notre choix d'utiliser des canaux de communication de type "bloquant" pour connecter les différents composants (plus de détails dans la section 4.7.1). Autrement dit, nous utilisons des instructions *wait()* sensitives sur des événements (dans notre cas c'est l'arrivée des réponses) ce qui permet au noyau SystemC d'ordonnancer les différents processeurs au niveau de chaque instruction. Ceci a comme avantage d'éviter qu'un processeur avance dans le temps sans que les autres le fassent. La deuxième raison est que nous avons besoin d'utiliser les instructions *wait()* dans le modèle de temps pour l'estimation de performance comme il sera détaillé dans la section 4.8.2.

Le texte ci-dessous est un exemple de thread pour implémenter la fonctionnalité de la partie calcul (processeur) décrite au sous-niveau PVT-TA.

```
void MIPSCore::run()
{
// Boucle d'exécution
  while(next_pc!=0)
  {
// Lecture du code de l'opération à partir du cache d'instructions
  initiator_ins_port.read( next_pc , instruction , time );
  Wait(Transaction_delay,NS)
// Décodage des instructions
  IDecode(instruction, ins_opcode);
// Execution des instructions
  switch (ins_opcode) {
    case OP_ADD:
    {...
    Wait(ADD_delay,NS)
    }
    break;
    case OP_BEQ:
    {...
    Wait(BEQ_delay,NS)
    }
    break;
// Cas de chargement de données
    case OP_LB:
    {...
```

```
     initiator_data_port.read( adr , d , time );
     Wait(Transaction_delay,NS)
    }
    break;
// Cas de stockage de données
    case OP_SB:
    {...
     initiator_data_port.write( adr , d , time );
     Wait(Transaction_delay,NS)
    }
    break;
    ...
}
```

Dans ce texte nous identifions les étapes nécessaires pour l'exécution de l'application au niveau des instructions. Le processeur commence par une phase de lecture de la prochaine instruction à partir du cache d'instructions en initialisant une requête *initiator_ins_port.read(next_pc, instruction, time)*. Le paramètre *next_pc* spécifie l'adresse contenue dans le registre compteur programme, *instruction* représente la variable dans la quelle nous devons recevoir l'instruction et le paramètre *time* est utilisé pour l'estimation du temps d'exécution comme il sera détaillé dans la section 4.8.2. La deuxième phase consiste à décoder l'instruction pour identifier le type de l'opération via la fonction *IDecode(instruction, ins_opcode)*. La phase suivante est la lecture des opérandes à partir de la mémoire si c'est nécessaire. La dernière phase consiste en l'exécution de l'instruction courante et la mise à jour des registres du processeur et du compteur programme.

La description au sous-niveau PVT-TA produit le même comportement du point de vue des transactions que le niveau CABA, d'où notre appellation PVT "précis au niveau transaction" ou bien PVT Transaction Accurate (PVT-TA).

4.5.2 Synchronisation des tâches et ordonnancement

Au sous-niveau PVT-TA, pour assurer la cohérence des données entre plusieurs tâches exécutées sur différents processeurs nous avons implémenté un mécanisme de synchronisation. Dans ce mécanisme, les processeurs partagent des variables de synchronisation qui peuvent être lues et écrites seulement à partir de la mémoire de données partagée et qui ne sont pas mises dans les caches. D'une autre façon, ces variables peuvent être vues comme des verrous au niveau des tâches qui nécessitent de la synchronisation. C'est le cas des tâches B et C ou B et D dans la figure 4.2. L'avantage de ce mécanisme est qu'il peut être utilisé pour différentes topologies de réseaux d'interconnexion. Néanmoins, la synchronisation avec des variables partagées ne permet pas d'obtenir le maximum des performances en comparaison avec des solutions liées à des types de réseaux d'interconnexion particuliers. C'est le cas du mécanisme d'espionnage *snoop* utilisé pour le bus.

Au sous-niveau PVT-TA, la simulation des accès mémoires des variables de synchronisation va nous permettre d'éviter l'utilisation de l'ordonnanceur de tâches. En effet, une tâche synchronisée avec d'autres tâches sera exécutée lorsque son verrou est débloqué. Lorsque plusieurs tâches peuvent être ordonnancées en même temps, celle qui va avoir en premier son verrou débloqué sera exécutée. Ceci au sous-niveau PVT-PA n'était pas possible, car les

variables de synchronisation sont déclarées au niveau du simulateur SystemC donc au point de vue architecture nous ne pouvons pas savoir leur instant de déblocage contrairement au sous-niveau PVT-TA.

4.5.3 Les communications

La communication entre les composants au sous-niveau PVT-TA est la même que celle décrite au sous-niveau PVT-PA. Comme au sous-niveau PVT-PA, les spécifications du protocole de communication ne sont pas définies. Les événements internes de l'architecture se produisent successivement sans respecter les délais entre eux ce qui présente une source d'erreur pour l'estimation de performance.

En résumé le sous-niveau PVT-TA intègre des simulateurs de processeurs au niveau des instructions (ISS) pour l'exécution de l'application. Il permet de visualiser les mêmes transactions que celles simulées au niveau CABA ce qui minimise l'erreur d'estimation de performance. Néanmoins, ceci est obtenu au prix d'une augmentation du temps de simulation. En effet, à l'exception du composant processeur, la description des autres composants aux sous-niveaux PVT-PA et PVT-TA est identique. De ce fait, la vitesse de simulation dans ces deux sous-niveaux est semblable. Le passage du sous-niveau PVT-PA au PVT-TA peut se faire avec une étape de raffinement automatique en compilant les tâches de chaque processeur avec le compilateur approprié. Il nécessite néanmoins la disponibilité du même composant processeur à deux niveaux d'abstraction différents. Les segments du code binaire générés seront ensuite placés dans la mémoire d'instructions. Nous remarquons que l'utilisation des ISS rend difficile l'intégration des outils de débogage au cours de la simulation puisque le jeu d'instructions du processeur cible est à priori différent de celui de la machine de simulation.

4.6 Le sous-niveau PVT Event Accurate (PVT-EA)

L'objectif du sous-niveau PVT-EA est de pouvoir atteindre des estimations de performance très proches de celle du niveau CABA. Pour ce faire, nous avons identifié les informations intéressantes à ajouter au sous-niveau PVT-TA concernant les deux parties calcul et communication pour augmenter la précision sans pénaliser de façon importante le temps de simulation. Parmi les détails de la micro-architecture, la prise en compte des spécifications du protocole de communication et le respect des délais entre les événements ont montré expérimentalement les conséquences sur l'amélioration de la précision d'estimation.

En effet, dans un système MPSoC plusieurs tâches sont exécutées simultanément. Pour assurer une exécution dans l'ordre des événements entre les processeurs, il est nécessaire de prendre en considération les délais d'exécution correspondants (exécution des instructions, préparation des requêtes commande et réponse, transmission des requêtes via le réseau d'interconnexion, etc.). Cette attention aux délais d'exécution permet d'établir le même séquencement d'événements que celui obtenu par simulation au niveau CABA, d'où notre appellation PVT "précis au niveau événements" ou bien PVT Event Accurate (PVT-EA). Néanmoins, certaines activités peuvent avoir un délai d'exécution imprévisible. Comme exemple, nous citons le cas de l'exécution désordonnée des instructions dans un processeur superscalaire. Le temps de terminaison d'un tel type d'événement étant imprévisible, cela constituera une source d'erreur possible pour l'estimation de performance. Au sous-niveau PVT-EA, la structure de données traitées, la synchronisation et l'ordonnancement des tâches se présentent de

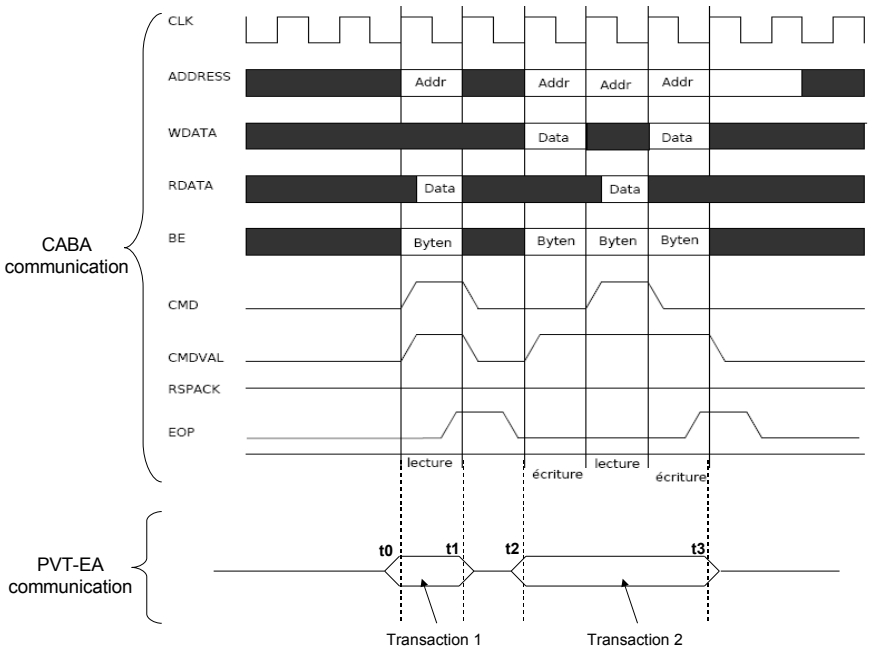


FIG. 4.3 – *Communications aux niveaux CABA et PVT-EA*

la même façon qu'au sous-niveau PVT-TA.

4.6.1 Les communications

Au sous-niveau PVT-EA, les spécifications du protocole de communication utilisé doivent être précisées par l'utilisateur ainsi que les délais d'exécution correspondants. Les délais de la micro-architecture, comme le temps de réponse du réseau d'interconnexion, doivent aussi être précisés. Ces délais peuvent être estimés par l'utilisateur ou mesurés à un bas niveau comme cela sera détaillé dans la section 4.8.3. Cette opération est très importante pour obtenir un comportement réaliste de la partie communication en comparaison avec le niveau CABA. La figure 4.3 illustre le séquencement de deux transactions de tailles différentes. La transaction 1 contient un seul mot alors que la deuxième contient trois mots. L'instant auquel chaque transaction arrive au réseau d'interconnexion intervient pour définir l'ordre des accès aux ressources partagées vis-à-vis des autres transactions venant d'autres processeurs. Ceci permet d'obtenir un séquencement d'événements réaliste et d'offrir le maximum de précision. Le défi au sous-niveau PVT-EA est de détecter les instants auxquels se produisent les transactions et ceux auxquels elles arrivent au réseau d'interconnexion. Or ceci ne peut se faire qu'en prenant compte du nombre de mots dans la requête, du temps de préparation du paquet et des contentions sur le réseau d'interconnexion.

L'avantage majeur du sous-niveau PVT-EA est qu'il permet de prendre en considération les spécifications du protocole de communication au plus tôt dans le processus de développement (avant le niveau CABA). Il offre aussi la possibilité d'estimer le temps d'exécution d'une façon plus précise que PVT-PA et PVT-TA. En contrepartie, PVT-EA génère une accélération de simulation plus faible que les deux autres sous-niveaux. Le passage du sous-niveau PVT-TA au sous-niveau PVT-EA peut se réaliser avec une étape de raffinement en ajoutant

les détails du protocole et de temps à la description du système MPSoC au sous-niveau PVT-TA. La prise en compte de ces détails correspond à insérer des instructions *wait()* avec les arguments appropriés dans la description de l'architecture du système.

4.7 Modèles de composants pour la conception des MPSoC

Dans cette section nous allons présenter les modèles de composants qui ont été développés aux 3 sous-niveaux. Ces composants sont génériques et nous permettent d'évaluer les performances pour une classe de systèmes MPSoC à mémoires partagées aux sous-niveaux PVT. Les modèles de composants conçus dans notre travail sont : le processeur, les mémoires caches, les réseaux d'interconnexion (bus, crossbar), les mémoires d'instructions et de données, le contrôleur DMA et l'accélérateur matériel TCD-2D (Transformée en Cosinus Discrète). Ces composants sont partagés principalement en deux classes : actifs et passifs. Un composant est dit actif s'il est décrit à travers un thread, autrement il est dit passif. Notons, qu'un composant actif peut être initiateur ou cible suivant s'il peut initier ou non une transaction. A l'opposé un composant passif est toujours vu comme une cible. Lors de la description de chaque composant, nous allons justifier les choix qui ont été réalisés.

En utilisant la version 2.1 de SystemC et la version 1 de la bibliothèque TLM, chacun des composants a été conçu et implémenté aux 3 sous-niveaux PVT-PA, PVT-TA et PVT-EA. Cette section 4.7 présente les éléments communs aux 3 sous-niveaux pour tous les composants. C'est dans la section 4.8 que les différences entre les sous-niveaux seront soulignées.

4.7.1 Modèle du processeur

Avec l'approche TLM, le comportement d'un processeur peut avoir 3 principales descriptions (figure 4.4). La description la plus précise du processeur est celle où un simulateur au cycle précis est utilisé (CAS dans la figure 4.4). Dans cette description, les détails liés à la micro-architecture sont simulés tels que les étages du pipeline et la prédiction de branchement ce qui permet d'obtenir un modèle de processeur précis. Le processeur MIPS R3000 utilisé dans SoCLib correspond à cette description. Dans la deuxième description, le processeur est modélisé avec un simulateur précis au niveau des instructions (*ISS : Instruction Set Simulator*). Les instructions sont dans ce cas exécutées séquentiellement sans se référer à la micro-architecture du composant. La spécification des délais d'exécution de chaque instruction permet d'estimer le temps d'exécution de l'application. Pour implémenter cette description à l'échelle ISS, nous avons modifié la description du processeur MIPS R3000 au niveau CABA pour qu'elle soit utilisée dans la simulation des deux sous-niveaux PVT-TA et PVT-EA. Dans la troisième description, le processeur réalise l'ensemble de tâches de l'application. Ces tâches seront ensuite exécutées par la machine de simulation. Néanmoins, des annotations de temps peuvent être ajoutées aux tâches pour estimer approximativement le temps d'exécution de l'application. Cette description est utilisée pour modéliser un processeur au niveau PVT-PA.

La figure 4.4 illustre la variation de l'accélération de la simulation en fonction de la précision de la description. L'utilisation des modèles de processeur aux niveaux CABA ou RTL permet d'obtenir des estimations précises avec des temps de simulation importants. A l'opposé, l'exécution native de l'application ou bien l'utilisation des ISS permettent d'accélérer la simulation au prix d'une perte de précision. Pour pallier cet inconvénient, nous avons

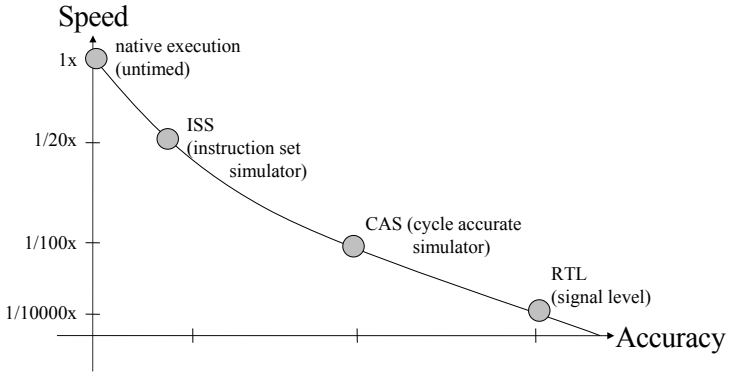


FIG. 4.4 – *Niveaux de modélisation du processeur [88]*

développé des modèles d'estimation de performance aux sous-niveaux PVT-PA, PVT-TA et PVT-EA qui minimisent l'erreur d'estimation.

Dans chaque sous-niveau, la fonctionnalité du processeur est décrite avec un module SystemC en utilisant le processus SC_THREAD. Par conséquent, un processeur est considéré comme un composant actif. En effet, il y a deux façons possibles de connecter le processeur à d'autres composants de l'architecture.

- La première façon est de faire communiquer le processeur avec des composants passifs donc cibles comme un bus, une mémoire, etc. Dans ce cas les opérations de ces derniers composants sont exécutées avec le contrôle du thread du processeur. La figure 4.5 illustre ce principe. Lorsqu'une nouvelle transaction est réalisée, le processeur envoie une requête *read()* ou *write()* et reste bloqué. Le thread reste actif dans le modèle du bus ou de la mémoire jusqu'à ce qu'il reçoit la réponse de la requête. Ici, nous parlons d'opération de lecture ou écriture bidirectionnelle puisque nous utilisons le même chemin pour la requête et la réponse. Une telle implémentation permet d'augmenter les gains en accélération de simulation, car il n'y a pas de changement de contexte nécessaire pour l'ordonnanceur SystemC. Néanmoins, avec cette implémentation, il n'est pas possible de modéliser des composants qui s'exécutent simultanément. A titre d'exemple, pour augmenter les performances (temps d'exécution), le processeur peut continuer à exécuter son code sans attendre la réponse de la mémoire s'il n'a pas besoin de la donnée. De plus, un processeur peut être connecté à un autre composant actif comme le cache de données ou d'instructions, ce qui empêche l'utilisation de ce mécanisme.
- La deuxième technique consiste à faire communiquer le processeur avec des composants actifs, comme le contrôleur DMA ou le cache de données ou d'instructions. Dans ce cas, la communication entre les deux composants n'est pas directe. Elle se fait à travers un composant intermédiaire qui peut être un canal de communication (*Channel*) ou bien un réseau d'interconnexion. La figure 4.6 montre la façon de modéliser une opération de lecture ou d'écriture par un message unidirectionnel pour chacun des deux composants actifs. Dans ce cas, le processeur ou l'initiateur envoie une nouvelle requête et continue à s'exécuter. Au moment où il a besoin de la donnée, celle-ci sera récupérée du canal de communication une fois que la réponse de la cible est reçue. Ce mécanisme permet une modélisation plus exacte de ce qui se passe en bas niveau. Cependant, l'utilisation des threads pour implémenter les composants de type cible

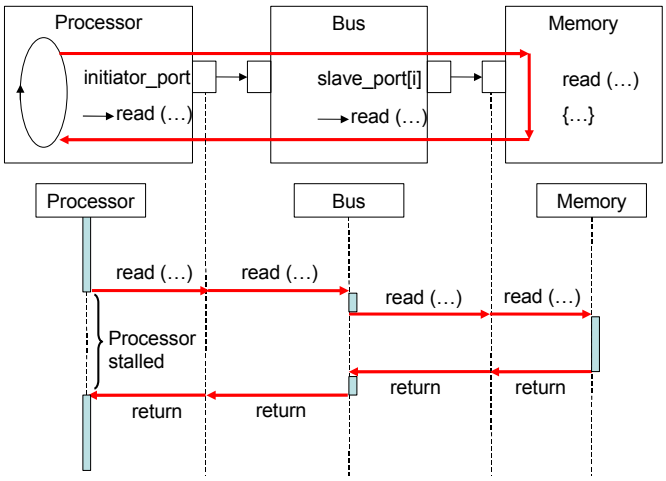


FIG. 4.5 – *Communication bloquante entre un composant actif (processeur) et un composant passif (mémoire)*

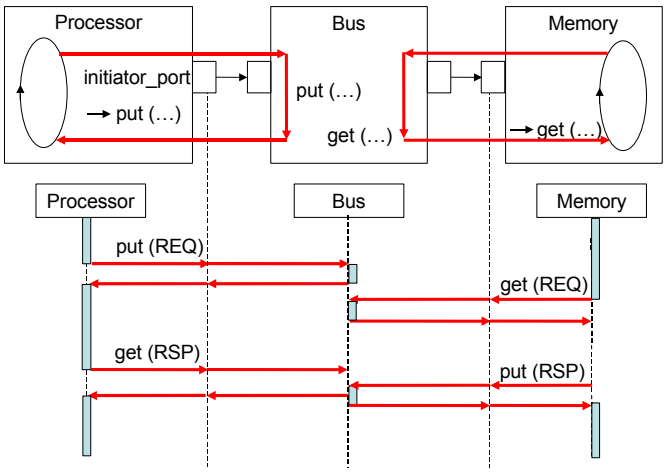


FIG. 4.6 – *Communication non-bloquante entre composants actifs*

ralentit les performances en terme de temps de simulation.

4.7.2 Modèle de la mémoire cache

Le modèle de la mémoire cache est habituellement intégré dans le composant processeur [51, 138]. A l'opposé, dans notre environnement, ce composant est représenté par un modèle séparé du processeur. L'architecture du cache utilisée (taille, associativité, stratégie d'écriture, etc.) influe sur les caractéristiques du système en terme de performance, surface et consommation d'énergie. Par ailleurs, dans un système MPSoC, nous avons besoin de spécifier un mécanisme de cohérence de cache qui dépend du type de réseau d'interconnexion (Bus, Crossbar, etc.) utilisé. Ce mécanisme peut être implémenté soit à travers une solution matérielle, telle que l'espionnage du bus ou le répertoire centralisé [136, 81], ou bien à travers une solution logicielle [134, 112]. Comme on peut le voir la séparation entre le modèle du processeur et celui de la mémoire cache permet d'obtenir une modularité de la conception.

Ceci offre la possibilité d'explorer et d'évaluer différentes architectures.

Nous avons conçu un module mémoire cache générique permettant à l'utilisateur de configurer la structure du composant. Les paramètres du cache sont : nombre de blocs, taille du bloc en octets, l'associativité, et la stratégie d'écriture (écriture simultanée ou différée). Ce module est un composant actif et joue le rôle de cible du côté processeur et initiateur du côté réseau d'interconnexion. Cette fonctionnalité est implémentée par un module SystemC possédant deux ports. Le premier port est de type *cible* pour communiquer avec le processeur connecté à la mémoire cache. Le deuxième port, de type *initiateur*, est relié au réseau d'interconnexion pour être utilisé dans le cas de défaut de cache (*cache miss*) ou dans le cas d'opération d'écriture de données en mémoires partagées.

La figure 4.7 montre les communications entre le processeur et le cache d'un côté et le cache et le réseau d'interconnexion de l'autre côté. Dans notre implémentation, nous avons choisi de connecter le processeur et le cache via des canaux de communication bidirectionnels bloquants. Nous pensons que ceci émule ce qui se passe réellement dans un processeur de type scalaire comme le processeur MIPS R3000. En effet, ce processeur permet une exécution dans l'ordre des instructions avec une vitesse d'une instruction/cycle. Le processeur ne peut pas avancer dans l'exécution lors d'un défaut de cache avant d'avoir la donnée manquante. Ceci explique le choix des interfaces bloquantes. Cependant, pour des architectures de processeurs plus complexes, comme les processeurs superscalaires, il sera intéressant d'utiliser des interfaces non bloquantes. Ceci va permettre au processeur l'exécution les instructions qui ne dépendent pas de la donnée manquante dans le cache (exécution désordonnée).

Du côté cache/réseau d'interconnexion, les transferts peuvent être réalisés selon deux modes. Le premier correspond au mode simple et consiste à transférer une seule donnée. Le deuxième correspond au mode rafale. Ce dernier est utilisé dans le cas de défaut de cache pour augmenter les performances. La figure 4.7 détaille le mode lecture en rafale dans le cas de défaut de cache. Ainsi, une seule requête commande du cache permet de charger plusieurs données correspondant à des adresses contiguës. Deux spécifications temporelles ont été ajoutées pour estimer le temps d'exécution : le temps d'accès et le temps de cycle à la mémoire cache. Ces deux spécifications ont été obtenues en utilisant l'outil CACTI [30].

4.7.3 Modèle du réseau d'interconnexion

La fonction du réseau d'interconnexion consiste à router les requêtes commandes générées par les initiateurs (processeurs, caches) vers les différentes cibles (mémoires, périphérique d'E/S) et les requêtes réponses dans le sens opposé. Dans cette thèse, nous nous limitons à l'utilisation des composants bus et crossbar. Ce dernier que nous avons développé est basé principalement sur les deux unités fonctionnelles : le routeur et l'arbitre (figure 4.8). Le routeur est un composant générique qui permet de diriger une requête provenant d'un initiateur vers la cible concernée, en utilisant une table de routage spécifiée. Ce module est passif et exécuté sous le contrôle du thread de l'initiateur auquel il est relié. A titre d'exemple dans la figure 4.7, le routeur est exécuté sous le contrôle du cache d'instructions ou de données. Comme le montre la figure 4.8(a), lors d'une nouvelle transaction de l'initiateur, le routeur lit l'adresse correspondante et sélectionne un port de sortie. Pour simuler un comportement similaire au bas niveau, le routeur développé permet d'implémenter les spécifications des différents types de protocoles de communication (VCI, OCP, etc.). Cette prise en compte des détails du protocole permet, comme il sera présenté par la suite, d'obtenir des

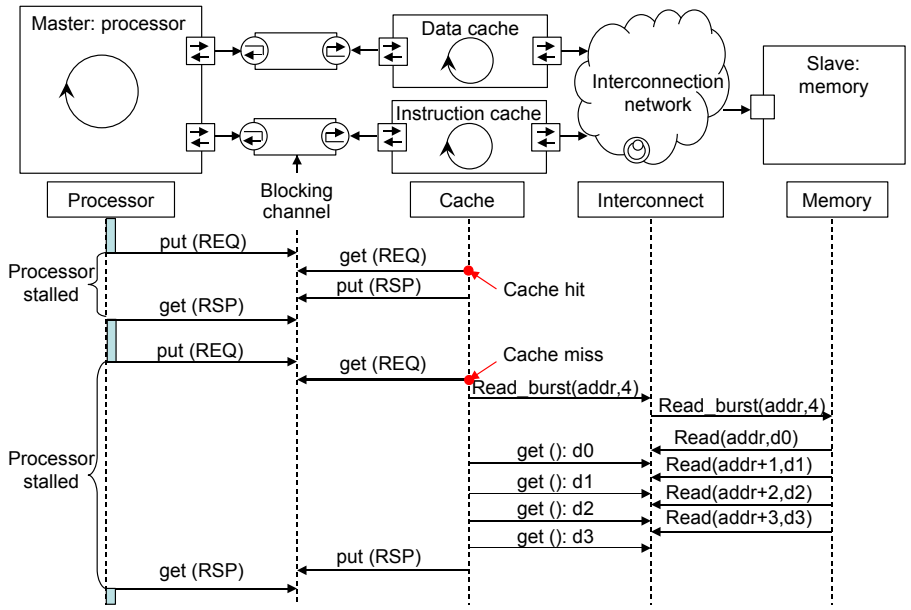
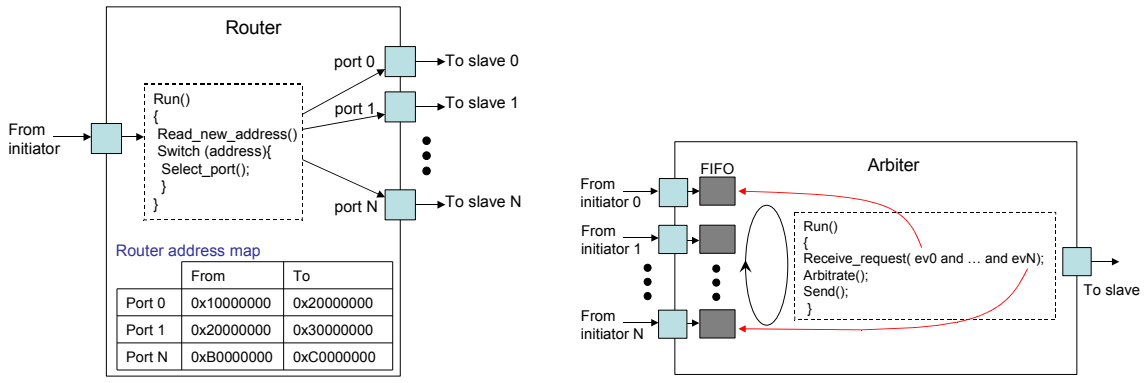


FIG. 4.7 – *Communication du module mémoire cache avec les autres composants dans le niveau PVT. Taille des blocs=4 mots*

estimations proches de celles du niveau CABA.



(a) Le composant routeur (b) Le composant arbitre

FIG. 4.8 – *Fonctionnement et composition des 2 composants : routeur et arbitre*

Pour gérer les conflits entre plusieurs demandes d'accès simultanées à une cible, il est nécessaire d'utiliser un composant dont le rôle est de définir la priorité de chaque requête. Pour ce faire, nous avons conçu un composant actif, nommé "arbitre" afin d'ordonnancer les accès aux ressources partagées. Lorsqu'un initiateur a besoin d'accéder à une cible partagée, via le réseau d'interconnexion, il transmet une requête en utilisant le canal de communication correspondant et attend la réponse. Au niveau de l'arbitre, un thread lit les requêtes présentes dans la FIFO de chaque canal de communication et sélectionne la requête prioritaire en se basant sur la stratégie d'arbitrage round-robin (figure 4.8(b)). Après le traitement par la cible, l'arbitre transmet la réponse dans la FIFO du canal de communication correspondante. L'initiateur récupère la réponse et termine la transaction. Cette gestion des communications

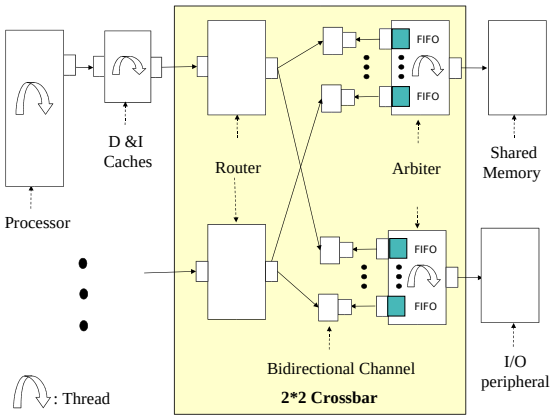


FIG. 4.9 – *Implémentation d'un crossbar 2x2*

bloque un routeur pendant le traitement de la requête par la cible, ce qui peut représenter un inconvénient. Elle a néanmoins comme avantage, une simplification du protocole et une réduction du nombre de ports. Ces deux facteurs permettent d'accélérer la simulation. Notre arbitre joue un deuxième rôle, très important, concernant l'estimation des délais dans le réseau d'interconnexion comme nous allons le détailler dans la section 4.8.1.

La figure 4.9 illustre l'implémentation d'un Crossbar 2x2 à partir des deux modules routeur et arbitre que nous venons de présenter. Cette architecture est certes relativement simple, mais suffisante pour atteindre notre objectif de pouvoir observer les contentions et récupérer les informations sur les latences. Par ailleurs à partir de cette architecture, plusieurs topologies de réseaux d'interconnexion, peuvent être conçues comme le réseau multi-étages. Les composants proposés (crossbar et bus) sont génériques et permettent à l'utilisateur de spécifier le nombre d'initiateurs et de cibles à connecter ainsi que la latence entre les différents ports pour l'estimation du temps d'exécution.

4.7.4 Modèle de la mémoire

Le module mémoire que nous avons conçu est un composant passif de type "slave" et comprend deux méthodes : read() et write(). Comme ceci a été expliqué dans la section 4.7.1, cette structure nous permet d'accélérer la simulation. En utilisant le mécanisme sc_export introduit dans la version SystemC 2.1, ces deux méthodes sont directement appelées et exécutées dans le thread de l'initiateur connecté au composant mémoire. Ce mécanisme de sc_export est utilisé pour modéliser toutes les cibles de notre système MPSoC tel que les périphériques d'entrées/sorties, etc. Son avantage est qu'il évite l'utilisation des threads pour modéliser les composants de type cible. Par conséquent, il augmente les performances en terme de temps de simulation.

Dans notre environnement, le port de la cible est connecté directement au routeur de l'initiateur si le module est privé à un seul processeur. A l'opposé, si le module est partagé entre plusieurs processeurs, il est connecté à l'arbitre correspondant. Pour améliorer les performances, des mémoires multi-ports peuvent être modélisées au niveau PVT pour simuler des accès simultanés à la même cible. Les paramètres temps d'accès et temps de cycle sont ajoutés à la description du composant pour l'estimation des performances.

4.7.5 Modèle de contrôleur DMA

Pour implémenter le contrôleur DMA au niveau PVT, nous avons simplifié la description du composant par rapport à sa description au niveau CABA. Le modèle contrôleur DMA au niveau PVT est toujours un composant SystemC de type actif, mais la FSM qui contrôle le composant a été supprimée. Ce composant joue le rôle d'un initiateur, pour réaliser le transfert d'un bloc de données d'une adresse source à une adresse de destination, et joue le rôle de cible configurée par un processeur. De ce fait, pour connecter le contrôleur DMA avec le réseau d'interconnexion, nous utilisons trois ports : un de type *cible* et deux de type *initiateur*. Ces deux derniers ports permettent un transfert de données simultané entre une adresse source et une autre destination.

Pour améliorer les performances au cours du transfert de données, le contrôleur DMA utilise aussi les modes de lecture et écriture en rafale comme nous l'avons expliqué dans le composant cache. Par ailleurs, pour imiter le fonctionnement CABA, les délais des événements de la micro-architecture sont mesurés et ajoutés au modèle PVT. Ces délais peuvent être utilisés ou non suivant le sous-niveau PVT dans lequel la simulation est faite.

4.7.6 Modèle de l'accélérateur matériel TCD-2D

La description de l'accélérateur matériel TCD-2D au niveau PVT dérive de notre implémentation au niveau RTL. Ce composant reçoit une séquence de vecteurs de 8 pixels chacun à partir du contrôleur DMA. Après la réception d'un bloc d'images (8x8), le composant opère successivement deux transformations TCD-1D sur les 8 lignes ensuite sur les 8 colonnes. A la fin du traitement, un transfert du résultat vers la mémoire partagée est réalisé par le contrôleur DMA. Un composant actif SystemC a été conçu pour intégrer la fonctionnalité de cet accélérateur. Pour estimer le temps d'exécution de ce composant, les délais de la micro-architecture sont mesurés à partir du niveau RTL et ajoutés au niveau PVT. A titre d'exemple, le temps de chaque transformation TCD-1D vaut 12 cycles.

4.8 Estimation de performance au niveau PVT

Cette section illustre l'utilisation des modèles de composant précédemment décrits pour l'estimation des performances pour les trois sous-niveaux PVT-PA, PVT-TA et PVT-EA. Un modèle de temps pour chaque sous-niveau a été développé. Ces modèles ont été par la suite intégrés dans le simulateur MPSoC en vue d'une exploration des architectures. La méthodologie d'estimation de performance doit satisfaire le critère de flexibilité pour qu'elle soit adaptable aux différentes architectures. La méthodologie proposée doit aussi prendre en considération les problèmes de temps liés à la synchronisation des processeurs, aux contentions dynamiques dans le réseau d'interconnexion et à la spécification du protocole de communication.

4.8.1 Estimation de performance dans PVT-PA

L'estimation de performance au sous-niveau PVT-PA revient à évaluer le temps d'exécution des deux parties calcul et communication. Comme nous l'avons déjà expliqué, dans PVT-PA les tâches sont exécutées nativement sur la machine de simulation. Ceci rend difficile l'estimation du temps d'exécution réel sur un processeur ciblé comme exemple le MIPS

Field	description
task_time	Délai de la tâche
hit_time	Délai d'un succès de cache
miss_time	Délai d'un défaut de cache
waiting_time	Délai du réseau d'interconnexion
read_time	Délai de l'accès mémoire en lecture
write_time	Délai de l'accès mémoire en écriture

TAB. 4.1 – *Description des symboles*

R3000. Pour augmenter la précision, nous avons proposé de caractériser ces tâches du point de vue temps d'exécution en utilisant un simulateur de bas niveau. Dans notre cas, pour évaluer les délais de chaque tâche nous avons utilisé le simulateur du processeur MIPS R3000 au niveau CABA. L'estimation des délais d'exécution des tâches dépend bien sûr des paramètres architecturaux, tels que la taille de la mémoire cache d'instructions utilisée. Dans notre cas, ce problème a été simplifié en supposant que cette taille est suffisante pour stocker la totalité du code de la tâche. Pour évaluer le temps d'exécution de la partie calcul, à chaque processeur nous attribuons un compteur de temps local. Sa valeur est incrémentée après l'exécution d'une tâche. Pour respecter les délais d'exécution, des instructions wait() ont été ajoutées après le traitement des tâches (figure 4.10, étape 1).

Pour estimer les latences des communications lors d'une opération de lecture ou écriture des motifs, notre stratégie d'estimation consiste à identifier les activités pertinentes des composants qui interviennent dans les opérations de communication. Ceci concerne les mémoires caches, le réseau, le DMAC, les modules mémoires partagés et enfin l'accélérateur matériel. Ci-après la liste des activités pertinentes qui ont été identifiées :

- Pour le cache : le nombre de succès (*Hit*) et de défauts (*Miss*)
- Pour le réseau d'interconnexion : le nombre de paquets transmis et reçus
- Pour la mémoire partagée : le nombre d'accès en lecture et écriture
- Pour le contrôleur DMA : le nombre de mots transférés
- Pour l'accélérateur matériel TCD-2D : le nombre de blocs traités

L'estimation du temps d'exécution exige la spécification du délai pour chaque type d'activité. Dans notre approche, les délais d'exécution sont mesurés soit à partir d'une caractérisation physique du composant matérielle, soit à partir d'un modèle analytique. Par exemple, les délais d'accès à la mémoire cache sont obtenus en utilisant le modèle analytique proposé par l'outil CACTI [30]. La figure 4.10 montre l'utilisation du compteur de temps local du processeur et les délais des activités pour estimer le temps d'exécution au sous-niveau PVT-PA. Les symboles utilisés dans cette figure sont décrits dans le tableau 4.1.

Lors d'une opération de lecture de donnée de la mémoire cache, le processeur réalise un appel à la fonction (figure 4.10, étape 0). Le paramètre *time* est utilisé pour mesurer l'intervalle de temps entre la transmission de la requête et la réception de la réponse. Il sera récupéré par le processeur pour incrémenter son compteur de temps local. Dans le cas d'un succès de cache, ce temps correspond au temps d'accès pour une opération de lecture du cache (figure 4.10, étape 2). Dans le cas inverse, le cache envoie une nouvelle requête qui sera transmise via le réseau d'interconnexion à la cible concernée (figure 4.10, étape 3). Le temps de l'opération est la somme du temps de transmission, le temps d'accès à la mémoire et le temps d'accès au cache (en cas de lecture) (figure 4.10, étape 4). La difficulté ici est d'estimer

le temps de transmission qui n'est pas constant. Ce dernier dépend de la charge dynamique (en terme de transactions) sur le réseau d'interconnexion et des contentions possible pour accéder à une même cible.

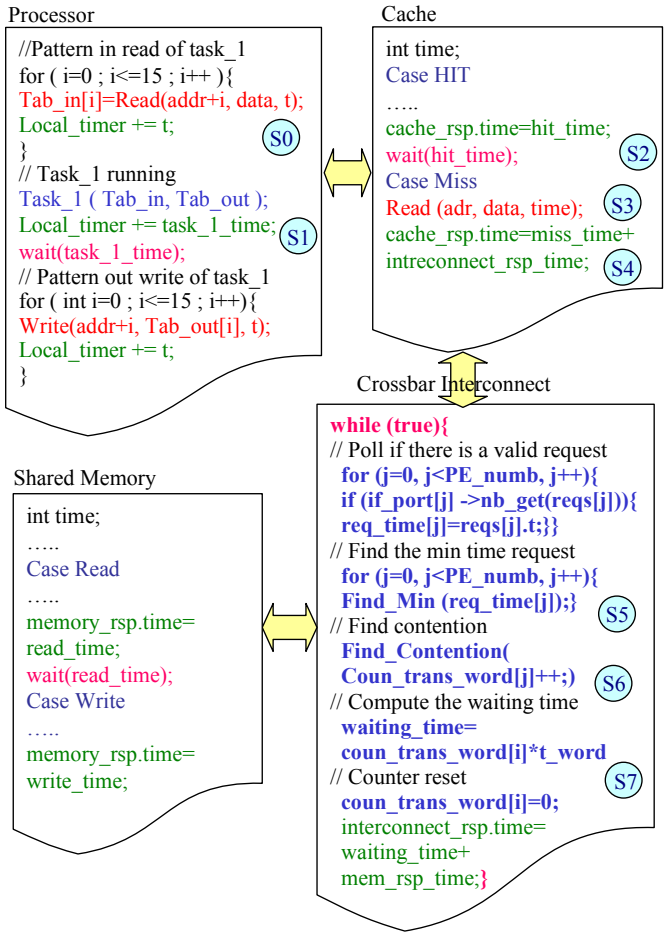


FIG. 4.10 – *Estimation de performance au sous-niveau PVT-PA*

Avant de transmettre une requête à la cible, l'arbitre correspondant parcourt l'ensemble des FIFO des différents canaux de communication. L'objectif est de comparer le paramètre *time* des requêtes présentes et de déterminer la plus prioritaire. Ce test permet aussi de localiser les contentions entre les requêtes. Pour évaluer le temps d'attente des requêtes au niveau du réseau d'interconnexion, un compteur est alloué à chaque entrée de FIFO dans le routeur. Lorsqu'une requête est mise en attente, son compteur (*coun_tran_word*) est incrémenté (figure 4.10, étape 5). Plus tard, lorsqu'une FIFO est sélectionnée, la valeur du compteur est lue puis multipliée par l'unité de temps (*t_word*) pour transmettre un seul mot (latence du réseau). Ce compteur est mis ensuite à zéro pour la requête suivante (figure 4.10, étape 6). Nous utilisons la même approche lors de la configuration du contrôleur DMA pour un transfert de données.

Dans notre modèle de temps, nous devons prendre en considération le cas où le processeur attend qu'un de ses motifs d'entrées soit prêt pour reprendre l'exécution. Pour estimer ce temps de repos, l'ordonnanceur implémenté au sous-niveau PVT-PA teste à chaque inter-

valle de temps si l'une de ses variables de synchronisation est à l'état valide. Si ce n'est pas le cas, alors le compteur local du processeur est incrémenté par un intervalle de temps fixé et nous exécutons une instruction *wait()* pour permettre au simulateur SystemC d'ordonnancer d'autres processeurs. Concernant l'intervalle de temps à choisir, un compromis doit être fait entre la précision et l'accélération de la simulation. En effet, si l'intervalle de temps est trop petit nous pouvons détecter plus rapidement les changements d'état du processeur. Néanmoins, un nombre plus important de changements de contexte entre les processeurs est nécessaire, ce qui va retarder la simulation. A l'opposé, si l'intervalle de temps est trop grand, l'instant de changement d'état du processeur ne sera pas détecté d'une façon précise. Dans notre cas, nous avons choisi un intervalle de temps représentatif du temps de lecture ou d'écriture d'un motif.

4.8.2 Estimation de performance dans PVT-TA

La différence entre le modèle du temps au sous-niveau PVT-PA et celui de PVT-TA concerne la partie calcul. En effet, au sous-niveau PVT-TA nous utilisons des ISS pour exécuter l'application. Pour cela nous avons identifié principalement le nombre et le type d'instructions exécutées comme activités pertinentes dans le composant processeur. Dans notre cas les délais d'exécution des instructions du processeur MIPS R3000 sont estimés à partir de la documentation technique donnée par [89]. Le compteur de temps local du processeur est incrémenté après l'exécution de chaque instruction (figure 4.11, étape 0).Ici *Op_add* correspond à l'exécution de l'instruction addition. Le compteur de temps local est incrémenté par le délai d'exécution de l'instruction. Pour estimer le temps d'une opération de lecture de donnée ou d'instruction à partir de la mémoire cache (en cas de succès) ou bien de la mémoire centrale (en cas de défaut de cache), la procédure est la même que celle décrite au sous niveau PVT-PA (figure 4.11, étape 1). Le cas particulier concernant l'estimation du temps lorsque le processeur est mis en attente ne se pose pas dans le cas du sous-niveau PVT-TA. En effet la synchronisation entre les tâches se fait à l'aide de variables déclarées au niveau de l'architecture. A l'état d'attente, le processeur fait des lectures successives sur ces variables et par conséquent le temps de repos est la somme des temps d'accès mémoires avant que celui-ci ne reprenne l'exécution d'une nouvelle tâche.

4.8.3 Estimation de performance dans PVT-EA

L'inconvénient du sous niveau PVT-TA c'est qu'il ne tient pas compte des délais temporels entre les événements. Dans le cas de défauts de cache pour deux processeurs différents, l'instant où un processeur effectue son accès mémoire correspondant peut influer sur l'instant d'accès de l'autre processeur en cas de collision. La figure 4.12.a montre un exemple d'erreur de détection de contention dans le réseau d'interconnexion à cause du non-respect des délais des événements (*Packet setup, Routing and Arbitration*). Lorsque le paquet transmis issu du processeur 1 arrive au niveau du routeur (R&A dans la figure 4.12.a), il n'y a aucune possibilité de détecter l'occupation du routeur par le traitement du paquet issu du processeur 0. En effet, les événements dans le sous-niveau PVT-TA sont exécutés instantanément (*zero delay*). Cette abstraction change le comportement de la partie communication dans le système, ce qui diminue la précision d'estimation de performance. Pour résoudre ce problème, nous avons amélioré le sous-niveau PVT-TA par l'introduction d'instructions de synchronisation. Elles prennent en considération les délais des activités des composants, les

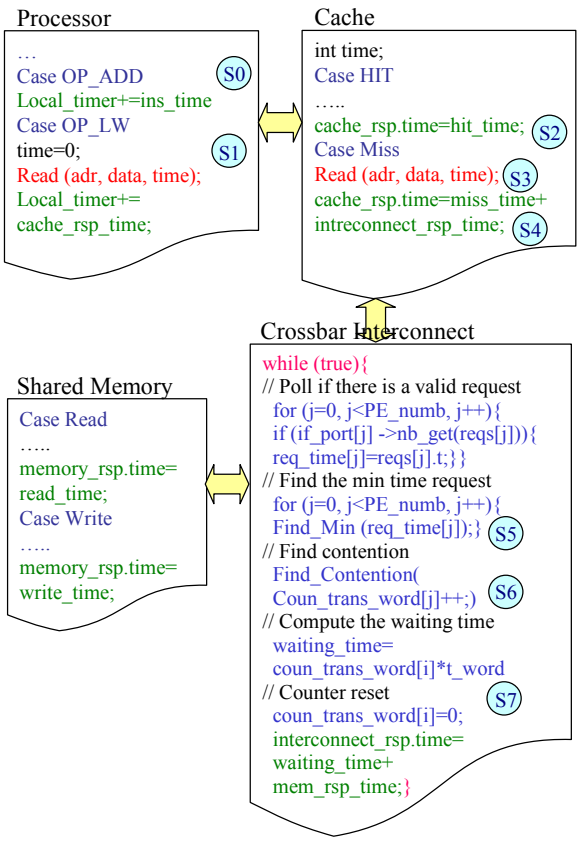
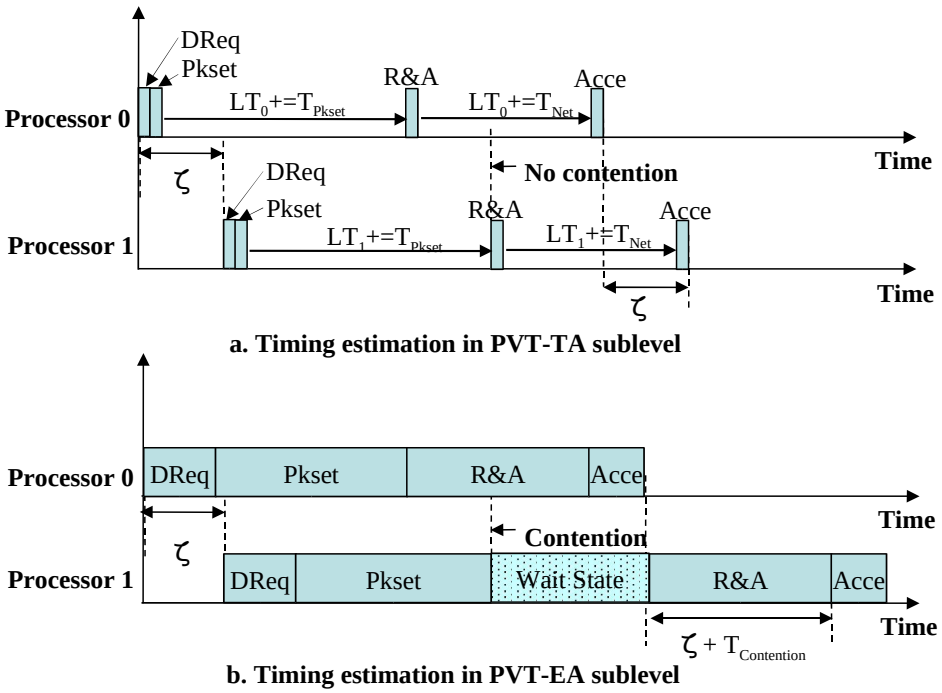


FIG. 4.11 – *Estimation de performance au sous-niveau PVT-TA*

délais de transmission des paquets et finalement le protocole de communication. Ce sont les caractéristiques du sous-niveau PVT-EA. Les événements à ce sous-niveau se produisent de la même façon qu'au niveau CABA (figure 4.12.b).

Pour pouvoir comparer l'erreur d'estimation entre le sous-niveau PVT-EA et le niveau CABA pour la plateforme SoCLib, il nous a fallu implémenter les spécifications du protocole VCI. Néanmoins, notre modèle d'estimation de performance est flexible et nous permet d'implémenter une variété de protocoles de communication. Pour émuler le même comportement du protocole VCI au sous-niveau PVT-EA et respecter le comportement des composants, des instructions *wait()* ont été ajoutées dans la description des composants ainsi qu'avant la transmission des requêtes commande et réponse. Les instructions *wait()* à ajouter nécessitent des arguments exprimés en unité de temps telle que nano seconde (ns) ou en nombre de cycles. Dans nos expérimentations, ces arguments sont mesurés depuis la plateforme CABA. Le tableau 4.2 présente les délais apportés au sous-niveau PVT-EA.



LT0: Local Timer 0; **LT1**: Local Timer 1; **DReq**: Data Request

Pkset: Packet Setup; **R&A**: Routing and Arbitration; **Acce**: Memory Access

T_{Pkset}: Packet Setup time; T_{Net}: Network time

FIG. 4.12 – *Estimation de temps dans les sous-niveaux PVT-TA et PVT-EA*

4.9 Simulation et résultats

4.9.1 L'environnement de simulation

Dans cette section nous allons évaluer la pertinence de notre environnement PVT proposé avec ces 3 sous-niveaux en vue d'une exploration d'architecture pour les systèmes MPSoC. Plusieurs expérimentations ont été menées. L'objectif était de mesurer pour chaque sous-niveau décrit précédemment l'accélération de la simulation, la précision de prédiction de performance et l'effort de développement nécessaire pour modéliser les composants du système MPSoC. Ces métriques sont rapportées en comparaison avec le niveau CABA et en faisant varier la taille du cache de données et d'instructions et le nombre de processeurs. Il est intéressant de noter que cet environnement peut être utilisé pour accomplir une exploration architecturale en fonction d'autres paramètres (exemple : type de processeur, type de réseau d'interconnexion, etc.) dans un intervalle de temps raisonnable et une précision acceptable. La figure 4.13 montre l'architecture du système MPSoC utilisée dans les expérimentations. Nous avons parallélisé et testé plusieurs applications de traitement de données intensif : la multiplication de matrices, le downscaler [75], une version logicielle de la TCD et le codeur H.263 [37]. Ces applications sont parallélisées en partageant le calcul d'une façon équitable entre les différents processeurs. Par exemple, la TCD est parallélisée en attribuant différents

Activités	Délais
Préparation d'une requête de commande VCI	4 cycles
Préparation d'une requête de réponse VCI	4 cycles
Exécution d'une instruction	1 cycle
Succès de cache	1 cycle
Défaut de cache	12 cycles + cycles dus aux contentions sur le réseau
accès mémoire en lecture	2 cycles
accès mémoire en écriture	2 cycles
accès au DMAC	2 cycles
TCD 1-D appliquée à un bloc 8x8	12 cycles

TAB. 4.2 – *Délais des activités utilisés dans les expérimentations*

segments d'image pour chaque processeur.

Le facteur d'accélération de la simulation est défini comme suit :

$$Speed\_fact = PVT\ Simul.time / CABA\ Simul.time$$

Dans cette expression, PVT correspond au sous-niveau PVT-PA, PVT-TA ou PVT-EA. Dans toutes les expérimentations, une machine équipée d'un processeur Pentium M (1.6 GHz) et 1GB de mémoire est utilisée. Nous notons que les tests au niveau CABA exigent plusieurs heures de simulation et dépendent de la configuration du système simulé. L'application H.263 est choisie pour illustrer nos expérimentations, elle sera détaillée dans la sous-section suivante.

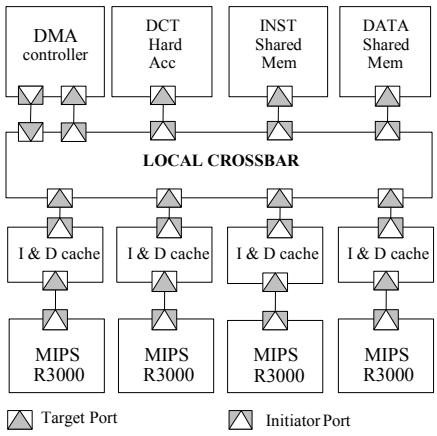


FIG. 4.13 – *Exemple de structure MPSoC*

4.9.2 Le codeur H.263

Le standard H.263 de codage a été développé pour la transmission de la vidéo sur des lignes à très bas débits pour les applications de visiophonie, les systèmes de surveillance sans fil, etc. Dans nos expérimentations, nous nous intéressons à l'implémentation du codeur

H.263. Cette application est composée de trois tâches ; la Transformée en Cosinus Discrète (TCD), la quantification (Quant) et le codage (VLC). La TCD permet d'éliminer la redondance de données et de transformer les données du domaine spatial en une représentation fréquentielle. La tâche de quantification consiste à diviser chaque coefficient de la TCD par un pas de quantification et mettre les coefficients non significatifs à zéro. La tâche de codage permet d'encoder les coefficients TCD quantifiés en attribuant aux différents symboles (luminance, chrominance) un mot binaire. La méthode de codage utilisée est celle de Huffman [59].

Les données manipulées sont structurées sous forme de macroblocs qui représentent un espace de 16x16 pixels d'une image vidéo. Le format de données du macrobloc est le YCbCr qui contient trois composantes : luminance (Y), chrominance bleue (Cb) et chrominance rouge (Cr). Les blocs de luminance décrivent l'intensité des pixels tant que les blocs de chrominance contiennent des informations sur les couleurs des pixels. Un macrobloc contient 6 blocs de 8x8 : 4 blocs contiennent les valeurs de la luminance, un bloc contient les valeurs de chrominance bleue et un bloc contient les valeurs de chrominance rouge. Cette application est ici parallélisée pour qu'elle puisse s'exécuter en utilisant un système MPSoC formé de 4 à 16 processeurs. La tâche TCD est affectée à l'accélérateur matériel TCD-2D. Les deux tâches quantification et codage sont partagées équitablement entre les processeurs en allouant différents macroblocs à chaque processeur. Au cours des simulations, le codeur H.263 est appliqué à la séquence vidéo Foreman.qcif (figure 4.14).

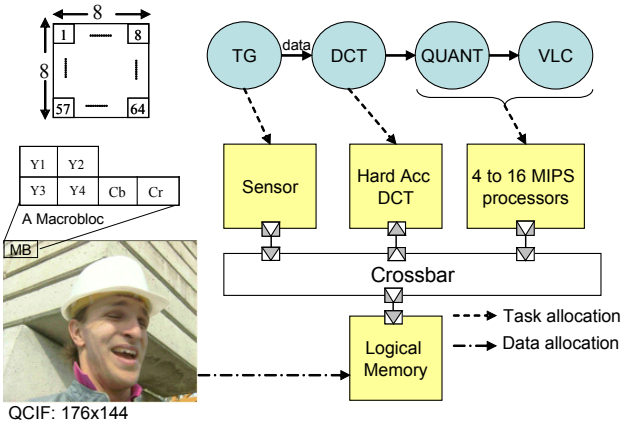


FIG. 4.14 – *L'application codeur H.263*

4.9.3 Résultats de simulation au sous-niveau PVT-PA

Au sous niveau PVT-PA, nous avons identifié les différentes tâches de l'application H.263, la spécification du domaine d'itération ainsi que les synchronisations nécessaires entre les différentes tâches. Nous avons exécuté notre application en faisant varier le nombre de processeurs entre 4 et 16 pour évaluer le facteur d'accélération et la précision d'estimation. Comme premier résultat de simulation, nous avons vérifié notre hypothèse concernant le flux d'accès à la mémoire d'instructions qui est négligeable devant le flux d'accès à la mémoire de données. En CABA nous observons que les instructions ne représentent que 0.7%

du nombre d'accès total.

La figure 4.15 nous montre les résultats de simulation au sous-niveau PVT-PA en faisant varier le nombre de processeurs. Nous observons que l'augmentation du nombre de processeurs entraîne une amélioration du facteur d'accélération qui peut atteindre la valeur 32 avec un système de 16 processeurs. Une analyse précise de la trace produite par le simulateur SystemC nous montre que 25% du temps de simulation concerne l'exécution de la fonction du réseau d'interconnexion alors que le temps de simulation de la partie calcul est presque nul. En particulier, l'utilisation d'un nombre assez important de processeurs crée plus de contentions sur le réseau d'interconnexion lors des accès simultanés aux ressources partagées. PVT-PA est efficace pour la modélisation d'applications nécessitant beaucoup de communications. Malgré sa performance en terme d'accélération, le sous-niveau PVT-PA comporte une erreur d'estimation de performance raisonnable qui varie en fonction du nombre de processeurs et qui ne dépasse pas 27% (figure 4.15). Cette erreur est due principalement à deux causes. La première vient du fait que nous avons négligé les accès mémoires qui correspondent aux variables locales des fonctions et des variables de synchronisation. Ce type d'accès peut présenter jusqu'à 10% de l'accès total aux ressources partagées. La deuxième cause correspond à l'erreur de détection des contentions dans le réseau d'interconnexion. En effet, les transferts de motifs dans le sous-niveau PVT-PA sont une approximation des transferts réels observables au niveau CABA.

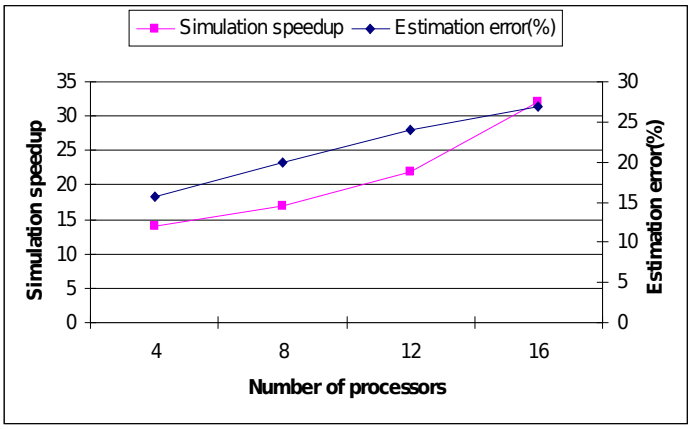


FIG. 4.15 – *Estimation de performance et de l'erreur d'estimation au sous-niveau PVT-PA*

Jusqu'à présent, nous nous sommes focalisés sur l'utilité du sous-niveau PVT-PA pour accomplir une validation fonctionnelle du système MPSoC avec des facteurs intéressants d'accélération de la simulation. Ce sous-niveau peut aussi être utilisé pour l'observation des contentions au cours de l'exécution de l'application. Il est possible de détecter les goulots d'étranglement dans le réseau d'interconnexion. La figure 4.16 présente le nombre de contentions cumulées chaque 1000 cycles au cours de l'exécution. Ce résultat est obtenu avec un système de 4 processeurs. Les allures des deux courbes PVT-PA et CABA sont proches l'une de l'autre. Les transferts de données au sous-niveau PVT-PA se font de façon régulière avant et après l'exécution de la tâche contrairement à ce qui se passe réellement en bas niveau où les transferts de données se font au cours de l'exécution de la tâche. Pour cela le nombre de contentions au sous-niveau PVT-PA est présenté sous forme d'une valeur moyenne par rapport aux contentions au niveau CABA. Ce qui est intéressant à partir de cette figure 4.16

c'est que nous pouvons détecter aussi facilement les goulots d'étranglement dans le réseau d'interconnexion au niveau PVT-PA qu'au niveau CABA.

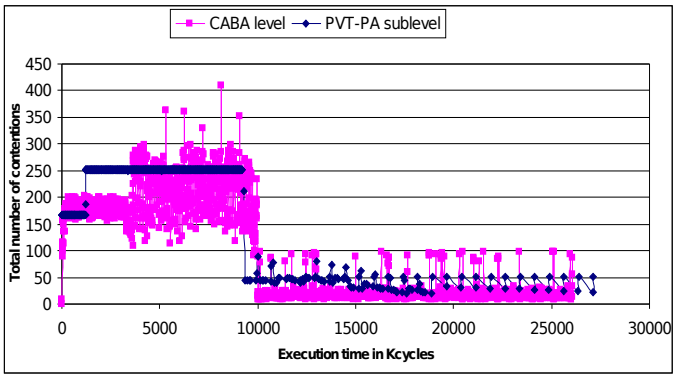


FIG. 4.16 – *Contentions dans le réseau d'interconnexion dans PVT-PA and CABA*

4.9.4 Résultats de simulation au sous-niveau PVT-TA

Pour réduire l'erreur d'estimation de performance au sous-niveau PVT-PA, nous allons utiliser des ISS pour exécuter l'application. Les tâches de l'application H.263 sont compilées vers notre processeur MIPS R3000. Notre étude à ce sous-niveau va être plus détaillée qu'au paragraphe précédent puisque nous envisageons une estimation assez précise de performance et ensuite de la consommation d'énergie (chapitre suivant). Une analyse par trace de l'exécution de l'application montre que l'introduction des ISS des processeurs dans le système MPSoC produit un ralentissement de la simulation de la partie calcul par un facteur de 30. Néanmoins, cette partie ne présente que 6% de la charge totale de la simulation et la partie communication reste prédominante pour le temps de simulation.

Pour évaluer l'impact de la taille des caches d'instructions et de données sur le facteur d'accélération de la simulation, nous avons exécuté plusieurs applications au sous-niveau PVT-TA en utilisant un système MPSoC de 4 processeurs. La taille des caches varie de 1 Koctets à 32 Koctets (Ko) en adoptant la configuration suivante : l'associativité=1, la taille du bloc= 32octets et la stratégie d'écriture est "Write-Through". La figure 4.17 montre que la réduction de la taille des caches diminue les temps de simulation. En effet, les caches de taille réduite amplifient le nombre de défauts et par conséquent le trafic sur le réseau d'interconnexion. Nous notons aussi un écart important dans l'accélération lorsque le noyau de l'application et les données correspondantes ne peuvent pas être totalement stockés dans les caches. Dans notre exemple, un écart important dans l'accélération est obtenu à partir des tailles de caches inférieures à 16 Ko pour l'application H.263. Pour la version software de la TCD, cet écart important est détecté à partir des tailles de caches inférieures à 2 Ko.

Pour évaluer l'impact du nombre de processeurs sur le facteur d'accélération de la simulation, nous avons exécuté l'application H.263 au sous-niveau PVT-TA avec un système MPSoC formé de 4 à 16 processeurs. La figure 4.18 montre qu'avec le sous-niveau PVT-TA, il est possible d'accélérer la simulation avec un facteur qui peut atteindre 18. L'addition de nouveaux processeurs au système augmente le facteur d'accélération, qui s'explique par l'amplification de la communication entre les processeurs et les modules mémoires partagés.

Le sous-niveau PVT-TA comporte une faible erreur d'estimation de performance.

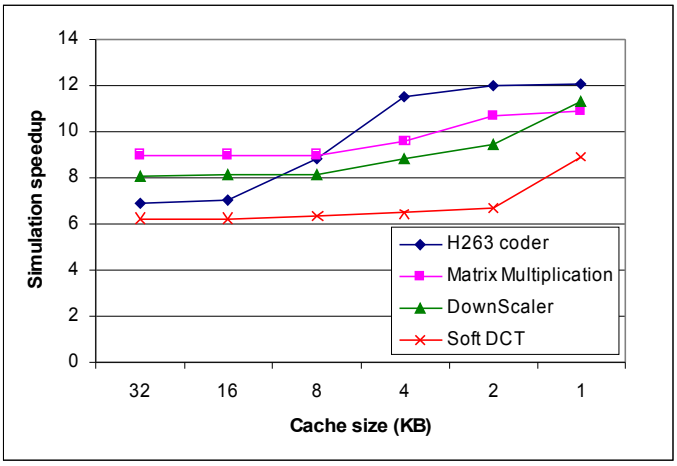


FIG. 4.17 – *Accélération de la simulation pour différentes applications*

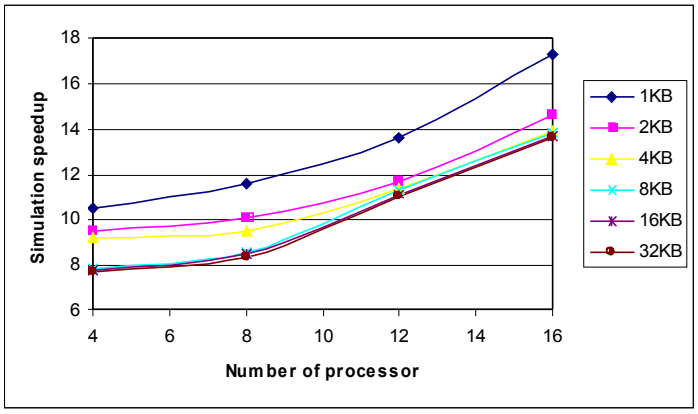


FIG. 4.18 – *Accélération de la simulation dans le sous-niveau PVT-TA*

Comme le montre la figure 4.19, l'augmentation du nombre de processeurs ou l'utilisation des caches de petite taille accentue les communications et l'erreur d'estimation. Cette erreur peut atteindre au maximum 7.2% avec un système de 16 processeurs et 1Ko de taille de cache. Toujours pareil, plus il y a de communication, plus il y a d'erreurs de détection des contentions.

4.9.5 Résultats de simulation au sous-niveau PVT-EA

Plusieurs expérimentations ont été menées en utilisant la même application et les configurations du système MPSoC pour évaluer le sous-niveau PVT-EA. Dans le modèle de temps, nous intégrons les spécifications du protocole VCI ainsi que les délais entre les événements. L'erreur de précision avec PVT-EA est réduite presque à zéro pour toutes les configurations testées. Nous pensons que l'utilisation d'un processeur d'architecture simple avec une exécution dans l'ordre (*in-order*) est parfaitement adaptée à notre niveau de simulation et permet ainsi de minimiser l'erreur d'estimation. Par rapport au sous-niveau PVT-TA, le sous-niveau PVT-EA ralentit la simulation de 30% (figure 4.20) [8].

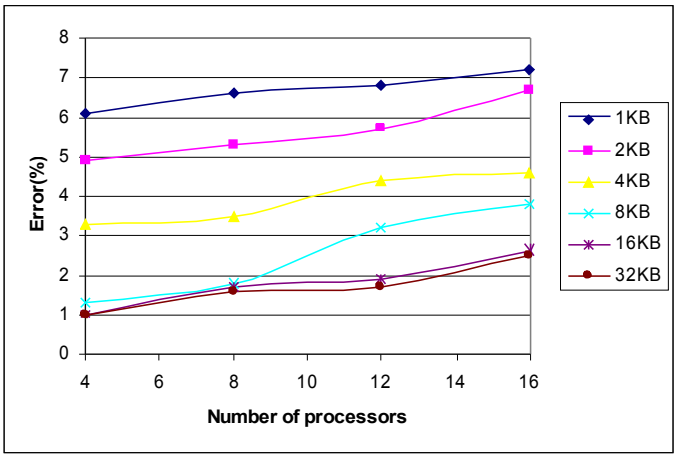


FIG. 4.19 – *Erreur d'estimation dans le sous-niveau PVT-TA*

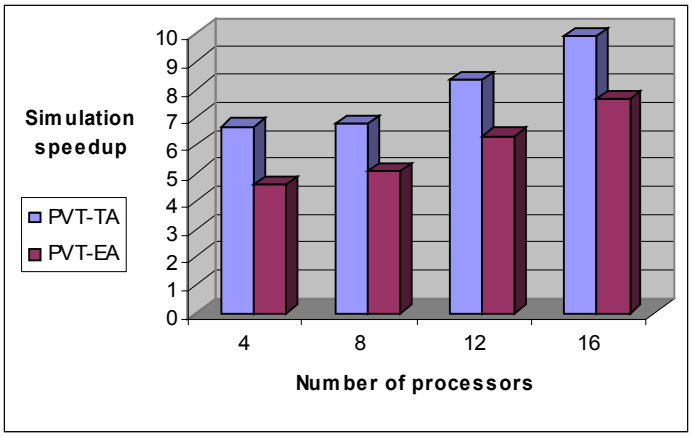


FIG. 4.20 – *Comparaison de l'accélération entre PVT-TA et PVT-EA*

4.9.6 Effort de modélisation

Jusqu'à maintenant nous avons montré l'utilité de notre approche en terme d'accélération de la simulation et en terme d'estimation de performance. Néanmoins, cette approche a prouvé son efficacité aussi en terme d'effort de modélisation. Elle permet aux concepteurs le développement et la validation des systèmes MPSoC dans un temps réduit. Le tableau 4.3 présente l'effort de modélisation exprimé en terme de lignes de code (*LOC :lines of code*) nécessaires pour la conception d'un système MPSoC dans les niveaux CABA et PVT (PVT-PA, PVT-TA et PVT-EA). Selon les résultats, l'effort de modélisation avec PVT-PA, PVT-TA et PVT-EA est réduit respectivement d'un facteur de 68.53%, 44.04% et 37.23%. L'utilisation d'une stratégie de simulation multi-niveaux (par objectifs) permet rapidement de focaliser sur un sous-ensemble de systèmes MPSoC sans avoir à multiplier les efforts de modélisation pour chaque niveau d'abstraction [3].

Abstract level	Modeling effort(LOC)							Reduction (%)
	Processor	Cache	Interconnect	Memory	DMA	TCD	Total	
CABA	1578	553	399	312	650	300	3792	
PVT-EA	1486	244	177	183	115	175	2380	37.23
PVT-TA	1259	238	170	180	105	170	2122	44.04
PVT-PA	330	238	170	180	105	170	1193	68.53

TAB. 4.3 – *Comparaison de l'effort de modélisation*

4.10 Synthèse et généralisation

Au cours de ce chapitre, nous avons présenté un environnement décrit au niveau PVT pour simuler des systèmes MPSoC. Cet environnement se décline en 3 sous-niveaux PVT-PA, PVT-TA, et PVT-EA. A chaque sous-niveau d'abstraction, un modèle d'estimation de performance a été développé pour évaluer le temps d'exécution d'une application donnée. En utilisant ces trois sous-niveaux, plusieurs objectifs ont été atteints :

- Les trois sous-niveaux permettent une vérification fonctionnelle du système MPSoC. Néanmoins, c'est PVT-PA qui offre la vérification la plus rapide en comparaison avec le niveau CABA.
- Les trois sous-niveaux intègrent un modèle d'estimation de performance et offrent différents compromis accélération/précision. PVT-TA présente le meilleur compromis.
- Avec le sous-niveau PVT-PA, nous avons pu visualiser les contentions sur le réseau d'interconnexion. Ceci constituera un outil pour vérifier l'adéquation du support de communication avec le système et pour détecter les goulots d'étranglement vers une cible particulière.
- Avec le sous-niveau PVT-EA, les spécifications du protocole de communication ont été prises en considération au plus tôt dans le processus de développement.
- En utilisant les 3 sous-niveaux, nous avons pu réduire considérablement l'effort de développement par rapport au niveau CABA.

Pour faciliter aux concepteurs la simulation et l'évaluation de performance des systèmes MPSoC en utilisation les 3 sous-niveaux développés, nous allons intégrer les différents composants matériels conçus dans notre environnement de conception Gaspard. En plus, nous allons définir les concepts nécessaires permettant de prendre en considération le sous-niveau choisi et les informations temporelles correspondantes comme il sera décrit dans le chapitre 6.

Les composants matériels seront utilisés depuis notre bibliothèque au niveau PVT. Pour intégrer d'autres composants décrits au niveau transactionnel dans notre bibliothèque, le concepteur doit adapter les méthodes de communication à notre définition (*read(adr, data, time)* et *write(adr, data, time)*). Dans notre proposition, nos modèles d'estimation de performance sont fortement liés à la description des composants. Néanmoins, nous pouvons généraliser notre approche. Pour les initiateurs qui exécutent les tâches, nous définissons un compteur de temps local pour chacun afin d'estimer le temps d'exécution de la tâche correspondante. Pour les autres composants qui jouent le rôle d'une cible ou d'un intermédiaire de communication, la requête est récupérée et le paramètre *time* est lu. Après l'exécution de sa fonctionnalité, chaque composant ajoute les délais appropriés au paramètre *time* de la requête. Ainsi le temps de transmission totale est calculé. Ce paramètre sera récupéré par

l'initiateur pour incrémenter son compteur local.

Comme ceci a été décrit au niveau CABA, nous visons avec notre environnement Gaspard la génération automatique du code SystemC au niveau PVT à partir d'une description de haut niveau du système MPSoC. Le même ensemble de transformations décrit au niveau CABA sera réutilisé pour les sous-niveaux PVT-PA, PVT-TA et PVT-EA. Ceci est considéré parmi les avantages de la méthodologie de conception dirigée par les modèles. La distinction entre les spécifications de chaque sous-niveau fera l'objet de la phase de déploiement qui sera détaillée dans le chapitre 6.

4.11 Conclusion

Pour réduire la complexité de développement et accélérer l'exploration de l'espace de solutions architecturales, il s'avère indispensable de modéliser les systèmes MPSoC dans les premières phases du flot de conception. Notre contribution dans ce chapitre consiste à décrire les systèmes au niveau transactionnel PVT. A ce niveau, nous avons proposé un environnement décrit avec 3 sous-niveaux PVT-PA, PVT-TA et PVT-EA permettant l'accélération de la simulation des systèmes MPSoC. Différents composants matériels ont été conçus pour implémenter les trois sous-niveaux. Afin d'obtenir une prédiction précise du temps d'exécution dans notre environnement, nous avons enrichi les trois sous-niveaux par des modèles de temps permettant une erreur d'estimation relative à la précision de description du système. Les résultats de simulation montrent une complémentarité entre les trois sous-niveaux PVT-PA, PVT-TA et PVT-EA qui offrent différents compromis accélération/précision.

Dans notre travail, nous nous sommes limités à des systèmes MPSoC à mémoires partagées contenant des architectures de processeurs simples tels que le MIPS R3000. Il sera intéressant d'étudier notre approche d'accélération pour des systèmes MPSoC plus complexes tels que des architectures à mémoires distribuées. Dans les perspectives nous proposons d'étudier le comportement de processeurs plus complexes tels que des topologies superscalaire ou VLIW (*Very Large Instruction Word*). Nous nous sommes aussi intéressés dans ce travail aux applications de traitement de données intensif qui ne nécessitent pas généralement l'utilisation d'un système d'exploitation (SE). La prise en considération du comportement du SE lors de la simulation logicielle/matérielle des systèmes MPSoC ainsi que les aspects de temps correspondants reste des problèmes difficiles qui dépassent les objectifs de cette thèse.

Certes, l'estimation de performance est un critère efficace pour comparer les différentes alternatives d'un espace de solutions architecturales. Aujourd'hui dans les systèmes embarqués, il est indispensable de prendre en compte la consommation d'énergie comme un critère de développement au même titre que la vitesse et la surface. Une exploration fiable des systèmes MPSoC nécessite de définir des outils d'estimation de consommation d'énergie à différents niveaux d'abstraction. Ce défi fait l'objectif de notre prochain chapitre.

Chapitre 5

Estimation de la consommation d'énergie dans les MPSoC

5.1 Introduction

Dans le chapitre précédent, nous avons montré l'intérêt de la modélisation au niveau transactionnel (PVT) pour l'accélération de la simulation des MPSoC. Dans ce dernier chapitre, nous nous sommes basés uniquement sur le critère de performance pour évaluer les différentes alternatives de l'espace de solutions architecturales. Du fait de l'augmentation de la densité d'intégration des circuits dans les SoC, il est indispensable de prendre en compte la consommation comme critère de conception d'un système au même titre que la vitesse. En effet, avec des fréquences d'horloge de plus en plus élevées, une quantité de ressources matérielles de plus en plus grande et un degré d'intégration de plus en plus important, les circuits se rapprochent des limites physiques et thermiques supportables. Par conséquent, l'exploration d'architectures dans le flot de conception des systèmes sur puce nécessite des outils capables d'évaluer la consommation d'énergie à différents niveaux d'abstraction. Ceci permet de prendre des décisions à chaque niveau pour réduire l'espace des solutions architecturales. Dans ce chapitre, l'objectif que nous nous sommes fixés est de concevoir des outils permettant une estimation de la consommation statique et dynamique de l'énergie. Ces outils sont destinés pour être utilisables dans les environnements de conception de MPSoC aux niveaux CABA et PVT.

Comme toutes les unités de la plateforme (processeur, hiérarchie mémoire, et réseau d'interconnexion) contribuent dans cette consommation, il devient important dans un souci de précision d'estimer la contribution de chaque unité. L'un des objectifs de notre travail est donc, de développer une méthodologie permettant la conception de modèles de consommation pour ces composants. Dans cette méthodologie nous tenterons de regrouper à la fois les critères de précision et de flexibilité.

Le mot "consommation" est souvent utilisé pour désigner à la fois la dissipation de puissance et la dissipation de l'énergie. La consommation de puissance est un paramètre important si l'on s'intéresse à la dissipation thermique dans un système. En effet, pour pouvoir dimensionner les circuits de refroidissement d'un système, il est nécessaire de connaître avec précision la puissance dissipée par ce système. A l'opposé, l'énergie est un paramètre à prendre en compte si l'on veut étudier la durée de vie des batteries. Ce paramètre est, bien sûr, lié à la puissance consommée (P) par l'application mais également au temps d'exécution (T) de l'application. Ainsi, l'énergie est calculée par ($E = P \; x \; T$) [70]. Dans ce document, nous utilisons le terme consommation dans le contexte des systèmes embarqués pour indiquer la dissipation d'énergie.

Ce chapitre est structuré comme suit. La deuxième et la troisième sections traitent l'estimation de la consommation d'énergie respectivement aux niveaux CABA et PVT. Notre méthodologie pour le développement des modèles de consommation à différents niveaux d'abstraction est présentée dans la section 4. La section 5 de ce chapitre détaille la mise en œuvre de notre méthodologie. Dans cette section, plusieurs modèles de consommation pour les composants, tels que le processeur, les mémoires, le réseau d'interconnexion crossbar, et l'accélérateur matériel TCD-2D, sont présentés. La section 6 présente les résultats de simulation et une comparaison de la précision de l'estimation de la consommation aux niveaux PVT et CABA. La réutilisation de l'environnement pour réaliser une exploration architecturale est aussi présentée dans cette section. Enfin, un aperçu sur l'intégration des modèles de consommation dans notre environnement Gaspard est donné dans la section 7.

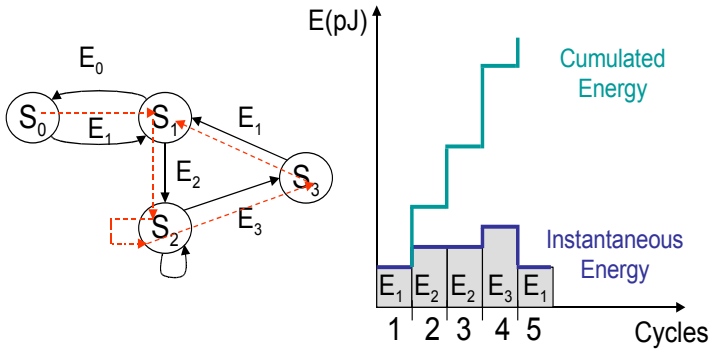


FIG. 5.1 – *Automate d'états et estimation de la consommation d'un composant x*

5.2 Estimation de la consommation d'énergie au niveau CABA

L'évaluation des systèmes MPSoC nécessite des outils permettant une estimation précise de la consommation au niveau de la micro-architecture. Ces outils seront utiles pour réaliser la comparaison des différentes solutions dans une exploration architecturale. Ils doivent être flexibles et modulaires pour découpler la partie traitement de la partie communication et pour permettre une exploration locale au niveau d'un composant. Dans ce dernier cas, il s'agit de trouver les paramètres les plus adéquats de la micro-architecture d'un composant. L'estimation à ce niveau doit permettre, par ailleurs, d'analyser un système tout entier dans un temps raisonnable.

Dans ce paragraphe nous détaillons la méthodologie d'estimation de la consommation au niveau architectural en se basant sur une simulation au cycle précis dans les systèmes multiprocesseurs. La consommation d'énergie d'une application exécutée sur une plate-forme est égale à la somme des consommations dans les différents composants. Dans le chapitre 3, nous avons détaillé la description (sous forme de FSM) des composants au niveau CABA. Dans ces FSM, à chaque cycle une transition est réalisée, même s'il n'y a pas de changement d'état. De ces transitions résulte une consommation due à l'exécution des opérations associées au nouvel état. Ces opérations peuvent être de types : lecture, écriture, attente, calcul, etc. Une estimation suffisamment précise de la consommation revient par conséquent à identifier l'énergie dissipée par transition dans chaque FSM. La figure 5.1 illustre la méthodologie d'estimation de la consommation au niveau CABA pour un composant donné. Notre composant x est décrit avec une FSM à 4 états. A chaque transition vers un de ces états, nous associons un coût énergétique qui représente la consommation du composant à un cycle donné. Une autre approche consiste à accumuler les consommations de chaque cycle durant la simulation afin de calculer l'énergie totale.

Comme nous nous intéressons à la consommation dans chaque composant du MPSoC, l'objectif est de développer les modèles de consommation correspondants aux unités et les intégrer dans un simulateur qui prend en compte les paramètres architecturaux et les paramètres technologiques. Cette approche offre plusieurs avantages :

- Localiser l'unité fonctionnelle qui consomme le plus d'énergie. Ce qui permet au concepteur de proposer une nouvelle architecture de cette unité pour réduire ainsi l'énergie consommée.
- Evaluer le coût en consommation d'énergie de plusieurs alternatives architecturales

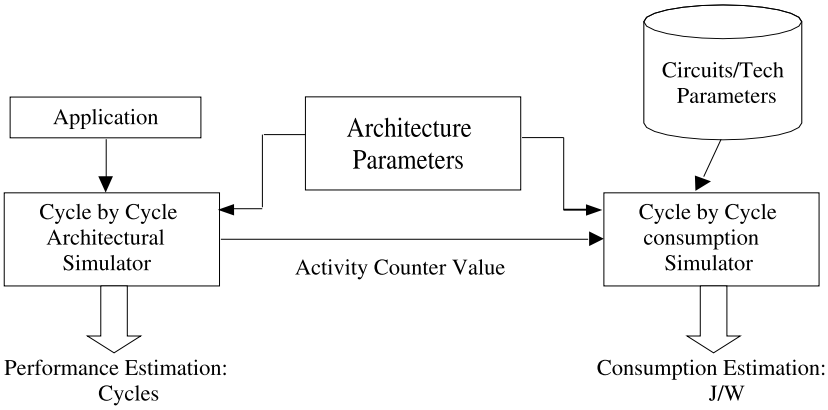


FIG. 5.2 – *Estimation de la consommation au niveau CABA*

afin d'implémenter au niveau physique celle la plus prometteuse.

Deux facteurs essentiels interviennent dans la détermination de la précision d'estimation d'énergie au niveau CABA. Le premier facteur est lié à la description du composant. En effet, il faut réussir à identifier les opérations qui sont exécutées au cours de chaque transition des FSM décrivant le composant et pouvoir les comptabiliser. L'exécution de ces opérations est toujours accompagnée par un changement dans l'état interne du composant, ce que nous appelons une activité. Chaque activité est considérée comme une source de consommation d'énergie à laquelle un compteur est associé. Plusieurs activités peuvent s'additionner pour définir la consommation totale d'un état de la FSM. Le deuxième facteur essentiel pour garantir la précision de l'estimation d'énergie est lié à l'évaluation du coût énergétique de chaque activité. Cette phase sera détaillée dans la section 5.4.

Dans notre travail, nous avons identifié les activités pertinentes dans chaque composant et un compteur a été attribué à chaque activité du composant. Au cours de la simulation, le compteur est incrémenté si l'état correspondant est valide. A la fin de la simulation, par le biais de ces compteurs, le nombre d'occurrences de chaque activité est obtenu. En plus des compteurs, un coût énergétique, calculé à partir des modèles que nous allons développer, est attribué à chaque activité. Cette phase sera détaillée pour chaque composant dans la section 5.5.

Ainsi, la consommation totale de l'application est déterminée à partir de l'équation 5.1 suivante :

$$E = \sum_{i} N_i \times C_i \tag{5.1}$$

N_i : Nombre de fois où l'activité i est réalisée
C_i : Coût d'une unité de l'activité i

La figure 5.2 détaille notre stratégie d'estimation. Les paramètres de l'architecture (nombre de processeurs, taille du cache, etc.) sont spécifiés avant la simulation. Au cours de l'exécution de l'application, les valeurs des compteurs sont transmises au simulateur de la consommation. Ces valeurs permettent de calculer la dissipation d'énergie par cycle ou la dissipation totale à la fin de la simulation. Le simulateur de consommation contient un modèle d'énergie pour chaque composant qui dépend des paramètres de l'architecture et des paramètres technologiques. Cette approche sera appliquée sur notre plateforme au niveau CABA présentée dans le chapitre 3.

5.3 Estimation de la consommation d'énergie au niveau PVT

Au niveau PVT, plusieurs obstacles rendent difficile l'opération d'estimation de la consommation d'énergie. Le premier obstacle concerne les détails de la micro-architecture qui sont absents dans ce niveau de la modélisation, ce qui rend impossible la récupération de certains types d'activités. Le deuxième obstacle est dû à l'absence de l'horloge dans la description au niveau PVT. En effet, la notion de cycle n'existe plus et les activités s'exécutent simultanément de façon atomique comme les transactions. Face à ces défis, nous proposons une stratégie d'estimation de la consommation qui se base sur la récupération des informations de bas niveau afin de les combiner et de produire une estimation dans le niveau supérieur, représenté ici par le niveau transactionnel. D'une façon similaire au niveau CABA, l'estimation de la consommation au niveau PVT, revient à estimer la dissipation dans chaque composant de notre MPSoC. La formule de l'équation 5.1 est aussi valide au niveau PVT.

Néanmoins, la granularité des activités pertinentes est plus grossière que celle au niveau CABA. En général, pour pouvoir estimer la consommation d'énergie à un niveau de description donné, il est nécessaire d'utiliser les activités présentes dans ce niveau. A titre d'exemple, au niveau CABA, il est envisageable de récupérer les occurrences des activités dont la granularité correspond à la micro-architecture. A l'opposé, pour estimer la consommation au niveau RTL, la connaissance des valeurs des compteurs de ces activités n'est pas suffisante puisque la granularité se situe cette fois-ci au niveau des éléments séquentiels (bascules, registre, etc.).

Dans notre stratégie nous nous sommes basés sur les modèles de consommation du niveau CABA pour déduire ceux du niveau PVT et obtenir ainsi le maximum de précision. Les différents modèles de consommation pour les composants du MPSoC au niveau PVT ont été intégrés dans un simulateur de consommation qui interagira avec le simulateur d'architecture. La figure 5.3 détaille notre stratégie d'estimation au niveau PVT. Nous pouvons remarquer que l'approche présentée ici est assez similaire à ce qui a été proposé au niveau CABA. Les deux points de différence concernent la granularité des activités pertinentes et l'intervalle de temps entre deux évaluations possibles de la consommation. En effet, au niveau CABA la consommation peut être estimée à chaque cycle, au niveau PVT ceci n'est pas possible. Les activités s'exécutent de façon atomique et nécessitent donc plusieurs cycles avant d'être terminées. Ce qui ne permet de connaître la consommation qu'à la fin des actions de gros-grain.

5.4 Méthodologie pour le développement des modèles de consommation

Dans cette section nous décrivons la méthodologie de développement des modèles de consommation pour les différents composants permettant de construire un système multiprocesseur aux niveaux CABA et PVT. Comme ceci a été précisé dans l'introduction de ce chapitre, notre méthodologie pour l'évaluation de la consommation est réalisée en deux étapes :

1. Localiser les activités pertinentes en terme de consommation d'énergie et associer à chaque activité un compteur afin de connaître le nombre d'occurrences correspondant lors de l'exécution de l'application.

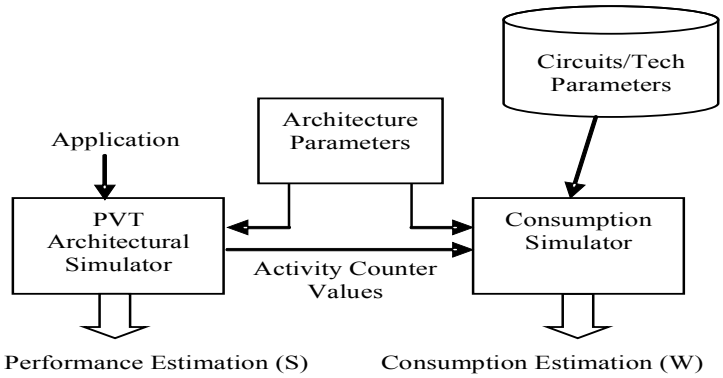


FIG. 5.3 – *Estimation de la consommation au niveau PVT*

2. Evaluer le coût énergétique élémentaire de chaque activité pertinente.

L'évaluation des coûts énergétiques des activités pertinentes dépend de l'architecture du composant et des paramètres technologiques. La consommation dans chaque composant est liée à 2 sources : la consommation dynamique, qui correspond aux changements de l'état interne du circuit (activités) et la consommation statique, qui est due principalement aux courants de fuite dans les transistors. Dans le passé, la dissipation dynamique était considérée plus importante que la dissipation statique. Avec les nouvelles technologies submicroniques, la situation a changé et les deux sources de dissipation ont le même ordre de grandeur. La précision des modèles de consommation analytiques existants dans la littérature et qui ne prennent pas en compte les sources de dissipation statique, est de ce fait remise en cause.

Pour cela, la solution qui offre le maximum de précision pour estimer la consommation est d'avoir physiquement le composant, de faire des tests et de réaliser des mesures directes sur le circuit. Cette approche nécessite néanmoins du temps et des outils de conception. En effet avec cette approche, il faut manipuler des outils de modélisation de bas niveau (au-delà de RTL).

Dans notre travail pour satisfaire au critère de précision, nous avons utilisé des simulations de bas niveau lorsque les outils pour ces simulations étaient disponibles et lorsque le composant se prête facilement à ces mesures. L'évaluation des coûts énergétiques dans les niveaux les plus bas et leur utilisation pour réaliser des estimations dans les niveaux supérieurs (CABA et PVT) a comme résultat une estimation hybride qui peut offrir un rapport précision/délai intéressant.

Néanmoins, cette évaluation avec des outils de CAO de bas niveau n'est pas réalisable pour tous les composants. En effet, cette approche n'est par exemple pas réalisable pour le réseau d'interconnexion crossbar. Ce dernier n'est pas un composant préexistant comme la mémoire ou le processeur. Rappelons, qu'un crossbar sert à connecter plusieurs initiateurs à différentes cibles, de ce fait la longueur de connexion dépend de l'emplacement de l'initiateur et de la cible sur la puce. Dans ce cas, il est nécessaire d'utiliser une approche analytique pour estimer le coût énergétique de l'activité.

Notre méthodologie de développement des modèles de consommation aux niveaux CABA et PVT est présentée comme suit [1] (voir la figure 5.4) :

1. Identifier les activités pertinentes consommant une part significative d'énergie sur les deux niveaux de granularité qui nous intéressent CABA (grain-fin) et PVT (gros-grain).

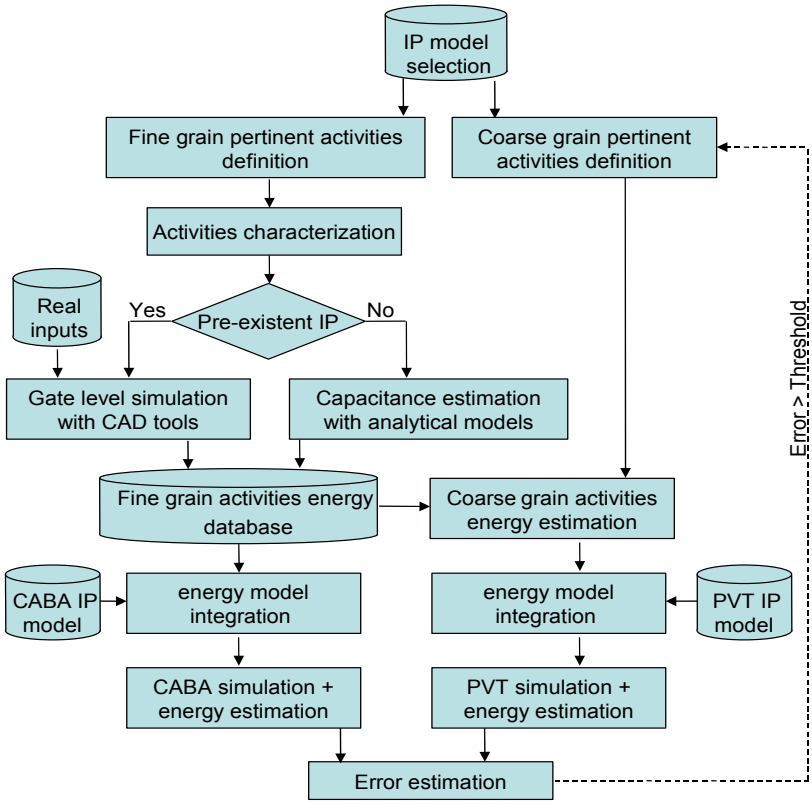


FIG. 5.4 – *Méthodologie de modélisation de l'énergie*

2. Caractériser la consommation d'énergie des activités de grain-fin :
   – Avec des simulations de bas niveau en utilisant des outils CAO pour les composants préexistants. Les simulations sont réalisées avec des données réelles.
   – Avec des modèles analytiques pour les composants non préexistants.
3. Déduire la consommation des activités de gros-grain à partir de leurs définitions et de la consommation des activités de grain-fin de l'étape 2.
4. Intégrer les modèles de consommation développés dans le simulateur d'architecture et réaliser des simulations.
5. Evaluer l'erreur d'estimation en comparant les résultats obtenus au niveau PVT à ceux du niveau CABA. Ces derniers sont considérés comme étant plus précis. Si l'erreur est supérieure à un seuil fixé, l'étape de définition des activités de gros-grain est refaite. Dans ce cas, la définition des activités de gros-grain est corrigée en intégrant de nouvelles activités de grain-fin afin d'obtenir une meilleure évaluation de la consommation dans le composant.

Cette méthodologie a été appliquée à l'ensemble de composants matériels aux niveaux CABA et PVT à savoir le processeur MIPS R3000, la mémoire cache, le réseau d'interconnexion, la mémoire de données et d'instructions, le contrôleur DMA et l'accélérateur matériel TCD-2D. Dans nos expériences, la technologie de fabrication à 90nm a été adoptée. Pour évaluer les paramètres physiques, nous avons utilisé le modèle BPTM (*Berkeley Predictive Technology Model*) [32] qui se base sur l'outil BSIM4 [90]. Le texte ci-après donne une idée sur le nombre important de paramètres à configurer (de l'ordre de 200) pour pouvoir réaliser les simulations de bas niveau.

```
* PTM 90nm NMOS

.model  nmos  nmos  level = 54

+version = 4.0          binunit = 1           paramchk= 1
+capmod  = 2            igcmod  = 1           igbmod  = 1
+diomod  = 1            rdsmod  = 0           rbodymod= 1
+permod  = 1            acnqsmod= 0           trnqsmod= 0

+tnom    = 27           toxe    = 2.05e-9     toxp    = 1.4e-9
+dtox    = 0.65e-9      epsrox  = 3.9         wint    = 5e-009
+ll      = 0            wl      = 0           lln     = 1
+lw      = 0            ww      = 0           lwn     = 1
+lwl     = 0            wwl     = 0           xpart   = 0
+xl      = -40e-9
+vth0    = 0.397        k1      = 0.4         k2      = 0.01

...

* PTM 90nm PMOS

.model  pmos  pmos  level = 54
```

```
+version = 4.0          binunit = 1           paramchk= 1
+capmod  = 2            igcmod  = 1           igbmod  = 1
+diomod  = 1            rdsmod  = 0           rbodymod= 1
+permod  = 1            acnqsmod= 0           trnqsmod= 0

+tnom    = 27           toxe    = 2.15e-009   toxp    = 1.4e-009
+dtox    = 0.75e-9      epsrox  = 3.9         wint    = 5e-009
+ll      = 0            wl      = 0           lln     = 1
+lw      = 0            ww      = 0           lwn     = 1
+lwl     = 0            wwl     = 0           xpart   = 0
+xl      = -40e-9
+vth0    = -0.339       k1      = 0.4         k2      = -0.01
....
```

La manipulation de ces paramètres nécessite une expérience dans le domaine de la microélectronique. En effet, pour la même technologie de fabrication, nous pouvons avoir plusieurs versions favorisant soit le critère de rapidité soit le critère de faible consommation. Ces difficultés nous ont poussé à développer des modèles de consommation qui dérivent des simulations de bas niveau, mais qui dépendent d'un nombre réduit de paramètres. Suivant le type de composant, un ensemble de paramètres spécifiques sera choisi. A titre d'exemple, pour une mémoire cache, le concepteur n'a pas à spécifier l'ensemble des paramètres en relation avec la structure des transistors nmos ou pmos utilisés, comme ceux donnés dans le tableau précédent. Il n'aura simplement, qu'à déterminer la taille des blocs, la taille totale du cache, l'associativité et la technologie utilisée. Ceci facilite la tâche des concepteurs de haut niveau pour aborder l'estimation de la consommation comme critère de développement.

5.5 Modèles de consommation d'énergie

5.5.1 Modèle de consommation du processeur MIPS R3000

Depuis quelques années, les concepteurs de processeurs accordent une importance de plus en plus grande à la consommation d'énergie. L'objectif des travaux réalisés ces dernières années est, là aussi, de pouvoir explorer les différentes alternatives matérielles et logicielles le plus tôt possible dans la chaîne de conception du système. Disposer d'un outil d'évaluation de la consommation d'énergie dans la phase où la sélection et le paramétrage du processeur est faite, permet sans aucun doute de réaliser un gain en temps et en consommation dans le produit final. En effet, jusqu'à une certaine date récente, la plupart des outils d'évaluation de la consommation d'énergie dans les processeurs existants travaillent au niveau de la fabrication physique du circuit. En plus du fait, que cette évaluation arrive tard et donc limite le nombre d'alternatives architecturales pouvant être testées, elle nécessite un temps de développement et de simulation très important et des outils très coûteux. Si ceci est vrai pour tous les composants du MPSoC, pour le composant processeur l'avantage de réaliser rapidement des évaluations de la consommation a une importance de premier ordre. En effet, le processeur est responsable d'une part considérable de la consommation totale du système. En plus, plusieurs familles de processeur peuvent être explorées telles que les GPP (General Purpose Processor), les DSP (Digital Signal Processor), les VLIW (Very Large Instruction Word), etc. Pour chaque famille de processeur, plusieurs fréquences de

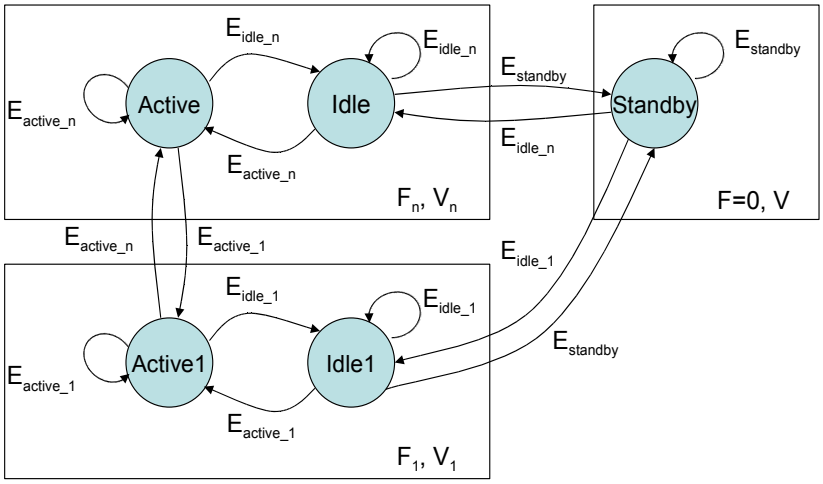


FIG. 5.5 – *La FSM qui contrôle un processeur avec 3 états de fonctionnement*

fonctionnement peuvent être choisies pour obtenir différent compromis entre les critères de performance et de consommation d'énergie.

Dans le cadre de notre étude, nous utilisons un processeur MIPS R3000 puisque nous possédons sa description SystemC ce qui facilite son intégration dans notre plateforme. L'architecture détaillée de ce processeur a été décrite dans la section 3.3.2 du chapitre 3. Notons que le modèle d'énergie proposé peut être adapté à d'autres types de processeurs tels que la famille ARM [15] ou les processeurs spécialisés DSP.

5.5.1.1 Estimation de la consommation au niveau CABA

Au niveau CABA, l'implémentation d'un processeur comporte la description de chaque étage de la micro-architecture au cycle précis. Au cours de l'exécution des instructions, les différents étages (chargement, décodage, exécution, etc.) participent à la consommation totale du processeur. Ce fonctionnement aux valeurs nominales de tension et de fréquence correspond au mode actif nominal. Cependant, le processeur peut être mis en attente dans le cas de défaut de cache ce qui correspond au mode inactif. Un processeur peut avoir d'autres modes de fonctionnement tels qu'actif à tension et fréquences différentes ou repos en basse consommation (*standby*), etc. Ces derniers modes sont utilisés pour optimiser la consommation d'énergie. La transition entre les différents modes de fonctionnement est assurée par la FSM qui contrôle le processeur. A chaque mode lui correspond une consommation d'énergie due aux activités des différents étages de la micro-architecture comme le montre la figure 5.5. Au niveau CABA, ces activités sont considérées de type grain-fin. La contribution de chaque activité dépend du mode de fonctionnement dans lequel se trouve le processeur. Dans cette thèse nous nous limitons aux deux principaux modes ; le mode actif nominal et le mode inactif.

Au niveau CABA, la consommation d'énergie dans chaque mode de fonctionnement est calculée cycle par cycle. Pour évaluer les coûts énergétiques correspondants, trois approches existent. La première approche est basée sur la documentation technique fournie par le constructeur du processeur. Le tableau 5.1 présente les caractéristiques techniques fournies par Intel pour le processeur StrongARM SA-1100 [62]. Dans ce tableau, nous retrouvons

Clock	133 MHz	190 MHz
Performance	150 Dhrystone 2.1 MIPS	220 Dhrystone 2.1 MIPS
Core power supply	**Vss** = 0.0 V dc **Vdd** = 1.5 V dc ± 5%	**Vss** = 0.0 V dc **Vdd** = 1.5 V dc ± 5%
I/O power supply	**Vssx** = 0.0 V dc **Vddx** = 3.3 V dc ± 10%	**Vssx** = 0.0 V dc **Vddx** = 3.3 V dc ± 10%
Typical power dissipation	Normal mode = <230 mW Idle mode = <50 mW Sleep mode = <50 µA	Normal mode = <330 mW Idle mode = <65 mW Sleep mode = <50 µA
Ambient operating temperature	0°C (32°F) min. 70°C (158°F) max.	0°C (32°F) min. 70°C (158°F) max.
Storage temperature	–20°C to +125°C (–4°F to +257°F)	–20°C to +125°C (–4°F to +257°F)
Packaging	208-pin LQFP†† 256 mBGA	208-pin LQFP†† 256 mBGA
Process technology	.35 µm, 3-layer metal	.35 µm, 3-layer metal
Transistor count	2.5 million	2.5 million

TAB. 5.1 – *Caractéristiques techniques du processeur StrongARM SA-1100 [62]*

la puissance dissipée respectivement dans les modes actif (*normal mode*), inactif (*idle mode*) et repos (*sleep mode*) pour deux valeurs différentes de la fréquence d'horloge (133MHz et 190MHz). L'énergie par cycle est déduite par l'équation suivante :

$$E = P \cdot T_f = \frac{P}{F} \tag{5.2}$$

Cette approche permet d'obtenir des estimations précises de la consommation puisque les valeurs d'énergie sont fournies par le concepteur du circuit. Néanmoins, il n'est pas possible d'obtenir des statistiques sur la contribution de chaque étage de la micro-architecture. De plus, cette approche n'est pas flexible si nous nous intéressons à explorer des paramètres du processeur comme : le nombre d'UAL entière ou la technique de prédiction de branchement utilisée.

Une deuxième approche pour estimer les coûts énergétiques de chaque mode de fonctionnement est d'utiliser la description du processeur au niveau RTL ou physique. Cette approche fait appel à des outils de CAO pour modéliser le circuit, faire des tests avec les données réelles et mesurer la consommation de l'énergie dans chaque mode de fonctionnement. Ainsi, cette approche permet des estimations précises. L'obstacle ici est le temps de simulation considérable pour obtenir les résultats. Ce temps dépend entre autres de la taille du circuit. Malheureusement, cette approche n'est pas applicable dans notre travail puisque nous ne possédons pas une description de bas niveau du processeur MIPS R3000.

Une troisième approche qui consiste à utiliser des simulateurs au niveau de la micro-architecture pour estimer la consommation des processeurs a été adoptée dans notre thèse. L'outil PowerAnalyzer [118] est un exemple de ce type de simulateur. Cet outil calcule la capacitance physique de chaque étage du pipeline du processeur et prend en compte la consommation statique. Avant l'exécution de l'application, le simulateur PowerAnalyzer nécessite deux fichiers de configuration. Le premier pour spécifier les paramètres de la technologie de fabrication adoptée, 90nm dans notre cas. Le deuxième fichier spécifie les para-

mètres de la micro-architecture du processeur. Le nombre de paramètres est de l'ordre de 40 et la liste ci-après en présente quelques-uns.

```
-fetch:ifqsize      \# Taille de la file de chargement des instructions
-bpred:bimod        \# Type de prédiction {nottaken|taken|bimod}
-decode:width       \# Bande passante du décodeur  (insts/cycle)
-issue:width        \# Bande passante d'exécution (insts/cycle)
-issue:inorder      \# Exécution dans l'ordre
-commit:width       \# Bande passante pour la terminaison des instructions (insts/cycle)
-ruu:size           \# Taille du registre RUU
-lsq:size           \# Taille de la file load/store
-res:ialu           \# Nombre d'UAL entières
-res:fpalu          \# Nombre d'UAL flottantes
....
```

Après l'exécution de l'application, l'outil PowerAnalyser fournit différentes statistiques concernant la consommation de puissance statique et dynamique dans chaque étage comme le montre le texte ci-après.

```
alu access                11012152 # number of times alu is accessed
alu max power               0.0001 # Maximum power for alu
alu total power           792.8749 # Total power for alu
alu avg power               0.0000 # Avg power for alu
mult access                      0 # number of times mult is accessed
mult max power              0.0000 # Maximum power for mult
mult total power            0.0000 # Total power for mult
mult avg power              0.0000 # Avg power for mult
fpu access                       0 # number of times fpu is accessed
fpu max power               0.0000 # Maximum power for fpu
fpu total power             0.0000 # Total power for fpu
fpu avg power               0.0000 # Avg power for fpu
...
uarch.switching         234658.9612 # uarch total in switching power dissipation
uarch.avgswitching           0.0098 # uarch avg in switching power dissipation
uarch.internal          824905.6538 # uarch total internal power dissipation
uarch.avginternal            0.0343 # uarch avg internal power dissipation
uarch.leakage            23044.0609 # uarch total leakage power dissipation
uarch.avgleakage             0.0010 # uarch avg leakage power dissipation
uarch.pdissipation     1082608.6757 # uarch total power dissipation
```

Notons ici, que l'outil PowerAnalyzer exprime la consommation en terme de puissance et non pas en terme d'énergie. Cependant, cette valeur peut être déduite facilement à partir de la puissance et le temps d'exécution de l'application. Différents modèles de consommation de puissance sont utilisés dans chaque étage de la micro-architecture pour estimer la consommation dans les différents modes de fonctionnement du MIPS R3000. Par la suite, les énergies consommées par les étages sont additionnées pour obtenir le coût énergétique total. Le tableau 5.2 résume les coûts en puissance et énergie obtenus pour les modes actif et inactif à une fréquence de fonctionnement de 100MHz. L'avantage d'une telle approche est

de fournir des mesures relativement précises. De plus, elle permet d'explorer les paramètres du processeur comme le nombre d'UAL utilisée et leur type, la prédiction de branchement, etc.

Processeur	Vdd (V)	f(MHz)	P_{active}(mW)	E_{active}(nJ)	$P_{inactive}$(mW)	$E_{inactive}$(nJ)
MIPS R3000	1.1	100	45	0.45	30	0.3

TAB. 5.2 – *La consommation du processeur MIPS R3000*

Au niveau CABA, nous avons donc simplifié le fonctionnement du processeur et nous avons considéré que son automate d'état peut effectuer deux types d'opérations : l'exécution d'une instruction ou l'attente de données ou d'instructions. Ce qui correspond respectivement aux modes actif et inactif. Deux compteurs correspondant aux deux modes précédents ont été ajoutés à la description du composant. A la fin de la simulation, les valeurs de ces compteurs seront multipliées par les coûts d'énergie de chacun des deux modes pour calculer la consommation totale du processeur. L'équation suivante résume la façon avec laquelle la consommation est calculée au niveau CABA :

$$E_{processor} = n_{running} \cdot E_{running} + n_{idle} \cdot E_{idle} \tag{5.3}$$

$n_{running}$ représente le nombre de cycles pendant lequel le processeur exécute des instructions (actif) et n_{idle} le nombre de cycles où le processeur est en attente (inactif).

5.5.1.2 Estimation de la consommation au niveau PVT

L'approche d'estimation de la consommation au niveau de la micro-architecture précédemment décrite ne peut pas être appliquée au niveau PVT. En effet dans ce deuxième niveau, nous avons fait le choix d'utiliser un simulateur au niveau des instructions (ISS) pour modéliser le fonctionnement du processeur. Ce type de simulateur exécute l'application instruction par instruction sans se référer à la micro-architecture du composant. Par conséquent, les activités de chaque étage ne peuvent pas être récupérées durant la simulation. Pour cela, il nous a fallu concevoir une autre méthode d'estimation de la consommation mieux adaptée à la description au niveau PVT. Au cours de la simulation, l'information pertinente qui peut être dégagée au niveau PVT est le type de l'instruction en cours d'exécution. D'où l'idée de développer un modèle de consommation qui considère une énergie dissipée pour chaque type d'instruction. L'ensemble des valeurs de la consommation des différentes instructions constituera le modèle d'énergie du processeur. Pour caractériser le coût de chaque instruction, nous avons effectué des simulations à l'aide de l'outil PowerAnalyzer. Le texte ci-après montre un exemple de code pour évaluer le coût de l'instruction *add*.

```
main()
{
int i=0;
for(i=0;i<nbr_iter;i++)
__asm(
         "add    r4,r3,r4\n"
     );
return 0;
}
```

Nous avons exécuté cette boucle deux fois, une fois avec l'instruction à tester et une deuxième fois sans cette instruction. Pour évaluer la consommation de l'instruction, la différence entre les deux simulations sera divisée par le nombre d'itérations de la boucle (nbr_iter fixé à 1 million dans les expériences).

Les résultats de simulation montrent une faible variation de l'énergie dissipée par cycle entre les différentes instructions. Cette valeur est de 8.3% ($E_{min} = 0.436nJ$, $E_{max} = 0.472nJ$). En effet, le processeur MIPS R3000 possède une architecture scalaire simple exécutant une instruction par cycle. Nous considérons ici le cas du mode actif. Les expérimentations réalisées sur les processeurs Hitachi HS-4 et StrongARM SA-1100 qui sont des architectures similaires au MISP R3000 ont donné des résultats proches [127] avec une variation maximale de consommation de 8%. Pour minimiser l'erreur, dans notre modèle de processeur nous avons utilisé un coût énergétique différent pour chaque instruction. Dans d'autres processeurs plus complexes comme ceux qui possèdent une architecture superscalaire, il est difficile d'appliquer cette méthode puisqu'il est assez difficile d'évaluer la part de chaque instruction dans un groupe de plusieurs instructions s'exécutant de façon concurrente. En effet, le coût d'une instruction peut dépendre des interactions avec les autres instructions concurrentes.

Pour notre architecture de processeur MISP R3000, le processeur est mis à l'état "inactif" dans le cas de défaut de cache. Le nombre de cycles d'attente est obtenu après chaque transaction *read (adr, data, time)* ou *write (adr, data, time)*. Le paramètre *time* est ici récupéré par le processeur pour déterminer le délai de la transaction en ns ou en cycles horloge. Dans le cas de défaut de cache, le temps d'attente du processeur est égal à la somme des valeurs suivantes : temps d'accès au cache, temps de transmission de la requête de lecture via le réseau d'interconnexion, temps d'accès de la mémoire partagée et enfin temps de transmission du bloc de cache vers le processeur via le réseau d'interconnexion. Comme il a été expliqué dans les chapitres précédents, ce temps d'attente n'est pas constant car il dépend des contentions sur le réseau d'interconnexion. L'estimation de ce paramètre influe directement sur la précision de l'estimation de la consommation du mode inactif. Pour évaluer le coût énergétique de ce mode, nous avons utilisé l'approche décrite au niveau CABA.

De l'autre côté, pour calculer l'occurrence de chaque instruction et le nombre de fois où le processeur passe dans le mode inactif, des compteurs ont été ajoutés à la description du composant. A la fin de la simulation, les valeurs de ces compteurs seront multipliées par les coûts d'énergétiques pour calculer la consommation totale du processeur. En conclusion, au niveau PVT, nous avons obtenu pour le processeur MIPS R3000 le modèle de consommation suivant :

$$E_{processor} = n_{add} \cdot E_{add} + n_{mul} \cdot E_{mul} + ... + n_{idle} \cdot E_{idle} \quad (5.4)$$

5.5.2 Modèle de consommation pour la mémoire SRAM

La mémoire dans les systèmes embarqués est responsable d'une grande partie de la consommation totale. Cette source de consommation s'amplifie avec les applications de traitement de données intensif due à la grande quantité de données transitant entre les mémoires, les processeurs et les accélérateurs matériels. Dans la littérature deux types de circuits mémoires existent : statique (SRAM) et dynamique (DRAM). Jusqu'à une date récente, seules les mémoires de type SRAM pouvaient être intégrées avec le reste du système sur la même puce. Néanmoins, aujourd'hui avec le progrès technologique, il est possible de combiner des circuits mémoires de DRAM dans un SoC [108]. Dans ce travail nous sommes intéressés à étudier le comportement des circuits mémoires de type SRAM d'un point de

vue consommation d'énergie. En général, les mémoires du type SRAM peuvent avoir plusieurs modes de fonctionnement au cours de l'exécution d'une application à savoir : lecture, écriture, inactif et repos ((*standby*). Le passage à ce dernier état est contrôlé extérieurement à partir de l'entrée *clock* ou bien *chip select*. Dans le module SRAM que nous avons développé, ce dernier mode n'a pas été utilisé.

5.5.2.1 Estimation de la consommation au niveau CABA

Au niveau CABA, les mémoires consomment une énergie moyenne par cycle pour chaque état de fonctionnement. Pour ce composant, nous avons identifié les trois activités pertinentes qui consomment de l'énergie : READ, WRITE et IDLE qui correspondent à des opérations de lecture, écriture et inactif. Pour estimer le coût énergétique des activités, trois approches existent. La première approche consiste à utiliser la documentation technique fournie par les concepteurs des circuits mémoires. Cette documentation spécifie les valeurs de l'intensité du courant pour chaque mode de fonctionnement. En général, les valeurs $I\_nominal$ et $I\_inactif$ qui correspondent respectivement aux modes nominal et inactif sont fournies. L'énergie pourra être calculée à partir de ces valeurs et des valeurs nominales de fréquence et de tension fournies elles aussi dans la documentation technique en utilisant l'équation suivante :

$$E = \frac{I \times V}{f} \tag{5.5}$$

Dans ce cas, l'énergie nominale correspond à l'énergie moyenne dissipée dans les modes de lecture et d'écriture. Le tableau 5.3 donne les valeurs d'énergie dans les modes nominal et inactif extraits de la documentation technique du circuit SRAM μPD431000A de NEC [94].

Mémoire	Taille	Tension	Fréquence	I_{nom}	I_{inac}	E_{nom}	E_{inact}
SRAM NEC μPD431000A	128Kox8bit	2.7 V	14 MHz	70 mA	100 μA	13.5 nJ	19.28 pJ

TAB. 5.3 – *Consommation d'énergie de la mémoire SRAM μPD431000A de NEC*

Cette approche présente l'avantage de la précision puisque les valeurs d'énergie dérivent de la documentation technique fournie par le concepteur du circuit. Malheureusement, cette approche n'est pas applicable si l'objectif est de réaliser une exploration de l'architecture de la mémoire.

Une deuxième approche qui consiste à utiliser des équations analytiques pour évaluer le coût énergétique de chaque mode de fonctionnement de la SRAM existe [84, 85]. Cette approche est basée sur l'utilisation de l'équation suivante :

$$E = 0.5 \times C \times V^2 \tag{5.6}$$

L'énergie dissipée dépend par conséquent de la tension d'alimentation (fixe) et de la capacitance totale. La difficulté ici est de calculer la capacitance totale pour chaque mode de fonctionnement. Cette approche peut être appliquée lorsque le circuit est décrit au bas niveau (schéma en transistors) et nécessite la connaissance des détails du fonctionnement du circuit. Nous pensons que cette approche n'est pas facilement abordable par les concepteurs

de systèmes de haut niveau. Notons néanmoins que cette approche permet de fournir un modèle paramétrable.

Pour remédier aux inconvénients des deux approches précédentes, nous proposons une approche qui se base sur des simulations de bas niveau afin d'obtenir un modèle paramétrable. Pour estimer le coût énergétique des activités, nous avons simulé des circuits SRAM de différentes tailles au niveau physique avec l'outil ELDO de Mentor Graphics [11]. Ces mémoires ont été conçues au Laboratoire d'Informatique de Paris 6 (LIP6)[6], la figure 5.6 présente la structure générale des circuits SRAM simulés. Notre mémoire possède un seul port pour la lecture et l'écriture, deux lignes de précharge pour chaque colonne et une ligne de mot pour chaque ligne mémoire. Les composants de base qui constituent cette mémoire sont la cellule mémoire de base qui sert à stocker un bit, le décodeur d'adresses pour sélectionner une ligne mémoire et le *sense amplifier* pour accélérer la phase de décharge au cours de l'opération de lecture.

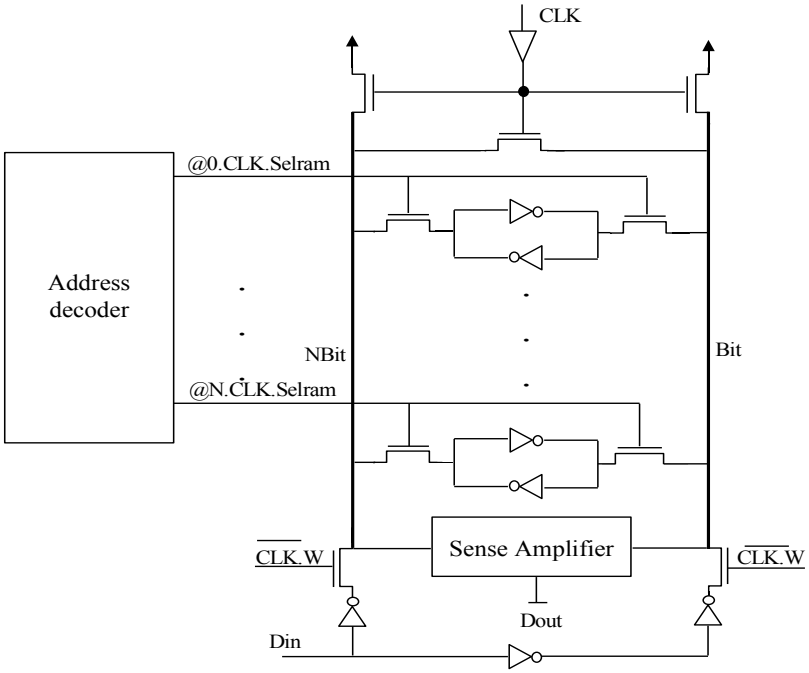
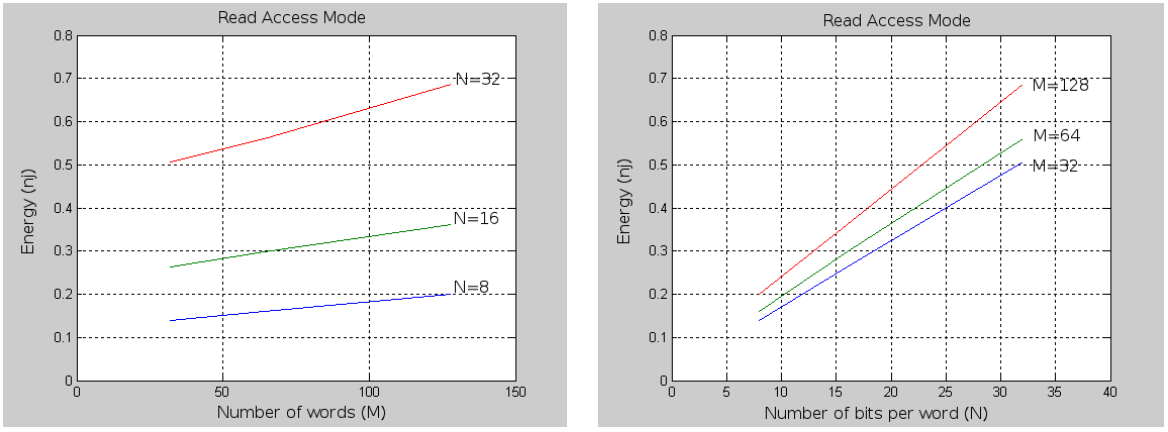


FIG. 5.6 – *Structure d'une mémoire SRAM*

La synthèse logique, le placement, le routage et l'extraction du layout sont réalisés par des outils de CAO de l'environnement de conception VLSI ALIANCE [10]. Notre objectif est d'avoir un modèle paramétré des coûts énergétiques pour les différentes activités d'une mémoire SRAM en fonction du nombre de mots (M) et du nombre de bits par mot (N). Ceci permet ultérieurement de faire de l'exploration architecturale. A partir des coûts énergétiques mesurés ainsi que les temps d'accès, nous avons déduit les coûts d'énergie des activités. Un tel modèle offre un niveau de précision relativement élevé du fait qu'il prend en considération toutes les sources de consommation possibles. Les résultats de simulation montrent que les coûts énergétiques varient linéairement avec le nombre de mots et le nombre de bits par mot. La figure 5.7.a, respectivement 5.7.b, donne le coût en énergie d'une opération de

[6]http ://www.lip6.fr

lecture en faisant varier le nombre de mots (M) avec une taille fixe de (8, 16 ou 32) bits par mot, respectivement le coût en faisant varier le nombre de bits (N) avec un nombre fixe de (32, 64 ou 128) mots.



(a) La variation du coût énergétique en fonction de M (b) La variation du coût énergétique en fonction de N

FIG. 5.7 – Coûts énergétiques en fonction du nombre de mots (M) et du nombre de bits par mot (N)

Les résultats expérimentaux nous ont permis d'écrire les équations suivantes pour chaque type d'activité :

$$\begin{aligned} E_{read} &= (R_0 + R_1 \cdot N) \cdot (R_2 + R_3 \cdot M) \\ E_{write} &= (W_0 + W_1 \cdot N) \cdot (W_2 + W_3 \cdot M) \\ E_{idle} &= (I_0 + I_1 \cdot N) \cdot (I_2 + I_3 \cdot M) \end{aligned} \tag{5.7}$$

Le coût énergétique de chaque activité est le produit de deux fonctions affines ayant N et M comme paramètres. Les constantes R_1, R_3, W_1, W_3, I_1 et I_3 représentent les coefficients directeurs des fonctions affines alors que R_0, R_2, W_0, W_2, I_0 et I_2 représentent les ordonnées à l'origine. A partir des valeurs expérimentales obtenues par les mesures sur les circuits existants, nous avons résolu ce système d'équations afin de déterminer les valeurs des coefficients R_i, W_i et I_i. Nous avons obtenu ainsi un modèle simple pour estimer la consommation des activités de base dans le composant SRAM. Il faut mentionner que ces paramètres sont valables pour une technologie donnée et pour avoir un modèle générique il faut trouver le coefficient de pondération α entre les paramètres d'une technologie λ_0 et une autre λ, on écrit alors :

$$\begin{aligned} R_i &= R_{i0} \cdot (\lambda/\lambda_0)^{\alpha} \\ W_i &= W_{i0} \cdot (\lambda/\lambda_0)^{\alpha} \\ I_i &= I_{i0} \cdot (\lambda/\lambda_0)^{\alpha} \end{aligned} \tag{5.8}$$

Dans [76], les auteurs proposent l'équation 5.9 permettant de déduire la consommation dans une technologie λ à partir d'une technologie de référence λ_0.

$$\begin{aligned} S &= \lambda/\lambda_0 \\ U &= V/V_0 \\ P' &= P \cdot (S/U^3) \\ f' &= f \cdot (S^2/U) \end{aligned} \tag{5.9}$$

Ici V_0 est la tension d'alimentation de référence, V est la tension d'alimentation ciblée. S et U représentent respectivement le facteur d'échelle de la technologie et de la tension. P' et f' représentent respectivement la puissance et la fréquence après la mise en échelle. L'énergie peut être déduite à partir de la puissance et du temps d'exécution.

Après avoir déterminé les coûts élémentaires des opérations, il est nécessaire de connaître le nombre de chaque type d'opération. Ainsi, des compteurs d'événements par opération ont été ajoutés dans la FSM qui contrôle le composant SystemC (figure 5.8). Nous avons ajouté des compteurs pour estimer le nombre de lectures, d'écritures et de cycles d'attente. Au

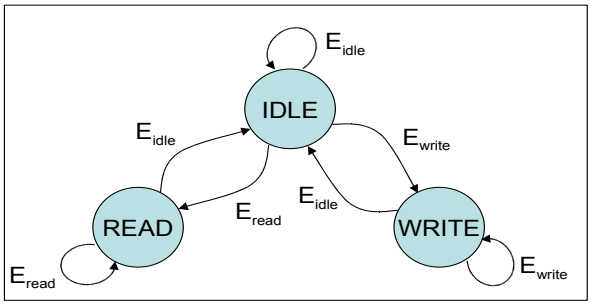


FIG. 5.8 – *FSM de la mémoire SRAM*

niveau CABA, la consommation d'énergie de la SRAM pour une application est déterminée par le modèle suivant :

$$E_{SRAM} = n_{read} \cdot E_{read} + n_{write} \cdot E_{write} + n_{idle} \cdot E_{idle} \tag{5.10}$$

n_read, *n_write* et *n_idle* sont les valeurs respectives des compteurs d'accès en lecture, en écriture et les cycles d'attente.

5.5.2.2 Estimation de la consommation au niveau PVT

Bien que la FSM qui contrôle le composant SRAM à ce niveau ne soit pas implémentée, nous définissons la même granularité pour les activités pertinentes que celle du niveau CABA. Ceci est vrai parce qu'il est possible de récupérer les activités définies (READ, WRITE et IDLE) au niveau transactionnel ce qui n'est pas toujours le cas avec les autres composants. Néanmoins, cette récupération n'est pas directement obtenue comme nous allons l'expliquer.

Au niveau PVT, la mémoire est décrite comme un composant passif comprenant deux méthodes : read() et write(). Dans la description du composant, deux compteurs sont ajoutés à chaque méthode pour déduire les occurrences correspondantes. L'obstacle ici est de déduire les occurrences qui correspondent aux cycles d'attente puisque la description au niveau PVT est dépourvue de la notion d'horloge. En plus, la fonction qui décrit la mémoire n'est exécutée que pendant les accès en lecture ou écriture. Pour résoudre ce problème nous allons utiliser le paramètre *time* des requêtes *read(address, data, time)* et *write(address, data, time)*. Ces requêtes sont transmises par les initiateurs. Le paramètre *time* contient la valeur du compteur de temps local de l'initiateur. Au niveau de la mémoire, ce paramètre est récupéré pour calculer les temps de repos (en nano seconde ou en cycles) de la mémoire. La mémoire déduit ainsi son horloge en utilisant les horloges locales des processeurs.

A chaque intervalle de temps, il est possible de calculer de la consommation d'énergie au cours de l'exécution de l'application. L'intervalle de temps est calculé entre deux transactions

successives issues du même processeur. A titre d'exemple, si le processeur 0 transmet une première requête à la mémoire partagée à l'instant 1000ns ensuite une deuxième à l'instant 1300ns alors il est possible de calculer le temps de repos entre les instants 1000 et 1300ns et par la suite la consommation d'énergie correspondante. Le temps de repos de la mémoire est égal à la différence de l'intervalle concerné (dans notre exemple 300ns) et la somme des temps d'accès à la mémoire en lecture ou en écriture. Supposons qu'il y a eu quatre accès en lecture de 50ns chacun au cours de cet intervalle de temps. Le temps de repos est égal à 100ns (300 - 4x50).

Après le calcul des occurrences des différentes activités, l'équation décrite au niveau CABA (équation 5.10) est de nouveau utilisée pour déduire la consommation totale dans le composant SRAM. En conclusion, la méthodologie présentée pour développer un modèle de consommation précis pour les mémoires SRAM nécessite un effort considérable. Cet effort nous a permis aussi d'avoir un modèle simple ne contenant pas un nombre important à configurer. Dans notre plateforme, le modèle de SRAM développé est utilisé pour estimer la consommation de la mémoire partagée d'instructions, la mémoire de données et des registres du contrôleur DMA.

5.5.3 Modèle de consommation de la mémoire cache

5.5.3.1 Estimation de la consommation au niveau CABA

Dans le chapitre précédent, nous avons détaillé la description du composant xcache au niveau CABA. Ce composant est formé principalement d'un cache de données, un cache d'instructions et une FIFO pour stocker les demandes de lecture ou d'écriture en mémoire partagée. Ces différents blocs participent à la consommation totale du composant. Au cours de l'exécution de l'application, les accès à ces blocs sont contrôlés par les deux FSM des caches de données et d'instructions. Les tableaux 5.4 et 5.5 représentent les différents états des FSM qui contrôlent le cache de données, le cache d'instructions et la FIFO de requêtes ainsi les activités qui leurs correspondent. A titre d'exemple, l'état INIT correspond à l'initialisation du cache de données ou d'instructions, l'état WRITE_UPDT correspond au chargement d'une donnée dans le cache, etc. Dans ces tableaux, *R* représente un accès en lecture, *W* un accès en écriture et "–" un état de repos. Chaque état correspond à un ensemble d'activité grain-fin de type écriture ou lecture dans le le tableau des étiquettes (Tag), écriture ou lecture dans le tableau de données ou écriture dans la FIFO. Exemple l'état INIT correspond à des écritures successives dans le répertoire et de cette façon nous identifions les activités de type grain-fin qui correspondent à chaque état.

En conclusion, l'estimation de la consommation d'un état revient à évaluer le coût d'accès en lecture ou écriture à une mémoire SRAM (le tableau des étiquettes ou le tableau de données) et le coût d'écriture dans la FIFO. Prenons comme exemple l'état INIT, nous avons $E_{INIT} = E_{write\_tag}$ et pour l'état IDLE, correspondant à un succès de cache, nous avons $E_{idle} = E_{read\_tag} + E_{read\_data}$. Ainsi, nous définissons la consommation d'énergie qui correspond à chaque état. La figure 5.9 montre les différentes énergies qui correspondent à chaque transition d'état de la FSM du cache de données.

Pour évaluer le coût d'accès en lecture ou écriture à la SRAM (le tableau des étiquettes ou le tableau de données), le travail est déjà fait dans la section précédente et pour la FIFO nous avons procédé de la même façon que la SRAM. Nous avons simulé avec l'outil ELDO des FIFO de différentes tailles pour trouver un modèle paramétrable. Dans chacun

	Activity Type		
DATA cache FSM	TAG	DATA	FIFO
DCACHE_INIT	W	–	–
DCACHE_IDLE	R	R	–
DCACHE_WRITE_UPDT	–	W	–
DCACHE_WRITE_REQ	R	R	W
DCACHE_MISS_REQ	–	–	W
DCACHE_MISS_WAIT	–	–	–
DCACHE_MISS_UPDT	W	W	–
DCACHE_UNC_REQ	–	–	W
DCACHE_UNC_WAIT	–	–	–

TAB. 5.4 – *La FSM du cache de données*

	Activity Type	
INST cache FSM	TAG	DATA
ICACHE_INIT	W	–
ICACHE_IDLE	R	R
ICACHE_WAIT	–	–
DCACHE_UPDT	W	W
DCACHE_UNC_WAIT	–	–

TAB. 5.5 – *La FSM du cache d'instructions*

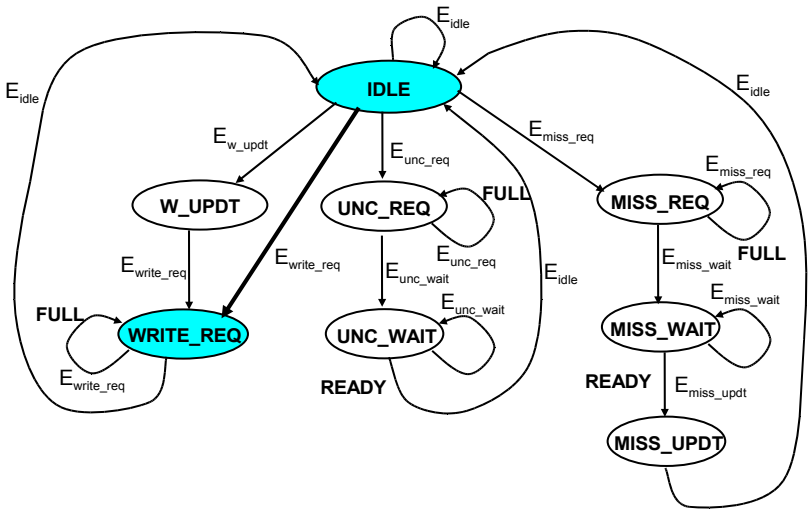


FIG. 5.9 – *FSM du cache de données*

des automates des caches, nous avons inséré des compteurs : WRITE_TAG, READ_TAG, WRITE_DATA, READ_DATA, WRITE_FIFO et IDLE. A la fin de la simulation, nous obtenons la valeur de chaque compteur qui sera multipliée avec le coût énergétique de l'activité pour trouver la consommation totale dans le composant xcache.

5.5.3.2 Estimation de la consommation au niveau PVT

Au niveau PVT, cette méthodologie pour estimer la consommation du cache d'instructions ou de données ne peut pas être appliquée du fait que la FSM qui contrôle le composant n'est pas implémentée. Les activités de grain-fin ne peuvent pas être récupérées. Pour cela nous avons défini un ensemble d'activités de gros-grain qui sont : *READ_HIT et READ_MISS* qui correspondent à un succès ou un défaut en lecture, *WRITE_HIT et WRITE_MISS* qui correspondent à un succès ou un défaut en écriture, et *IDLE* qui représente l'état inactif du composant. Chaque activité de gros-grain synthétise plusieurs transitions au niveau des états de la FSM du cache. En d'autres termes, une activité de gros-grain correspond à un ensemble d'activités de grain-fin de type écriture ou lecture dans le tableau des étiquettes, écriture ou lecture dans le tableau de données et écriture dans la FIFO.

La figure 5.10 montre la définition de chaque activité de gros-grain en fonction des états de la FSM du cache. A titre d'exemple, l'activité READ_MISS correspond à l'exécution de l'état MISS_REQ (demande de lecture d'un bloc de cache) ensuite l'exécution de l'état MISS_UPDT M fois pour la mise à jour de la ligne dans le cache. Le paramètre M présente la taille du bloc de cache. L'évaluation du coût énergétique de chaque activité de gros-grain est déduite à partir de sa définition et des coûts des activités de grain-fin qui découlent des mesures expérimentales, ce qui nous permet de produire le maximum de précision. Toujours dans le même exemple, la consommation d'énergie de l'activité gros-grain READ_MISS est la somme des consommations de l'état MISS_REQ et M fois l'état MISS_UPDT.

Au niveau PVT, un compteur est alloué à chaque activité de gros-grain dans la description du composant pour calculer l'occurrence correspondante au cours de la simulation. Comme nous l'avons déjà précisé pour la mémoire SRAM, une difficulté apparaît pour calculer les temps de repos du composant cache. Nous résolvons le problème de la même façon en utilisant le paramètre *time* des requêtes *read(address, data, time)* et *write(address, data, time)*. Néanmoins, le problème ici est plus simple car le composant cache n'est accédé que par un seul initiateur. Dans le cas de succès de cache, le temps de repos est calculé entre deux requêtes successives de l'initiateur. Dans le cas de défaut de cache, le composant instancie une nouvelle requête (*read(address, data, time)*) qui sera transmise via le réseau d'interconnexion. Le paramètre *time* sera récupéré pour savoir le temps mis pour avoir la réponse ce qui correspond au temps de repos.

Une autre méthodologie pour évaluer les coûts des activités de fin-grain et gros-grain est l'utilisation de l'outil CACTI [30]. CACTI est un outil pour estimer le temps d'accès, le temps de cycle, la surface et la consommation statique et dynamique pour les mémoires caches. En intégrant tous ces paramètres ensemble, le concepteur peut explorer différentes organisations et tailles de cache afin de retrouver la meilleure configuration pour son application. CACTI modélise analytiquement d'une façon fine les différents étages d'une mémoire cache et tient compte des paramètres de la technologie de fabrication, donc il permet une estimation de la consommation avec une précision acceptable. Cet outil modélise la consommation pour une architecture conventionnelle d'une mémoire cache. Pour les systèmes MPSoC, généralement nous utilisons des architectures plus complexes afin de gérer le problème de co-

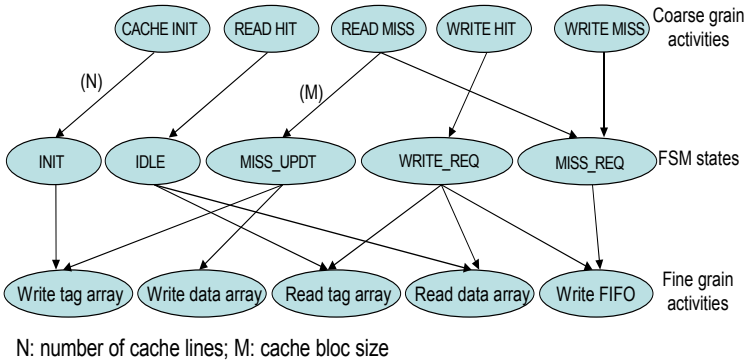


FIG. 5.10 – *Définition des activités du cache*

hérence de cache. Dans ce cas, le concepteur doit prendre en considération les modifications nécessaires dans le modèle de consommation pour obtenir des estimations assez précises.

5.5.4 Modèle de consommation du crossbar

Le modèle de consommation d'un réseau d'interconnexion dépend principalement de sa topologie (bus, crossbar, etc.) et du mode de communication (synchrone ou asynchrone). Ces derniers temps les systèmes GALS (Globalement Asynchrone, Localement Synchrone) deviennent de plus en plus répondus. Ces systèmes sont composés de plusieurs sous-systèmes (monoprocesseur ou multiprocesseurs) chacun utilisant sa propre horloge et d'un réseau d'interconnexion entre les sous-systèmes. Les unités qui composent un sous-système utilisent la même horloge et sont donc synchrones alors que le réseau d'interconnexion est conçu de façon asynchrone. Cette approche offre plusieurs avantages : évite la diffusion de l'horloge à tout le système (distorsion de l'horloge ou problème du "clock skew"), simplifie la réutilisation des composants et enfin diminue la latence de communication et la consommation d'énergie. Dans notre MPSoC nous utilisons le composant crossbar de la bibliothèque SoCLib comme réseau d'interconnexion. Le fonctionnement de ce composant est synchrone avec le reste du système. Il permet à l'utilisateur de spécifier le nombre d'initiateurs et de cibles ainsi que les latences de communication en nombre de cycles. Néanmoins, les valeurs exactes de latences dépendent principalement des longueurs de fils entre les différents ports.

Au niveau du crossbar, il y a trois principales sources de consommation d'énergie [141] :

- La consommation dissipée dans chaque fil de connexion lors d'une transition d'un bit de 0 à 1 ou de 1 à 0. Nous considérons que les énergies correspondant aux états statiques ($E_{bit0\rightarrow0}$) et ($E_{bit1\rightarrow1}$) sont nulles.
- La consommation des FIFO utilisées pour stocker les requêtes en attente à chaque entrée et sortie du réseau d'interconnexion. C'est le modèle de consommation de la SRAM qui a été développé dans la section 5.5.2 qui sera utilisé pour estimer le coût de ces FIFO. Le nombre d'accès en lecture et en écriture ainsi que les cycles d'attente sont donc calculés pour estimer la consommation totale des FIFOs.
- La consommation due à la logique nécessaire pour implémenter les fonctions de routage et d'arbitrage dans le crossbar. Des multiplexeurs ou démultiplexeurs sont généralement utilisés pour réaliser ces tâches. Dans notre travail nous considérons que cette source de consommation est négligeable.

5.5.4.1 Estimation de la consommation au niveau CABA

Parmi les trois sources de consommation, celle des fils de connexions est considérée comme étant la plus importante. Au niveau CABA, le transfert de données entre deux ports de l'interface VCI présente une activité pertinente de grain-fin d'un point de vue consommation d'énergie. La prise en compte de la consommation d'énergie au niveau du crossbar doit se faire en prenant en compte la taille en nombre de bits des différents ports. En effet cette valeur diffère d'un port à l'autre. A titre d'exemple, le port ADDRESS comprend 32 bits alors que le port CMD ne comprend que 2 bits. Rappelons ici que généralement, le transfert de données entre deux interfaces du crossbar nécessite plusieurs cycles et la consommation d'énergie dans un cycle donné dépend du nombre de ports utilisés. La figure 5.11 montre un exemple de communication entre un initiateur et une cible (2 transactions). La première transaction de lecture prend un seul cycle et lui correspond une énergie totale (E_{C0}) égale à la somme des énergies dissipées par les ports ADDRESS, RDATA, CMD, CMDVAL et EOP. L'énergie de chaque port peut être calculée par la formule suivante :

$$E_{i,j} = \alpha \cdot E_0(L_{i,j}) \tag{5.11}$$

i et j : *Numéro des ports : initiateur et cible*

Le coût de transfert d'une donnée $E_{i,j}$ dépend du nombre de bits qui ont commuté entre les états "0" et "1" et vice versa, notée α), et de E_0, qui représente le coût pour transférer un seul bit. Le paramètre α est compris entre 0 et n avec n le nombre de bits du port considéré. Il peut être calculé durant la simulation par des compteurs. Cette approche est précise, mais risque de ralentir la simulation du fait qu'il faut calculer à chaque cycle la valeur α). Un certain nombre de simulateurs comme Wattch [28] estime ce nombre à la moitié du nombre de bits au niveau de l'interface (soit ici n/2). Dans nos travaux, nous avons utilisé aussi cette solution.

Le coût élémentaire pour transférer un seul bit E_0 dépend de la longueur du fil de connexion $L_{i,j}$ entre les deux ports i et j. Le problème que nous avons rencontré concerne l'estimation de la longueur des connexions entre les deux composants i et j. En effet cette longueur dépend de l'emplacement final des composants sur la puce. Nous avons utilisé l'outil GRAAL, éditeur de layout, de l'environnement ALLIANCE pour mesurer la taille des mémoires SRAM, du contrôleur DMA et des caches d'instructions et de données en fonction du paramètre technologique λ. Pour le processeur MIPS R3000, nous avons utilisé l'approche présentée dans [86]. L'estimation de la taille de notre accélérateur matériel TCD-2D est déduite à partir de l'implémentation au niveau RTL réalisée sur le composant FPGA STRATIX II EP2S60. A partir des tailles de nos composants, nous avons développé un modèle pour estimer les longueurs des fils de connexion tout en tenant compte du nombre de processeurs et des mémoires utilisés. Une fois la phase d'estimation des longueurs des fils de connexions est terminée, l'énergie E_0 est calculée comme suit :

$$E_0 = C(L_{i,j}) \cdot V_{dd}^2 \tag{5.12}$$

$C(L_{i,j})$ est la capacitance du fil de longueur $L_{i,j}$. Pour la technologie 90nm adoptée dans nos expérimentations, la capacitance du fil est estimée à $0.25fF/\mu m$. V_{dd} est la tension des fils fixée dans nos travaux à 1.1v. Pour récupérer les occurrences de chaque port des interfaces VCI de type commande ou réponse, des compteurs ont été ajoutés à la description du

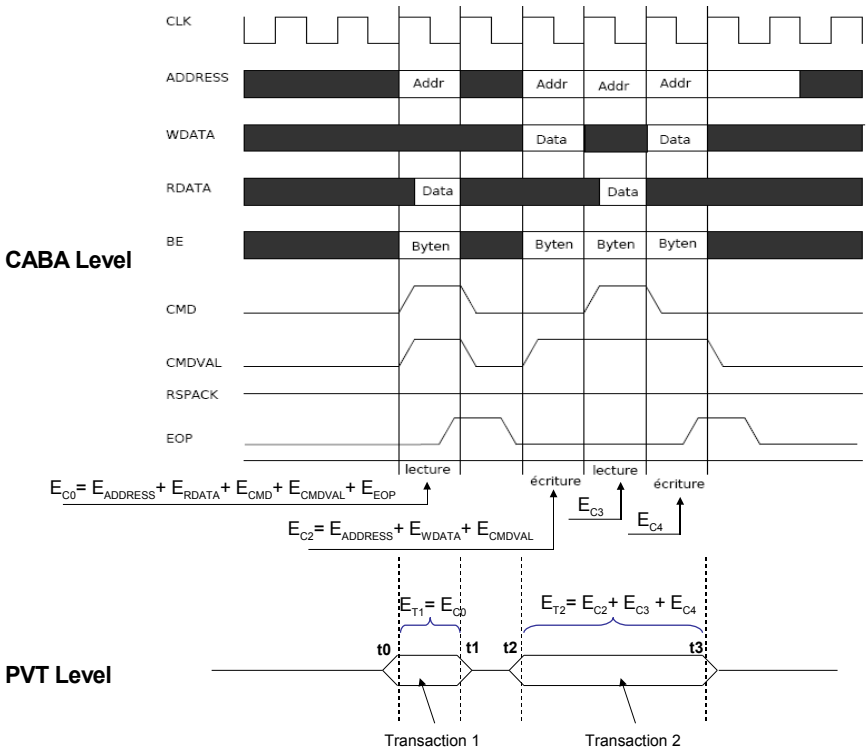


FIG. 5.11 – *Consommation de la partie communication aux niveaux CABA et PVT*

crossbar. Pour alléger la simulation, la consommation dans les fils représentant les signaux de contrôle (1 bit) a été négligée. La consommation des fils de connexion va être ajoutée à celle des FIFO des entrées et des sorties du crossbar afin de calculer la consommation totale du crossbar.

5.5.4.2 Estimation de la consommation au niveau PVT

Au niveau PVT, les transferts de données se font de façon atomique. Ils sont considérés comme des activités de gros-grain et correspondent à plusieurs phases élémentaires de transfert de données. La figure 5.11 montre que l'énergie d'une transaction au niveau PVT correspond à la somme des consommations relatives aux transferts de plusieurs mots mémoire (comme c'est le cas de la Transaction 2). Chaque mot est responsable de l'activation de plusieurs ports (activités de grain-fin). La consommation d'énergie à ce niveau est calculée par paquet pouvant contenir un ou plusieurs mots. L'équation suivante présente l'énergie dissipée lors de la transmission d'un paquet d'une source i à une cible j :

$$E = \sum_{N} E_{i,j} \tag{5.13}$$

i et j : *Respectivement, le numéro de l'initiateur et de la cible*
N : *Nombre de mots transmis dans le paquet*
$E_{i,j}$: *Coût énergétique du transfert d'un mot entre i et j*
Le coût énergétique pour transférer un mot entre deux interfaces i et j est calculé d'une façon

sommaire (pas au niveau de chaque port) avec l'équation suivante :

$$E_{i,j} = \alpha \cdot E_0(L_{i,j}) \tag{5.14}$$

Ici, α est le nombre de bits qui ont commuté entre 0 vers 1 et vice versa dans chaque interface. Dans le cas du protocole VCI, l'interface VCI commande est composée de 93 bits ($0 < \alpha \leq 93$) et celle de réponse est composée de 46 bits ($0 < \alpha \leq 46$). Dans notre travail, α est estimée à la moitié du nombre de bits de chaque interface. Au niveau PVT, des compteurs ont été ajoutés à la description du crossbar pour calculer le nombre de mots transférés entre les interfaces de type initiateur et les interfaces de type cible. Les occurrences obtenues seront multipliées par les coûts énergétiques pour déduire la consommation des fils de connexion. La valeur de cette consommation va être ajoutée à celle des FIFO des entrées et des sorties du crossbar afin de retrouver la consommation totale de ce composant.

5.5.5 Modèle de consommation de l'accélérateur matériel TCD-2D

L'objectif essentiel de l'utilisation des accélérateurs matériels dans la conception des systèmes embarqués est de pouvoir atteindre le maximum de performance en termes de temps d'exécution et de consommation. Cet objectif est généralement atteint avec une implémentation figée (ASIC) sur la puce au prix d'un manque de flexibilité dans notre système. Pour atteindre un meilleur compromis entre performance et flexibilité, une nouvelle approche consiste à intégrer des ressources reconfigurables (FPGA) dans les systèmes embarqués [92]. Ces ressources seront utilisées pour implémenter en particulier des accélérateurs matériels.

La consommation d'énergie dans un accélérateur matériel dépend essentiellement de la fonctionnalité réalisée par le composant. Pour cela, il n'existe pas dans la littérature des modèles génériques pour l'estimation de la consommation dans ces composants. Par ailleurs, les accélérateurs matériels ont généralement des architectures dédiées et s'apprêtent donc très mal à une réutilisation de celles-ci. Pour cette raison, l'utilisation des modèles analytiques pour estimer les coûts énergétiques des activités pertinentes pour ce type de composant n'est pas réalisable. Ainsi, la méthode la plus adéquate pour caractériser les accélérateurs matériels, du point de vue de la consommation, est de réaliser des simulations de bas niveau.

5.5.5.1 Estimation de la consommation au niveau CABA

Pour estimer la consommation d'énergie de l'accélérateur matériel TCD-2D, nous avons synthétisé et implémenté le composant sur la plateforme FPGA STRATIX II EP2S60 (figure 5.12). L'outil Quartus d'Altera nous permet l'estimation de la consommation statique et dynamique. Pour notre accélérateur matériel, nous allons considérer deux modes de fonctionnement possibles. Le premier est le mode actif correspondant à l'exécution de la TCD-2D sur un bloc d'image de 8x8. Le deuxième est le mode inactif correspondant à l'état d'attente des données.

Pour évaluer le coût énergétique du mode actif, des expérimentations ont été menées sur plusieurs blocs d'image de différentes fréquences. L'objectif est d'estimer l'énergie dissipée par cycle dans ce mode de fonctionnement. Les résultats de simulation montrent une faible variation de consommation entre les différents blocs. Cette variation est inférieure à 5%. Pour cette raison, nous avons adopté une méthode d'évaluation d'énergie qui considère une valeur moyenne par cycle (E_{actif}). Pour évaluer le coût énergétique du mode inactif, l'outil Quartus fournit la consommation statique de l'architecture dissipée à l'état inactif. Le

tableau 5.6 résume les coûts en puissance et énergie obtenus pour les modes actif et inactif à une fréquence de fonctionnement égale à 100MHz.

	Vdd (V)	f(MHz)	P_{active}(mW)	E_{active}(pJ)	$P_{inactive}$(mW)	$E_{inactive}$(pJ)
Accélérateur TCD-2D	1.1	100	2.13	21.3	0.51	5.1

TAB. 5.6 – *La consommation de l'accélérateur matériel TCD-2D*

Au niveau CABA le modèle d'énergie suivant est utilisé pour estimer la consommation de l'accélérateur matériel TCD-2D :

$$E_{TCD-2D} = n_{running} \cdot E_{running} + n_{idle} \cdot E_{idle} \tag{5.15}$$

$n_{running}$ représente le nombre de cycles pendant les quels l'accélérateur matériel exécute la TCD-2D (actif) et n_{idle} le nombre de cycles où le composant est en attente (inactif). Deux compteurs, qui représentent les deux modes actif et inactif, ont été ajoutés à la description du composant. A la fin de la simulation les valeurs de ces compteurs seront multipliées par les coûts d'énergie correspondants pour calculer la consommation totale de l'accélérateur matériel.

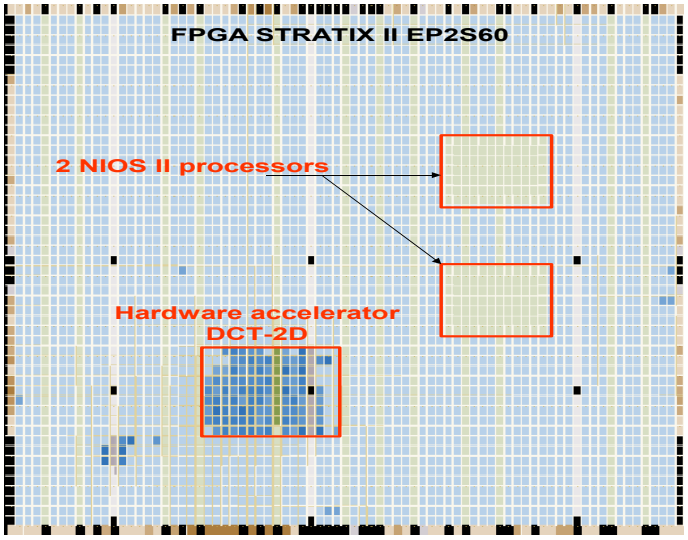


FIG. 5.12 – *L'accélérateur TCD-2D implémenté sur le FPGA STRATIX II EP2S60*

5.5.5.2 Estimation de la consommation au niveau PVT

Au niveau PVT, la description du composant n'intègre pas un compteur de temps. En effet, l'accélérateur TCD-2D est considéré comme une cible qui répond aux requêtes du contrôleur DMA. L'information pertinente que nous pouvons récupérer est le nombre de blocs traités. Nous allons considérer que l'exécution de la TCD-2D sur un bloc d'image de 8x8 est l'activité pertinente de ce composant à ce niveau. Le coût énergétique de cette activité de gros-grain est estimé à partir des mesures du niveau RTL comme ceci a été décrit précédemment. Le calcul de la consommation de l'état inactif peut se faire à chaque intervalle de

temps (en nano seconde ou en cycles) ou bien à la fin de la simulation. Dans le premier cas, l'accélérateur TCD-2D récupère le paramètre *time* des requêtes *read()* et *write()* transmises par le contrôleur DMA afin de calculer son temps de repos entre deux requêtes successives. La consommation totale d'énergie de l'accélérateur matériel TCD-2D est donnée à la fin de l'exécution de l'application. Le temps de repos est égal à la différence entre le temps d'exécution de l'application et la somme des temps de traitement des blocs traités et nous utilisons le coût énergétique de l'état inactif mesuré précédemment pour calculer la consommation statique. Des compteurs ont été ajoutés à la description du composant au niveau PVT afin de calculer les occurrences des différentes activités. Au niveau PVT le modèle d'énergie suivant est adopté :

$$E_{TCD-2D} = n_{blocs} \cdot E_{bloc} + n_{idle} \cdot E_{idle} \tag{5.16}$$

n_{blocs} représente le nombre de blocs d'image 8x8 auxquels nous avons appliqué la TCD-2D et n_{idle} le nombre de cycles où l'accélérateur matériel est à l'état inactif.

5.6 Résultats expérimentaux

Les différents modèles de consommation développés dans les deux niveaux CABA et PVT ont été intégrés dans les deux environnements de simulations MPSoC correspondants. Ces 2 environnements peuvent être utilisés pour évaluer les différentes alternatives de l'espace de solutions architecturales en prenant en compte la performance et la consommation d'énergie.

Parmi les trois sous-niveaux PVT présentés dans le chapitre précédent, nous avons choisi le sous-niveau PVT-TA pour intégrer les modèles de consommation des différents composants. En effet, ce sous-niveau présente le meilleur compromis entre l'accélération de la simulation et la précision d'estimation de performance. A l'opposé, le sous-niveau PVT-PA comporte une erreur d'estimation de performance pouvant atteindre 27%. Cette erreur influera sur la précision de l'estimation de la consommation. Par ailleurs à ce sous-niveau, l'architecture des processeurs n'est pas implémentée ce qui rend difficile l'estimation de la consommation de la partie calcul. L'intégration des modèles de consommation au sous-niveau PVT-EA permet de fournir sans aucun doute les estimations les plus précises. Néanmoins, les accélérations que nous avons obtenues à ce sous-niveau sont assez limitées.

Dans le chapitre précédent, nous avons mesuré pour le sous-niveau PVT-TA l'accélération de la simulation, la précision d'estimation de performance et l'effort de développement en comparaison avec le niveau CABA. Dans cette section, nous allons utiliser le même environnement expérimental et l'application H.263 pour évaluer la précision de l'estimation dans la consommation d'énergie du niveau PVT. Cette métrique sera présentée pour diverses tailles du cache de données et d'instructions et avec un nombre variable de processeurs. Notons cependant que notre environnement peut être utilisé pour déterminer, dans un intervalle de temps raisonnable et une précision acceptable, la configuration optimale pour d'autres paramètres architecturaux tels que : le type de processeur, le type de réseau d'interconnexion, etc.

5.6.1 Résultats de simulation au niveau CABA

Pour évaluer l'impact du nombre de processeurs sur les performances et la consommation d'énergie totale du système, nous avons exécuté l'application du codeur H.263 sur un

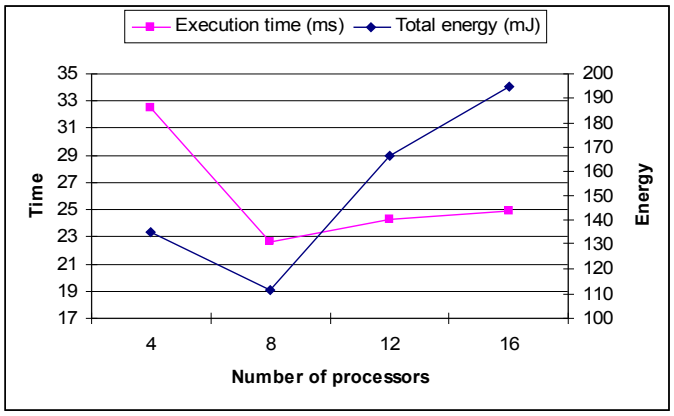


FIG. 5.13 – *Variation des performances et de l'énergie en fonction du nombre de processeurs*

système MPSoC formé de 4 à 16 processeurs. La taille des caches d'instructions et de données est fixée à 8 Ko et la fréquence des processeurs MIPS est fixée à 100 MHz. La figure 5.13 rapporte le temps d'exécution de l'application en ms et la consommation totale d'énergie en mJ estimés par le simulateur. En général, l'augmentation du nombre de processeurs fait diminuer le temps d'exécution de l'application. Ceci est le cas de notre système lorsque le nombre de processeurs passe de 4 à 8 processeurs où nous constatons une réduction de 30%. A l'opposé, le temps d'exécution augmente lorsque le nombre passe 12 de 16 processeurs du fait des collisions. En effet, les différents processeurs, partageant des ressources (cibles) communes, ne peuvent pas accéder en même temps à ces cibles. Du point de vue de la consommation, l'énergie totale du système décroît avec la diminution du temps d'exécution dans une première phase (entre 4 et 8 processeurs). Mais à partir d'une certaine limite (dans notre cas 8), l'augmentation du nombre de processeurs s'avère inefficace vis-à-vis de la consommation. En effet, là aussi, à cause de l'augmentation des conflits sur le réseau d'interconnexion des cycles d'attentes seront générés ce qui augmente la consommation totale.

Une deuxième étude a était faite sur la plateforme à 8 processeurs. L'objectif ici est d'évaluer l'impact des tailles des caches sur les performances et la consommation d'énergie totale du système. Pour cela, nous avons exécuté notre application en faisant varier la taille des caches de données et d'instructions entre 1 Ko et 32 Ko. La configuration des caches adoptée est la suivante : associativité=1, taille du bloc= 32 octets et stratégie d'écriture est "Write-Through". La figure 5.14 présente les résultats de simulation. Généralement, plus grande est la taille du cache, plus petit sera le temps d'exécution. Les résultats dépendent cependant de la taille des tâches (code) et des données à traiter. Dans notre exemple, le passage de 1 Ko à 2 Ko fait augmenter les performances de 23% alors que pour des tailles supérieures à 8 Ko ces performances ne changent pas. Du point de vue de la consommation, l'énergie totale du système décroît avec la diminution du temps d'exécution de l'application lorsque des tailles de caches de plus en plus grandes sont utilisées. Au-delà d'une certaine limite (8 Ko), la consommation du système à tendance à accroître.

La figure 5.15 présente la participation de chaque composant dans la consommation totale du système pour 8 processeurs. Les valeurs sont rapportées en pourcentage pour différentes tailles de mémoires caches. Dans notre MPSoC, les unités de calcul constituées par les processeurs MIPS R3000 et leurs caches représentent la première source de consommation dans le système. En effet, ces unités sont responsables de 37% à 59% de la consommation to-

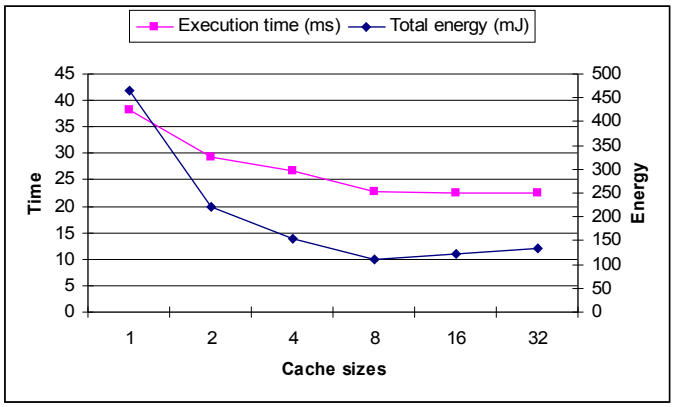


FIG. 5.14 – *Variation des performances et de l'énergie en fonction de la taille des caches*

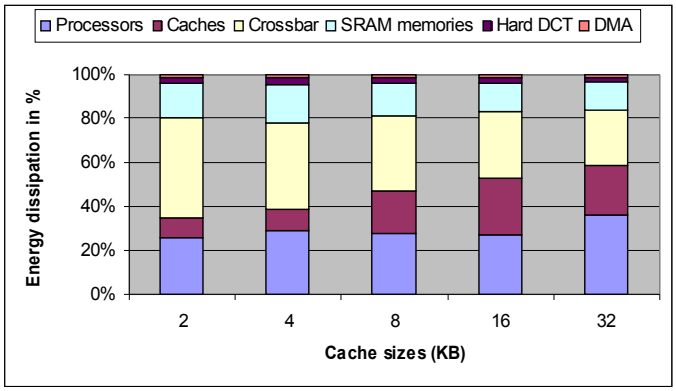


FIG. 5.15 – *Consommation d'énergie dans les différents composants*

tale du système. Il est intéressant de noter que l'augmentation des tailles de caches réduit : le nombre de défauts de cache, le temps d'exécution des processeurs, les communications sur le réseau d'interconnexion et enfin les accès en mémoire centrale. Par conséquent, l'utilisation des caches de grandes tailles fait diminuer significativement la consommation d'énergie des autres composants. Pour les caches, la consommation devient de plus en plus importante avec l'accroissement de la taille à cause de l'augmentation des coûts énergétiques des activités (lecture, écriture et repos). Le crossbar consomme entre 25% (pour 32Ko) et 45% (pour 1Ko) de la consommation totale alors que la mémoire partagée d'instructions et de données consomme moins de 20%. L'accélérateur matériel TCD-2D et le contrôleur DMA sont responsables d'un faible pourcentage de la consommation totale. Cette valeur est autour de 4%.

Nos outils peuvent aussi être utilisés pour mesurer la puissance dissipée par le système. Dans un circuit, la puissance consommée est responsable de la dissipation thermique qu'il faut prendre en considération pour des raisons de fiabilité. Ce problème est plus crucial pour les MPSoC qui intègrent plusieurs unités de calcul. La puissance dissipée peut être calculée directement à partir de l'énergie et du temps d'exécution donné par le nombre de cycles. L'environnement que nous avons développé permet de calculer la puissance dissipée par cycle ou par intervalle de temps dans chaque composant. Ceci nous permet de contrôler la

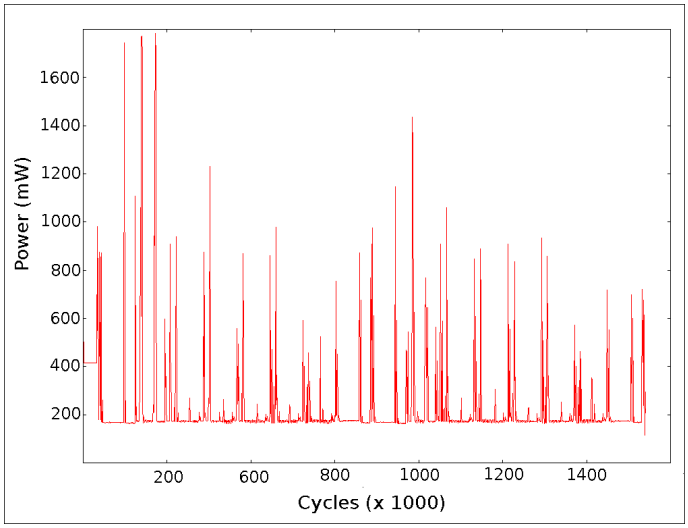


FIG. 5.16 – *Dissipation de puissance dans le crossbar sur des intervalles de 1000 cycles*

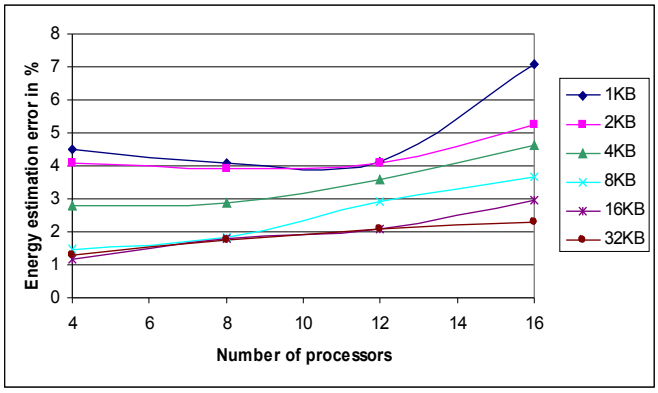


FIG. 5.17 – *Erreur d'estimation d'énergie dans le niveau PVT*

dissipation thermique totale ainsi que les pics de puissance qui diminuent la durée de vie des batteries. La figure 5.16 représente la consommation de puissance dans le crossbar pour des intervalles 1000 cycles, de cette façon nous pouvons vérifier le bon fonctionnement du circuit lors de l'exécution de l'application [2].

5.6.2 Résultats de simulation au niveau PVT

Au sous-niveau PVT-TA, nous avons intégré les différents modèles de consommation développés afin d'évaluer les solutions architecturales en se basant sur les deux critères d'énergie et de performance. Nous commençons cette section par la mesure de l'erreur dans l'estimation de l'énergie au sous-niveau PVT-TA. Cette erreur est calculée par rapport aux valeurs données dans le niveau CABA. Pour cela, nous avons exécuté notre application H.263 sur différentes configurations du MPSoC. La figure 5.17 donne l'erreur d'estimation en fonction du nombre de processeurs et de la taille des caches.

Dans cette figure, nous pouvons voir que dans le niveau PVT, l'erreur d'estimation

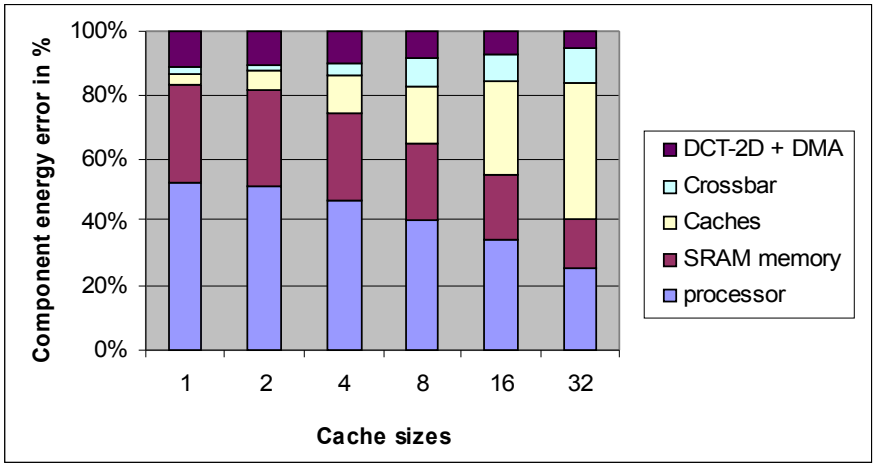


FIG. 5.18 – *Erreur dans l'estimation de la consommation d'énergie dans les différents composants*

d'énergie est de l'ordre de 7%. La courbe de cette erreur a la même tendance que celle de l'erreur de performance. Ceci est dû principalement à l'impossibilité de détecter les contentions dans le réseau d'interconnexion. En effet, ces contentions sont responsables de plusieurs cycles d'attente, ce qui génère une consommation statique des composants à l'état inactif. L'addition de nouveaux processeurs au système ou l'utilisation de caches de tailles réduites amplifie les contentions sur le réseau d'interconnexion et fait augmenter par conséquent l'erreur d'estimation de l'énergie.

Pour localiser les sources d'imprécision dans nos modèles de la consommation, nous avons mesuré l'erreur d'énergie au niveau de chaque composant. La figure 5.18 illustre l'erreur, en pourcentage, mesurée sur la plateforme à 8 processeurs en faisant varier la taille des caches de données et d'instructions entre 1 Ko et 32 Ko. Pour des caches de petite taille, les modèles de consommation du processeur et de la mémoire SRAM sont responsables d'une erreur importante de l'estimation (respectivement 52% et 30%). Ce comportement change avec l'augmentation des tailles des caches qui entraîne une diminution des cycles d'attente du processeur ainsi que des accès à la mémoire SRAM. Pour les caches, leur contribution dans l'erreur totale devient importante à partir d'une certaine taille. Dans notre exemple, cette erreur est supérieure à 29% pour une taille de 16 Ko. Pour les autres composants : le crossbar, le contrôleur DMA et l'accélérateur matériel TCD-2D, l'erreur est en général inférieure à 10%.

En conclusion, l'intégration des modèles de consommation au niveau PVT, nous a permis d'obtenir un environnement efficace pour l'exploration architecturale des systèmes MPSoC. Cet environnement offre une accélération de la simulation qui peut atteindre 18 en comparaison avec le niveau CABA et des erreurs d'estimation inférieures à 8%.

5.7 Intégration des modèles de consommation dans Gaspard

Pour faciliter l'évaluation de la consommation dans les systèmes MPSoC à différents niveaux d'abstraction, nous avons intégré les modèles d'énergie développés dans l'environnement de conception Gaspard. Cette intégration a commencé par une hiérarchisation de

ces modèles dans une bibliothèque selon le type du composant et le niveau d'abstraction. Cette bibliothèque peut éventuellement intégrer des modèles de consommation conçus par d'autres chercheurs. Pour adapter ces modèles à nos composants, il est nécessaire d'identifier les activités pertinentes afin d'ajouter les compteurs d'occurrence nécessaires dans la description de chaque composant. Dans notre approche, à un composant donné peut lui être associé plusieurs modèles de consommation, offrant ainsi différents compromis entre la précision et la rapidité d'estimation. Pour les composants dérivant d'autres bibliothèques et ayant des architectures proches des nôtres, le concepteur a la possibilité d'utiliser nos modèles de consommation pour l'évaluation de l'énergie.

Dans nos transformations dans l'environnement Gaspard, depuis la description de haut niveau jusqu'à la phase de génération de code, un lien sera créé entre les composants et les modèles de consommation. Dans une première phase, ces derniers nécessitent la spécification des paramètres d'architecture et de la technologie. Au cours de la simulation, les valeurs des compteurs sont transmises aux modèles de consommation permettant de calculer la dissipation d'énergie par cycle (CABA) ou par intervalle de temps (PVT). La dissipation totale sera calculée à partir des valeurs des compteurs transmises à la fin de la simulation. Dans le chapitre suivant, l'implémentation de notre proposition en se basant sur une approche dirigée par les modèles sera détaillée.

5.8 Conclusion

Une exploration architecturale fiable pour les systèmes MPSoC nécessite des outils d'estimation de performance et de la consommation d'énergie à différents niveaux d'abstraction. Ceci permet une réduction dans le temps de conception du système et une meilleure exploration de l'espace des solutions.

Dans ce chapitre, notre contribution a consisté à enrichir les simulateurs MPSoC décrits aux niveaux CABA et PVT par des modèles de consommation d'énergie. Nous avons proposé une méthodologie hybride d'estimation de la consommation basée sur des simulations de bas niveau et des modèles analytiques offrant un niveau acceptable de précision et de flexibilité. L'intégration du critère de la consommation dans notre travail nous a permis d'obtenir un environnement flexible pouvant être utilisé dans une exploration rapide et précise des systèmes MPSoC.

Dans notre travail, nous avons développé des modèles de consommation pour un nombre réduit de composants. Notre objectif était de valider la méthodologie de modélisation de la consommation et d'obtenir des estimations de bonne qualité. Dans les perspectives, il serait intéressant d'appliquer cette méthodologie pour d'autres types de processeurs et de réseaux d'interconnexion et ainsi enrichir la liste des IP pouvant être utilisés dans l'environnement Gaspard.

Les modèles des composants ainsi que les outils d'estimation de performance et de consommation d'énergie décrits dans les trois chapitres précédents, ont pour but de faciliter l'exploration sur plusieurs niveaux. Afin de faciliter l'utilisation de ces modèles et outils, nous proposons dans le chapitre suivant l'intégration de ces derniers dans l'environnement Gaspard basé sur une méthodologie d'ingénierie dirigée par les modèles pour la conception des systèmes MPSoC.

Chapitre 6

Ingénierie dirigée par modèles pour la simulation des MPSoC

6.1 Introduction

Ce chapitre présente notre environnement de conception des systèmes MPSoC depuis la modélisation jusqu'à la mise en œuvre suivant les concepts de l'ingénierie dirigée par les modèles (IDM). Cette approche vise à fournir un cadre de développement logiciel dans lequel des modèles passent d'un état contemplatif à un état productif. Les modèles deviennent les éléments de première classe dans le processus de développement. Dans notre contexte, l'état contemplatif correspond aux modèles de haut niveau utilisés pour exprimer les différents aspects de notre système ; application, architecture et association. Tandis que l'état productif correspond au code exécutable du système adapté à une plateforme cible. En particulier, nous nous intéressons à la simulation SystemC à différents niveaux d'abstraction. Dans ce chapitre nous allons nous attacher à l'obtention de manière automatique de ce code. Le passage entre les deux états contemplatif et productif se fait à travers des phases intermédiaires en application de processus de transformation de modèles. A ce niveau, l'avantage majeur de l'IDM est de favoriser la réutilisation à la fois des outils et des patrons de conception déjà prouvés. En pratique, cela permet d'accélérer le développement et par conséquent de réduire les coûts de conception.

La deuxième section du chapitre introduit le méta-modèle défini pour l'expression des MPSoC, il s'agit des couches hautes de l'environnement Gaspard. La principale caractéristique dans cette modélisation est l'utilisation de concepts de factorisation permettant de représenter de manière compacte des systèmes répétitifs réguliers. Cependant, par sa nature générique, ce méta-modèle n'est pas suffisant pour générer un code complet fonctionnel réalisant une simulation de bas niveau. Pour cela, comme première contribution nous proposons une évolution du méta-modèle Gaspard via l'introduction du paquetage *Deployment*. Ce dernier permet de faire le lien entre les composants élémentaires définis à haut niveau et les implémentations réelles correspondantes. De plus, des concepts liés à l'estimation de performance et de la consommation d'énergie sont introduits dans ce paquetage. Ceci fait l'objectif de la troisième section. La section 4 décrit, GaspardLib, une bibliothèque de composants logiciels et matériels pour la modélisation de SoC. Pour parvenir à notre cible de compilation, une chaîne de transformations est développée au sein de l'équipe. Cette dernière sera détaillée dans la section 5 où nous insistons sur la phase de génération de code qui est assimilée à une transformation de modèle-vers-texte et qui a été développée dans le cadre de cette thèse.

6.2 Un méta-modèle pour l'expression des systèmes MPSoC

L'environnement de conception Gaspard [140] fournit un méta-modèle pour l'expression des différentes parties constituant un système MPSoC. Ce méta-modèle est défini autour de cinq paquetages : *component*, *factorisation*, *application*, *HardwareArchitecture* et *association*. Ces trois derniers présentent les concepts de base du modèle Y de Gaspard. L'objectif des paquetages *component* et *factorisation* est de regrouper les concepts communs utilisés par les différentes parties du modèle Y pour favoriser leur processus de réutilisation. Afin de profiter des outils standards de modélisation graphique, un profil UML équivalent à notre méta-modèle est disponible. Cela permet aux utilisateurs de manipuler les différents concepts tout en ayant une présentation visuelle de leurs modèles. Ultérieurement, une transformation de modèles est utilisée pour passer du modèle conforme au profil UML vers le modèle

conforme au méta-modèle. Dans ce qui suit, nous nous limitons à présenter les cinq paquetages composant le méta-modèle. Le lecteur intéressé pourra trouver des approfondissements sur les concepts présentés dans le rapport technique dédié à ce sujet [7] et auquel nous avons contribué, plus spécifiquement dans le paquetage *HardwareArchitecture*.

6.2.1 Le paquetage *Component*

L'objectif principal de ce paquetage consiste à définir un support pour notre méthodologie orientée composant en permettant la représentation de la structure d'un composant indépendamment de son environnement d'utilisation. Le but de cette approche est de favoriser autant que possible l'aspect de réutilisation de composants logiciels et matériels. Le concept de composant représente la structure de base réutilisable dans un modèle Gaspard. Chaque composant est considéré comme un élément indépendant qui peut communiquer avec son environnement externe via la notion de *Port*. La structure et la composition hiérarchiques d'un composant sont définies via la notion d'*Instance* et de *Connector*.

De façon générale, la structure d'un composant peut être de type *élémentaire*, *composée* ou *répétitive*. Un composant de structure élémentaire (*ElementaryComponent*) ne contient aucune instance, il est vu généralement comme une boîte noire. Le concept de *CompoundComponent* est utilisé pour la définition des composants ayant une structure composée à partir d'autres composants afin d'en proposer une représentation hiérarchique. La figure 6.1 montre un exemple de composant composé, *ProcessingUnit*. Ce dernier est constitué de deux composants élémentaires, instanciés sous le nom de *c* et *mips*. Les ports et les connecteurs permettent d'indiquer l'organisation des instances pour répondre au service proposé par le composant. Un composant de type *RepetitiveComponent* possède une structure répétitive permettant de décrire la répétition de l'instance de ce composant.

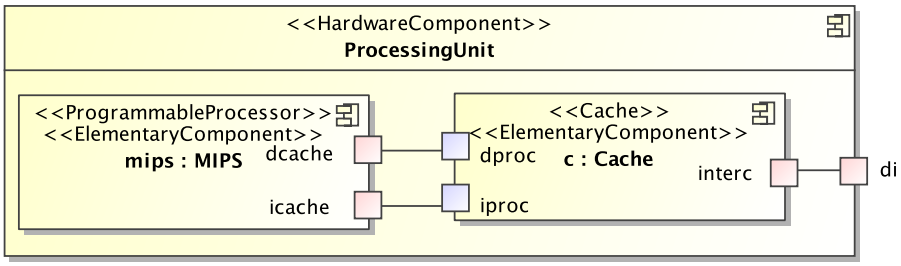


FIG. 6.1 – *Exemple de composant composé en utilisant le profil UML de Gaspard*

6.2.2 Le paquetage *Factorization*

Le paquetage *factorization* regroupe des concepts permettant de représenter de manière compacte des systèmes répétitifs réguliers. La sémantique dans ce paquetage est basée sur le langage Array-OL (Array Oriented Language) [41, 27] dédié aux applications de traitement de signal intensif. Néanmoins, cette sémantique sera généralisée afin de permettre aux développeurs de MPSoC de l'utiliser non seulement sur l'application, mais également sur l'architecture et l'association. Le premier concept introduit dans ce paquetage est celui de

Shape. Il est appliqué sur une instance de composant ou un port afin de spécifier la multiplicité sur cet élément sous forme d'un tableau multidimentionnel. Une *Shape* est décrite avec un vecteur d'entiers strictement positifs représentant le nombre d'éléments dans chaque dimension du tableau. A titre d'exemple, pour modéliser une instance de composant répétée 40 fois sous forme d'un tableau bidimentionnel 4×10, nous représentons une seule instance avec une *Shape* de (4 ;10).

Le paquetage *factorization* caractérise également les topologies de lien pour exprimer la connexion entre des éléments répétés sous une forme compacte. Les deux concepts *Tiler* et *Reshape* sont des exemples de ces topologies. Ils sont utilisés pour la spécification d'un lien entre deux ports répétés en se basant sur le langage Array-OL. Dans ce qui suit nous détaillons ces deux concepts.

6.2.2.1 Le concept de *Tiler*

L'idée générale de la notion de *Tiler* est d'identifier des sous-tableaux de points, nommés *motifs*, à l'intérieur d'un tableau (défini par une *Shape*), il faut pour cela définir la correspondance entre les points du tableau et ceux d'un motif. Ces motifs sont des tableaux multidimensionnels, par conséquent ils sont décrits à l'aide de la notion de *Shape* comme les autres tableaux. Nous appelons *tuile* un ensemble de points du tableau utilisés pour créer un même motif. Ces tuiles sont constituées de points régulièrement espacés, et les tuiles elles-mêmes sont régulièrement espacées. La description de l'espacement régulier des points d'une tuile est appelée *ajustage* (*fitting* en anglais), et la description de l'espacement régulier des tuiles dans le tableau est nommée *pavage* (*paving* en anglais). La description complète du positionnement des tuiles sur un tableau nécessite la description des dimensions du motif, l'ajustage, le pavage, une origine (qui est un point) et un espace de répétition. L'espace de répétition spécifie le nombre de tuiles. Il est également défini à l'aide d'une *Shape*.

La construction d'une tuile à partir d'un motif s'effectue à partir d'un *élément de référence* (ref) situé dans le tableau. La matrice d'ajustage est utilisée pour calculer la position des autres éléments de la tuile par rapport à ref. Cette matrice a autant de lignes que le tableau a de dimensions, et autant de colonnes que le motif a de dimensions. Les coordonnées des éléments de la tuile (e_i) sont construites comme la somme des coordonnées de ref et la multiplication de la matrice d'ajustage par les coordonnées i de l'élément dans le motif, comme suit :

$$\forall \mathbf{i}, \mathbf{0} \leq \mathbf{i} < \mathbf{s}_{\text{pattern}}, \mathbf{e_i} = \mathbf{ref} + F \cdot \mathbf{i} \mod \mathbf{s}_{\text{array}} \tag{6.1}$$

où $S_{pattern}$ est la *Shape* du motif, S_{array} la *Shape* du tableau, et F la matrice d'ajustage.

Un exemple est présenté dans la figure 6.2, une tuile (à gauche) correspondant à un motif (à droite) de 3×2 éléments (spécifié par la *Shape* $S_{pattern}$) est positionnée sur un tableau de 6×4 éléments (spécifié par la *Shape* S_{array}). La tuile est construite d'après la matrice d'ajustage (F). Cette matrice est de taille 2×2 car le tableau a deux dimensions et le motif a également deux dimensions. Le premier vecteur de la matrice, $\binom{2}{0}$, indique que pour trouver l'élément suivant de la tuile lorsque l'on parcourt le motif le long de la première dimension, il faut à chaque fois se décaler de deux points horizontalement dans le tableau. Le second vecteur de la matrice d'ajustage, $\binom{1}{1}$, indique que pour trouver l'élément suivant de la tuile lorsque l'on parcourt le motif le long de la seconde dimension, il faut à chaque fois se décaler d'un point horizontalement et d'un point verticalement dans le tableau.

Pour chaque répétition, le pavage permet de spécifier la position de l'élément de référence ref utilisé pour placer la tuile dans le tableau. Le placement de ces éléments de réfé-

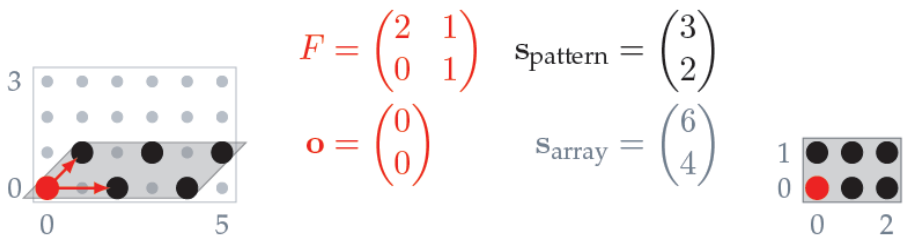


FIG. 6.2 – *Exemple de Tiler*

rence est fait de manière similaire à la construction de la tuile. La matrice de pavage a autant de lignes que le tableau a de dimensions, et autant de colonnes que l'espace de répétition a de dimensions. La position de l'élément de référence de la répétition de base est donnée par l'origine o. Les coordonnées ref_r de l'élément de référence pour une répétition r donnée sont calculées en faisant la somme de l'origine et la multiplication de la matrice de pavage par r, comme suit :

$$\forall \mathbf{r}, \mathbf{0} \leq \mathbf{r} < \mathbf{s}_{\text{repetition}}, \mathbf{ref_r} = \mathbf{o} + P \cdot \mathbf{r} \mod \mathbf{s}_{\text{array}} \tag{6.2}$$

où P est la matrice de pavage et $S_{repetition}$ est la *Shape* de l'espace de répétition.

Pour résumer, l'ajustage décrit les coordonnées des points de la tuile dans le tableau relativement à un point de référence. Le pavage décrit l'ensemble des points de référence des tuiles relativement à l'origine. L'origine est donc également le point d'indice $[0, \ldots, 0]$ de la tuile d'indice $[0, \ldots, 0]$ dans l'espace de répétition. Le processus de parcours du tableau par les tuiles est décrit par un *Tiler* possédant trois attributs : *origin* (un vecteur d'entiers), *fitting* (une matrice d'entiers), et *paving* (une matrice d'entiers). Les points de la tuile d'indice $\mathbf{r}$ dans l'espace de répétition sont énumérés comme suit. A partir d'un point donné d'indice $\mathbf{i}$ dans le motif, et pour un tableau ayant une taille $s_{tableau}$, les coordonnées du point correspondant dans le tableau sont :

$$origine + \begin{pmatrix} pavage & ajustage \end{pmatrix} \times \begin{pmatrix} r \\ i \end{pmatrix} \mod s_{tableau}. \tag{6.3}$$

Cette formule garantie que :

- Les points de la tuile sont régulièrement espacés car ils sont construits depuis le point de référence de la tuile par combinaison linéaire des vecteurs colonne de la matrice d'ajustage.
- Les points de référence des tuiles sont régulièrement espacés car ils sont construits depuis l'origine par combinaison linéaire des vecteurs colonne de la matrice de pavage.
- Tous les points de la tuile sont des points du tableau, grâce au fait que les calculs sont effectués modulo la taille du tableau.

L'utilisation du pavage pour parcourir le tableau complet en utilisant une tuile est montrée figure 6.3. Notons d'abord qu'à chaque répétition, la forme de la tuile est identique : c'est une ligne de dix points, comme spécifié par la matrice d'ajustage F et la *Shape* du motif $S_{pattern}$. Entre chaque répétition, seul le point de référence varie. Le déplacement de ce point est spécifié par la matrice de pavage P. Cette matrice est de dimension 2×1 car le tableau a deux dimensions et l'espace de répétition en a une. L'unique vecteur qu'elle contient indique qu'à chaque répétition il faut décaler le point de référence d'un point verticalement.

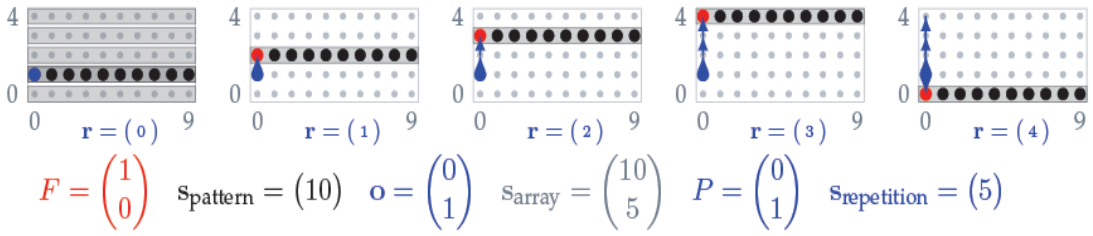


FIG. 6.3 – *Parcours de l'espace d'itération en utilisant la matrice de pavage*

Le vecteur origine *o* fait démarrer ce point de référence à la deuxième ligne. En raison des propriétés modulo du *Tiler*, la dernière répétition ne lit non pas en dehors du tableau mais la première ligne de celui-ci.

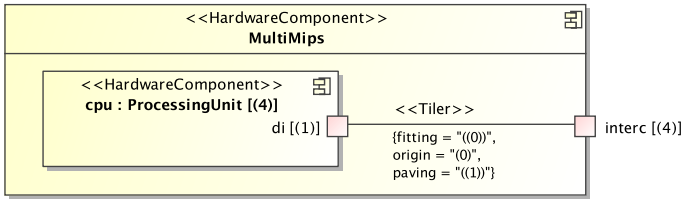


FIG. 6.4 – *Exemple d'usage d'un Tiler*

Dans le méta-modèle Gaspard, un *Tiler* est toujours utilisé comme connecteur de délégation, entre le port d'un composant et le port d'une instance contenue dans ce composant. Il est utilisable quelque soit l'orientation des ports. La taille du tableau du *Tiler* correspond à la *Shape* du port du composant. La taille du motif correspond à la *Shape* du port de l'instance. L'espace de répétition est spécifié par la *Shape* de l'instance.

Un exemple d'usage de *Tiler* est donné figure 6.4. Le *Tiler* permet de lier le port des quatre *ProcessingUnits* aux quatre ports du composant *MultiMips*. La *Shape* de l'instance de *ProcessingUnit*, (4), est vue comme l'espace de répétition du *Tiler*. La *Shape* du port cette instance, (1), est la forme du motif : le motif est donc un seul point. La *Shape* du port du composant *MultiMips*, (4), correspond à la forme du tableau. Les attributs *origin*, *fitting*, *paving* correspondent respectivement à l'origine, à l'ajustage, et au pavage. Le motif étant constitué d'un seul point, l'ajustage ne joue pas de rôle, et donc la construction de la tuile est directe : elle correspond au point de référence. Le point de référence part du premier point dans le tableau (car l'origine est nulle) et parcourt chaque point suivant du tableau au fur et à mesure des répétitions. Ainsi, le port de la première instance de *ProcessingUnit* est connecté au premier port de *MultiMips*, le port de la deuxième instance est connecté au deuxième port de *MultiMips*, etc.

6.2.2.2 Le concept de *Reshape*

Afin d'aider la tâche du concepteur lorsque des éléments de deux tableaux aux formes différentes sont liés un à un, il existe le concept de *Reshape*. Sémantiquement, il est équivalent

à deux *Tilers* mis bout à bout : le premier *Tiler* rassemble les éléments par motifs depuis le tableau d'entrée et le second *Tiler* les répartit sous une nouvelle forme dans le tableau de sortie. Ainsi, en plus des deux *Tilers* le *Reshape* définit une *patternShape* qui correspond à la forme du motif intermédiaire, et un *repetitionSpace* qui correspond à l'espace de répétition parcouru pour remplir le tableau de sortie.

6.2.3 Le paquetage *Application*

L'objectif du paquetage *Application* est de permettre l'expression des applications de traitement de signal intensif multidimensionnel. La sémantique associée à ce paquetage représente le langage de flot de données inspiré d'Array-OL, son modèle de calcul a été défini de manière complète dans [27]. Le paquetage *Application* introduit principalement la classe *ApplicationComponent* qui étend la notion de *Component* pour indiquer qu'un composant représente une partie de l'application. De plus, les concepts de factorisation précédemment décrits seront utilisés pour représenter de manière compacte le parallélisme dans les tâches d'une application.

Dans notre contexte, une application donnée est constituée de plusieurs tâches élémentaires qui peuvent être instanciées à partir des bibliothèques de fonctions (transformée de Fourrier, produit scalaire, etc.). Les tâches peuvent être assimilées à des "boîtes noires" qui prennent en entrée des tableaux lus par les ports d'entrée, ensuite exécutent la fonction concernée et enfin retournent des tableaux écrits sur les ports de sortie. Ces tableaux sont multidimensionnels l'une d'entre elles pouvant être de dimension infinie. Leur taille correspondante est représentée par une *Shape*. Dans notre description, le temps est pris en considération et peut être représenté par une (ou plusieurs) dimension des tableaux de données. Par exemple pour représenter une vidéo, un tableau tridimensionnel est nécessaire : deux dimensions pour l'image et une dimension pour le temps.

La complexité des applications provient de la combinaison des tâches et de la manière par laquelle les fonctions accèdent aux tableaux de données intermédiaires. Dans le méta-modèle Gaspard, pour définir un graphe de dépendance, les tâches d'une application sont représentées par des instances de *ApplicationComponent* connectés via leurs *ports*. Chaque port est caractérisé par la *Shape* et le type d'élément du tableau de données auquel il permet d'accéder. La méta-classe *ApplicationComponent* hérite du concept Component les trois types de structures, donc une tâche peut être élémentaire, composée ou répétitive. Si l'on veut en quelques mots définir la méthodologie de spécification d'applications efficace, il faut satisfaire les critères suivants :

- Exprimer au maximum le parallélisme potentiel entre les tâches, au niveau du graphe de tâches, éventuellement en utilisant une construction hiérarchique. Cette expression se traduit par un graphe de dépendances hiérarchique.
- Expliciter le parallélisme répétitif de chaque tâche autant sur l'espace temporel que physique.

6.2.3.1 Parallélisme de tâches

Le parallélisme de tâches est représenté via une structure de composant composée. La description interne du composant contient un graphe d'exécution acyclique. Chaque instance d'*ApplicationComponent* correspond à une tâche et chaque connecteur d'assemblage

correspond à une dépendance reliant deux ports de même type. Le sens de transfert de données est orienté des ports de sortie vers les ports d'entrée.

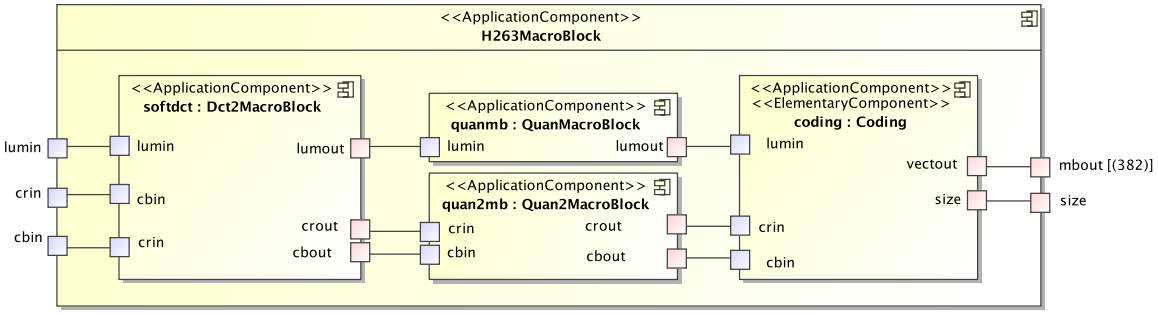


FIG. 6.5 – *Tâche composée représentée à l'aide du profil UML Gaspard*

Par exemple, la figure 6.5 présente un exemple de modélisation de l'application H.263 appliquée sur un seul macrobloc d'une image de type QCIF. Nous rappelons ici que chaque macrobloc est constitué de trois composantes ; luminance, chrominance rouge et chrominance bleue. Le graphe de dépendance est décrit à l'intérieur d'une tâche composée. Les connecteurs d'assemblage, qui sont orientés depuis les ports de sortie (dont les noms se terminent ici par *out*) vers les ports d'entrée (dont les noms se terminent ici par *in*), forcent une exécution de l'instance de la tâche *softdct* en premier lieu et celle de *coding* en dernier lieu. Concernant la tâche de quantification elle est modélisée par deux tâches qui s'exécutent en parallèle. La première *quantmb* opère sur la luminance alors que la deuxième *quant2mb* opère sur les deux chrominances rouge et bleue. Cette modélisation est possible car il n'y a pas de dépendance de données entre les différentes composantes d'un seul macrobloc.

Avant l'exécution d'une fonction donnée, une tâche élémentaire lit ses tableaux d'entrée. Ces derniers sont lus une seule fois afin de produire un tableau en sortie. La figure 6.5 présente un graphe de dépendance et non pas un graphe de flot de données. Après l'exécution, la tâche élémentaire écrit entièrement dans ses tableaux de sorties. Ainsi, il est possible d'ordonnancer les tâches juste à l'aide de ce graphe. Ce dernier permet d'exprimer tout le parallélisme de tâches potentiel. Cependant, il n'est pas possible de connaître le parallélisme de données puisqu'aucune information n'est indiquée concernant l'ordre d'accès aux données.

6.2.3.2 Parallélisme de données

Le parallélisme de données est exprimé par une tâche répétée. L'hypothèse de base est que chaque répétition de la tâche produit des données indépendantes. C'est-à-dire que les répétitions peuvent être exécutées dans n'importe quel ordre, y compris en parallèle. Par ailleurs, autour d'une tâche répétée, tous les connecteurs doivent être des *Tilers*. Les *Tilers* permettent le calcul d'adresses des motifs à partir des tableaux représentés par les ports de la tâche. Ces derniers sont représentés par les ports de la sous-tâche. Notons également qu'à partir des spécifications du *Tiler* il est possible pour chaque tableau de vérifier formellement que l'ensemble des répétitions d'une tâche n'écrivent qu'une seule fois chaque élément.

La figure 6.6 illustre un exemple de tâche répétée. Elle concerne l'application H.263 (*H263Encoder*) définie avec une sous-tâche *H263mb* qui est répétée 11×9 afin de traiter les

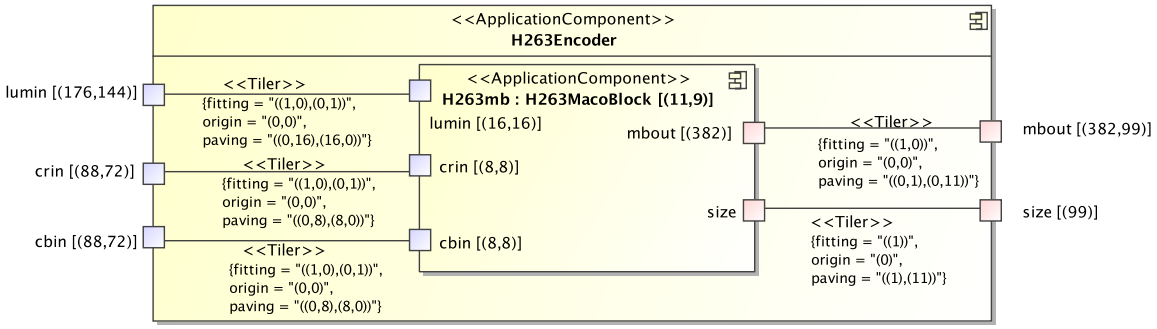


FIG. 6.6 – *Expression du parallélisme de données pour l'application H.263*

99 macroblocs d'une image de type QCIF (176 × 144 pixels). Cinq *Tilers* ont été spécifiés pour exprimer comment les motifs de la sous-tâche sont lus ou écrits depuis les tableaux de données de la tâche. Rappelons que tous les *Tilers* ont le même espace de répétition, qui correspond à la répétition de la sous-tâche, ici il est de 11 × 9.

6.2.4 Le paquetage *HardwareArchitecture*

L'objectif du paquetage *HardwareArchitecture* est d'introduire des concepts liés à la description des composants dans une architecture utilisée comme support d'exécution pour les applications. Une vue globale de ce package est donnée par la figure 6.7. Il existe quatre principaux types de composants matériels : *Processor* (les processeurs), *Memory* (les mémoires), *Communication* (les réseaux d'inter-connexion), et *IO* (les périphériques d'entrée/-sortie). Chacun de ces types est décomposé en un ensemble de types plus précis représentant une abstraction des ressources matérielles physiques.

Le paquetage *HardwareArchitecture* introduit la classe *HardwareComponent* qui étend la notion de *Component* pour indiquer qu'un composant représente une partie de l'architecture. Les concepts de port et de connecteur sont utilisés pour modéliser les liaisons directes entre les composants. De plus, les mécanismes de factorisation précédemment décrits peuvent être appliqués pour représenter de manière compacte le parallélisme dans l'architecture. Cette dernière peut être modélisée de façon hiérarchique en s'appuyant sur des structures de composants élémentaires, composées et répétitives.

Un exemple de structure d'architecture matérielle composée est présenté dans la figure 6.8. Le composant *HardwArchit* correspond à la globalité de la partie matérielle du système. Il est composé de plusieurs instances à savoir : quatre unités de calcul (*m*), quatre bancs mémoires de données (*d*), une mémoire d'instructions (*i*) et un réseau d'interconnexion (*interconnect*) qui les relies.

Un deuxième exemple montre l'utilisation du mécanisme de factorisation pour représenter une topologie d'architecture régulière. L'architecture modélisée dans la figure 6.9 est un SoC massivement multiprocesseur composé d'une grille de 16 × 16 unités de calcul (appelées *ProcessingUnit*). La topologie de lien *InterRepetition* est utilisée pour assurer la communication entre les différents éléments de la grille. Chaque instance de l'unité de calcul est reliée à quatre autres instances via les ports *n*, *e*, *s*, et *w*, ce qui forme une grille torique. Une *ProcessingUnit* est composée d'un processeur, d'une mémoire et d'un crossbar pour assurer les

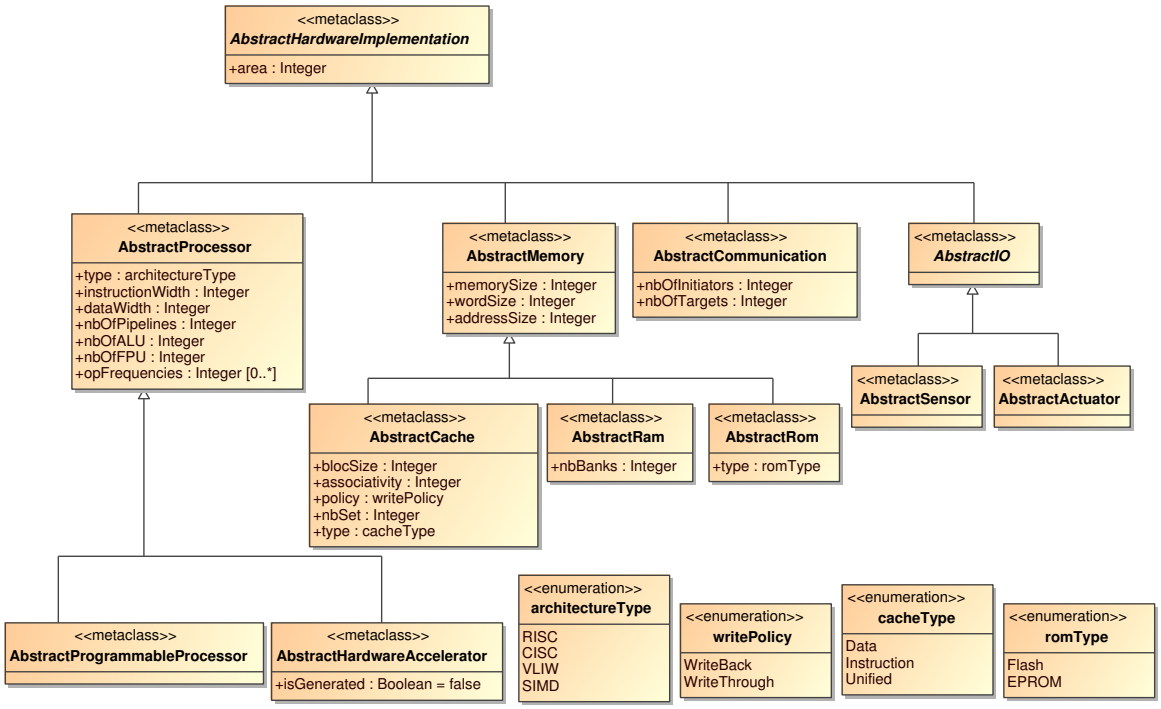


FIG. 6.7 – *Le paquetage HardwareArchitecture*

communications.

6.2.5 Le paquetage *Association*

Le paquetage *Association* permet au concepteur d'établir un lien entre une application et une architecture matérielle utilisée comme support d'exécution. Ce paquetage définit les concepts nécessaires pour permettre le placement des différents éléments de l'application sur les éléments de l'architecture. Le concept *TaskAllocation* permet de spécifier l'allocation d'une tâche sur une ressource de calcul (un processeur). *DataAllocation* est utilisée pour indiquer l'allocation d'un tableau de données sur une mémoire. Notons que l'allocation ne représente qu'un placement spatial de l'application sur l'architecture. De nouveau, les mécanismes de factorisation peuvent être appliqués pour exprimer la distribution d'une tâche répétée sur plusieurs processeurs ou indiquer la répartition d'un tableau de données sur un groupe de mémoires. La figure 6.10 montre un exemple de placement de l'application *ImageSegmentation* qui contient une tâche répétée (256×256) sur une grille de processeurs (16×16). Concernant l'allocation des tâches, nous spécifions une distribution par bloc de {16,16} des {256, 256} répétitions de l'instance de la tâche *xy* sur les {16,16} répétitions de l'instance d'architecture *p*. Pour l'allocation des données, nous spécifions une distribution des tableaux *IPH* et *IPV* par bloc de {82,50} sur les {16,16} répétitions de l'instance du composant mémoire de l'architecture. Nous notons ici que notre mécanisme d'association peut être appliqué à différents niveaux de hiérarchie d'application ou d'architecture.

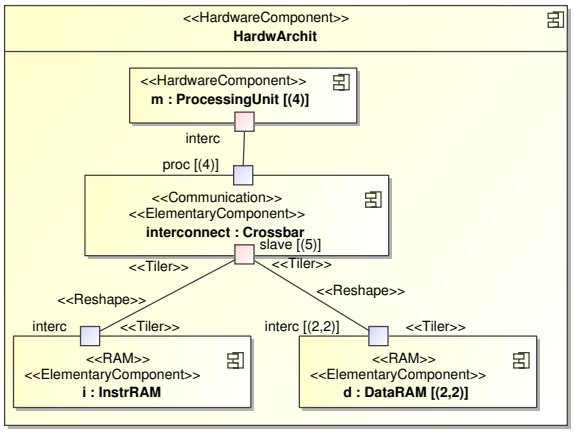


FIG. 6.8 – *Structure composée d'une architecture*

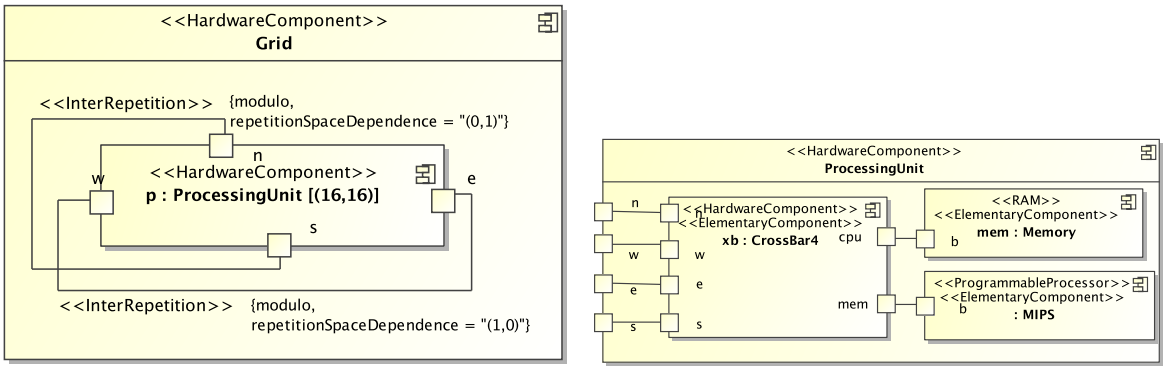


FIG. 6.9 – *Grille de* 16×16 *unités de calcul*

6.3 Méta-modèle de déploiement

Le méta-modèle précédemment décrit permet une modélisation de haut niveau de notre système MPSoC. Les différents aspects de l'application et de l'architecture sont représentés en se basant sur le concept de composant et plus particulièrement celui de structure élémentaire. Ce type de composant est vu simplement comme une brique de base possédant une fonctionnalité (calcul, stockage de données, FFT, etc.) précisée par le concepteur. Néanmoins, cette représentation est dépourvue de toute description interne. En résumé, le modèle MPSoC de haut niveau ne contient pas les informations nécessaires pour générer un modèle exécutable de bas niveau. Pour répondre à ce besoin, les composants élémentaires doivent être associés à une implémentation déjà existante sous forme de code. A titre d'exemple ce code peut correspondre à une description en VHDL ou en SystemC pour les composants matériels ou bien une description en assembleur ou en C pour les composants logiciels. Dans notre travail, nous appelons ces implémentations de briques de base, IP (Intellectual Property). De plus, la phase d'expression des liens entre la modélisation de haut niveau d'abstraction et les IP est appelé *déploiement*.

Dans cette thèse, nous avons proposé un ensemble de concepts pour exprimer cette phase

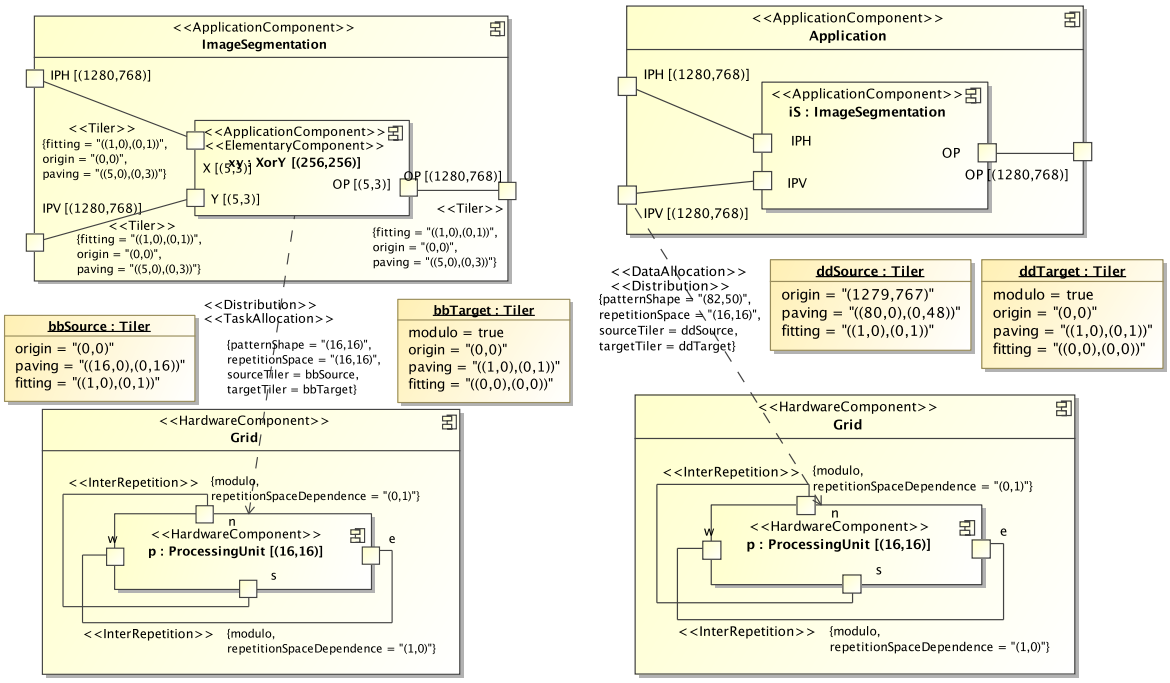


FIG. 6.10 – *Placement d'une application sur une grille de processeurs*

ce qui étend le méta-modèle Gaspard via l'introduction d'un nouveau paquetage nommé *Deployment* [4]. Les principaux attraits de ce paquetage sont de :

- Favoriser l'aspect de réutilisation de composants disponibles dans des bibliothèques d'IP. Ces dernières sont décrites à différents niveaux d'abstraction (CABA et PVT pour le matériel, C et assembleur pour le logiciel).
- Permettre de choisir l'implémentation de l'IP spécifique à une cible de compilation donnée. Par exemple en projetant une simulation SystemC au niveau TLM, il est préférable d'utiliser des composants matériels liés au même langage et niveau d'abstraction.
- Fournir suffisamment d'informations pour que l'intégration de ces IP puisse être faite d'une façon automatique lors de la génération de code et la compilation. Ceci nécessite des connaissances précises du format des interfaces d'entrée et de sortie et de la manière dont l'IP doit être appelé.
- Spécifier les informations liées à l'estimation de performance et à la consommation d'énergie en tenant compte des niveaux d'abstraction.

Comme le reste du méta-modèle Gaspard, ce paquetage est disponible à la fois sous forme de méta-modèle (utilisé par les transformations de modèles) et sous forme de profil UML (utilisé pour la conception à l'aide d'outils standards UML). Dans ce qui suit, nous détaillons le méta-modèle, néanmoins nous utilisons parfois le profil afin de bénéficier d'une représentation graphique des exemples à l'aide du langage UML.

6.3.1 Le Concept *AbstractImplementation*

Le concept *AbstractImplementation* que nous proposons reflète notre point de vue concernant la classification de nos IP logiciels et matériels. La figure 6.11 illustre une vue locale du paquetage *Deployment* autour du concept *AbstractImplementation*. Ce concept permet de regrouper les différentes implémentations (IP) du même composant logiciel ou matériel dans un seul ensemble. Ces implémentations peuvent se différer du point de vue langage de description, niveau d'abstraction, etc. Le concept *Implementation* permet de représenter directement un IP pour une cible donnée (simulation SystemC au niveau PVT). Via le lien de composition `implementation`, une *AbstractImplementation* peut contenir plusieurs *Implementations*. L'avantage d'une telle présentation est d'obtenir un modèle indépendant de la cible. De plus, l'utilisateur n'est pas obligé de modifier le modèle en fonction des transformations qu'il choisit d'appliquer. En effet, lors du déploiement d'un composant élémentaire, le concepteur du système peut lier directement le composant à l'*AbstractImplementation* sans devoir spécifier un IP précis. C'est dans une transformation de modèles que le choix entre les différentes implémentations sera fait. La transformation décide celle qui semble la plus adaptée au code qui va être généré.

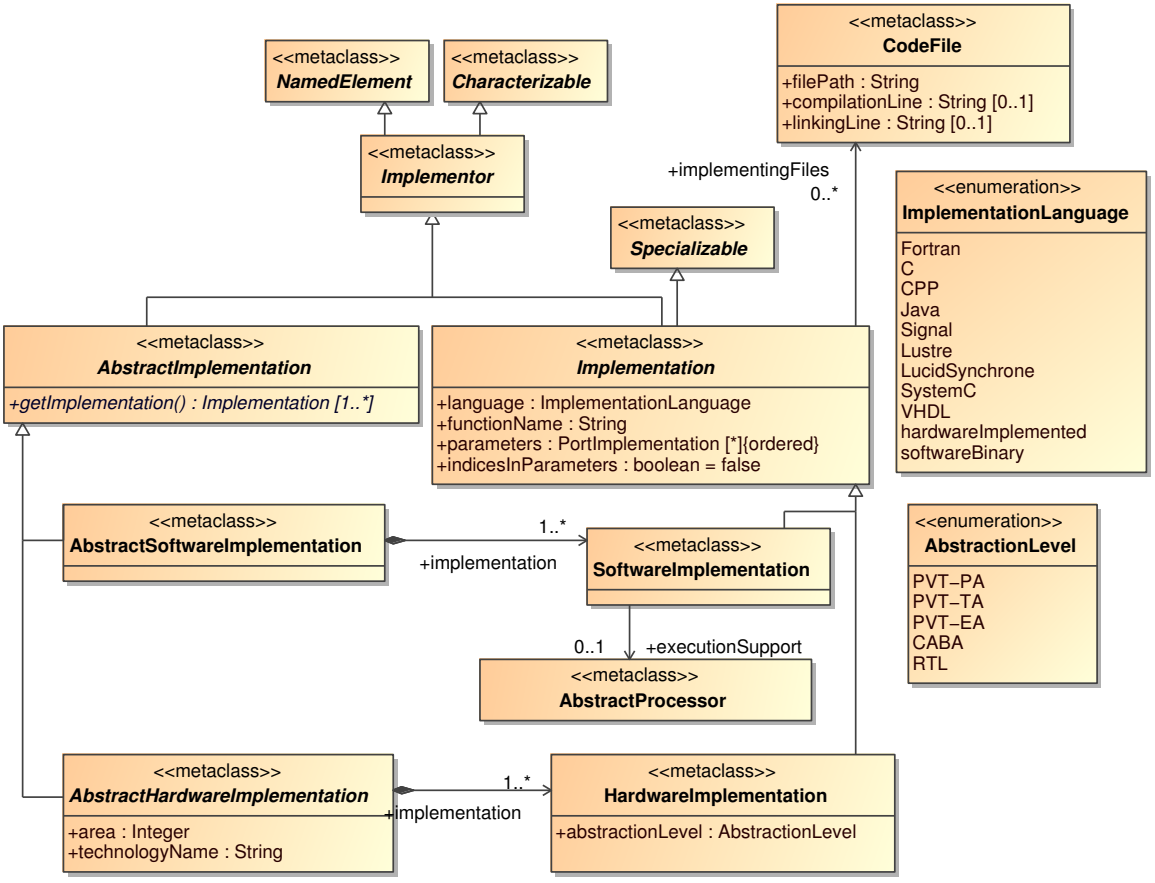


FIG. 6.11 – *Vue locale du paquetage Deployment*

6.3.2 Le Concept *Implementation*

Afin d'assurer le lien entre le code généré du système et le code de l'IP, quatre attributs sont définis dans la méta-classe *Implementation*. L'attribut `language` permet de spécifier le langage de description de l'IP qui prend une valeur de l'énumération *ImplementationLanguage* telle que Fortran, SystemC, etc. Cette information est importante pour savoir si nous avons besoin d'adapter le langage de l'IP avec celui du système généré. Un deuxième attribut important est celui de `functionName`. Ce dernier permet d'instancier un IP pendant la phase de génération de code. De manière générale, il y'a plusieurs façons de déclarer une fonction, de décrire le type de passage de ces arguments par valeur ou par référence et de retourner la valeur de sortie de la fonction. Pour cela, nous avons besoin d'adopter un protocole d'appel de fonction afin de réduire le nombre d'attributs requis et simplifier les transformations. Néanmoins, cette approche nécessite d'écrire des adaptateurs pour les IP qui ne respectent pas le protocole d'appel défini. Concernant les IP logiciels nous avons choisi d'utiliser le protocole d'appel standard C puisque les langages les plus importants que nous souhaitons supporter sont C, C++, Fortran, etc. En plus du nom de la fonction, nous avons besoin de lister les entrées et les sorties dans l'ordre lors de l'appel de fonction. Ceci fait l'objectif du troisième attribut `parameters`. Le protocole pour les langages matériels est relativement différent mais repose sur les mêmes informations. L'attribut `functionName` est utilisé pour indiquer le nom de l'IP matériel. Si le langage nécessite de spécifier l'ordre des ports de l'interface, ceci est donné par l'attribut `parameters`. En SystemC, les composants matériels sont déclarés en utilisant la classe SC_MODULE. L'instanciation du composant se fait par l'usage d'un `new` suivi du nom de la classe. Le dernier attribut `indicesInParameters`, de type booléen, est utilisé pour indiquer si la fonction appelée a besoin de l'indice de répétition courant.

La méta-classe *Implementor* présentée dans la figure 6.11 joue un rôle mineur. Elle est une généralisation de *AbstractImplementation* et *Implementation*. Elle autorise le déploiement d'un composant élémentaire directement sur une *Implementation* au lieu de le déployer sur une *AbstractImplementation*.

6.3.3 Le Concept *CodeFile*

Une fois que la phase de génération de code est réalisée, nous avons besoin de compiler ce code avec les fichiers sources de chaque IP utilisé. Pour cela nous avons introduit la notion de *CodeFile* pour préciser les informations concernant chacun de ces fichiers. Un *CodeFile* est utilisé pour représenter un seul fichier en spécifiant trois attributs. Le premier, `filePath` contient le répertoire où se trouve le fichier sur la machine de travail sur laquelle le code du système est généré et ensuite compilé. Les deux autres attributs, `compilationLine` et `linkingLine` indiquent des options particulières pour respectivement compiler et lier le fichier. La spécification du premier attribut est obligatoire alors que celle des deux autres attributs est optionnelle.

Nous utilisons la référence `implementingFiles` pour lister l'ensemble de *CodeFile* nécessaire pour une *Implementation* donnée. En réalité, il n'y a pas une correspondance directe entre ces deux derniers concepts. Une *Implementation* peut nécessiter plusieurs *CodeFile* tel qu'un IP matériel décrit avec un fichier code (master.cpp) et un fichier de déclaration (master.h). Comme aussi plusieurs implémentations peuvent partager le même *CodeFile* tel qu'un ensemble de fonctions logicielles déclarées dans un même fichier.

La figure 6.12 résume les trois principaux concepts du paquetage *deployment*. Elle montre l'usage d'*AbstractProgramableProcessor* qui est une spécialisation d'*AbstractImplementation* pour indiquer qu'il s'agit d'une implémentation abstraite d'un processeur programmable. Notre exemple concerne un processeur MIPS 32 bits qui peut être cadencé à 100, 150 ou 200 MHz. Deux implémentations du même processeur MIPS sont disponibles à différents niveaux d'abstractions, PVT et CABA, et toutes les deux sont codées en SystemC. Selon le niveau d'abstraction de la simulation que l'utilisateur souhaitera générer l'une ou l'autre sera sélectionnée pour simuler le processeur MIPS. Le *CodeFile* PVT-MIPS-CodeFile précise les informations concernant le fichier qui contient la description du composant au niveau PVT ainsi que les options de compilation et de liaison.

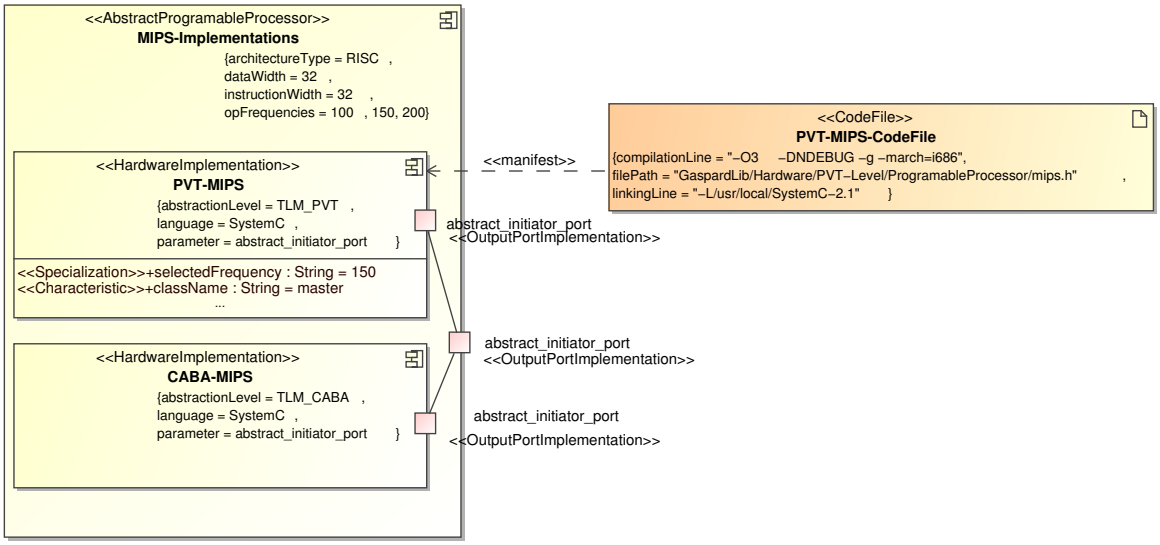


FIG. 6.12 – *Implémentation abstraite du processeur MIPS*

6.3.4 Distinction Software/Hardware

Dans notre méta-modèle *Deployment*, nous faisons la séparation entre une implémentation logicielle et matérielle. Un composant élémentaire du modèle d'application ne peut pas être déployé sur une implémentation de type matérielle et vice versa. De ce fait, nous avons séparé le concept *AbstractImplementation* en *AbstractHardwareImplementation* et *AbstractSoftwareImplementation*. De façon similaire, le concept *Implementation* a été séparé en *HardwareImplementation* et *SoftwareImplementation*.

Comme nous avons illustré dans la figure 6.7 (page 143), la méta-classe *AbstractHardwareImplementation* est une généralisation de plusieurs types de composants tels que *AbstractProcessor*, *AbstractMemory*, etc. Pour chaque type nous définissons un ensemble d'attributs liés à l'architecture du composant tels que la taille d'un cache, le nombre d'initiateurs et de cibles dans un réseau d'interconnexion, etc. Les attributs définis pour un *AbstractHardwareImplementation* particulier sont valables pour toutes ces *HardwareImplementations* et ils seront utiles lors de la compilation des fichiers sources correspondants. Les différentes implémentations d'un composant matériel se distinguent par les deux attributs `abstractionLevel`

et `ImplementationLanguage`. Le premier attribut est une énumération qui indique le niveau d'abstraction dans lequel est décrit l'IP. Les valeurs possibles sont PVT-PA, PVT-TA, PVT-EA, CABA et RTL. Dans le cadre de Gaspard, cela permet de choisir l'IP le plus adapté au niveau d'abstraction de la simulation que nous souhaitons générer. L'attribut `technologyName` définit le processus technologique tel que 130nm, 90nm etc.

Concernant la partie logicielle, la méta-classe *AbstractSoftwareImplementation* représente l'ensemble de *SoftwareImplementations* possibles d'une tâche élémentaire de l'application décrite dans Gaspard. A titre d'exemple, la tâche de transformée en cosinus directe (TCD) sera représentée par plusieurs *SoftwareImplementation* écrite en C++, VHDL, etc. Pour chaque implémentation, nous avons défini une référence `executionSupport` pour spécifier le type du support d'exécution de la tâche qui peut être un processeur programmable ou bien un accélérateur matériel. Cette spécification nous permettra ultérieurement de caractériser une tâche de point de vue temps d'exécution et consommation d'énergie pour un IP matériel donnée.

6.3.5 Le Concept *PortImplementation*

Afin d'intégrer un IP dans son environnement, nous avons besoin de spécifier son interface de communication. Cette interface est constituée de plusieurs ports via lesquels le composant est connecté au reste du système. Nous avons introduit le concept *PortImplementation* permettant de spécifier les informations importantes pour générer le code qui va lier l'IP avec le reste du système. Ce concept sera appliqué sur les ports d'une *Implementation* et ceux de l'*AbstractImplementation* englobante. Il se décline sous trois formes *OutputPortImplementation*, *InputPortImplementation*, et *InputOutputPortImplementation*. La première forme indique un port qui émet des requêtes, la seconde indique un port qui reçoit des requêtes et enfin la troisième correspond à un port qui peut aussi bien émettre que recevoir les requêtes.

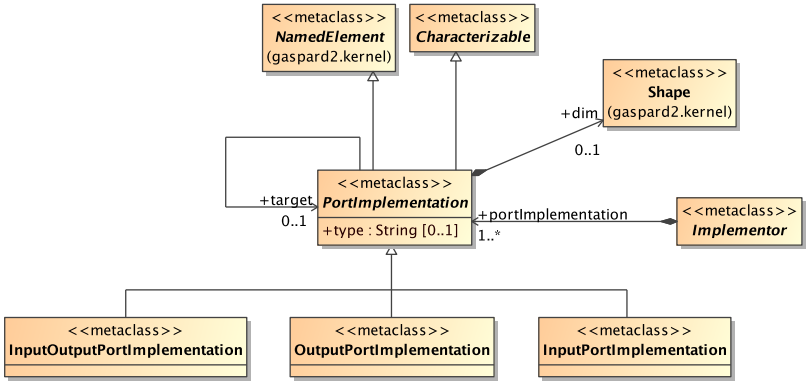


FIG. 6.13 – *Vue locale du paquetage Deployment autour de la notion de portImplementation*

La figure 6.13 présente la vue locale de la méta-classe de *portImplementation*. Quatre attributs sont définis pour cette méta-classe. L'attribut `name` joue un rôle plus au moins important dans la spécification des ports des composants. Dans le cas d'un port matériel, le nom est utilisé lors de la génération du code pour l'instanciation et la connexion avec les autres ports. En revanche, l'attribut `name` dans un port logiciel joue uniquement un rôle de documentation.

Le deuxième attribut `target` permet à une *PortImplementation* contenue dans une *Implementation* de référencer le *PortImplementation* équivalent de l'*AbstractImplementation* englobante.

Le troisième attribut `dim` permet de spécifier la taille du tableau de données dans le cas d'un port d'un IP logiciel. Au contraire dans le cas d'un IP matériel, la taille signifie que le port est répété selon la forme spécifiée. Le quatrième attribut `type` est spécifié par une chaîne de caractères qui sera utilisée telle quelle lors de la génération de code. Dans le cas d'un port matériel, cet attribut est utilisé pour indiquer le protocole utilisé tel que VCI au niveau CABA. Dans le cas d'un port logiciel, cet attribut spécifie le type des données dans le tableau.

6.3.6 Le Concept *ImplementedByConnector*

Les trois concepts présentés *AbstractImplementation*, *Implementation* et *PortImplementation* permettent la description des IP logiciels et matériels et leurs interfaces. Pour faire le lien entre ces IP et les composants élémentaires définis à haut niveau, nous avons proposé le concept *ImplementedByConnector*. Ce dernier joue le rôle d'un connecteur et il se divise en deux types. Un *ImplementedBy* permet de lier le composant à l'implémentation tandis qu'un *PortImplementedBy* permet d'associer un port du composant à un port de l'implémentation.

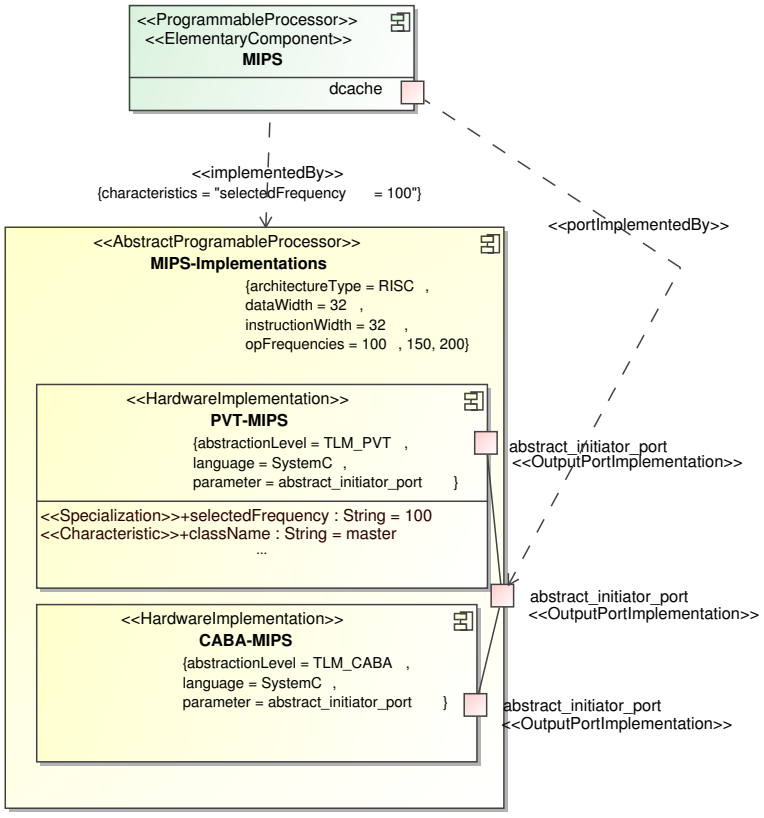


FIG. 6.14 – *Déploiement d'un composant élémentaire en utilisant PortImplementedBy et ImplementedBy*

La figure 6.14 présente un exemple d'usage de ces connecteurs. Le processeur MIPS

élémentaire est déployé sur l'implémentation abstraite *MIPS-Implementations* en utilisant le connecteur *ImplementedBy*. Le connecteur *PortImplementedBy* relie le port du composant élémentaire au port de l'implémentation.

6.3.7 Le Concept *EnergyModel*

Afin de prendre en considération l'estimation de l'énergie dans le code du système généré, un modèle de consommation doit être attribué à chaque composant matériel. Nous avons proposé le concept *EnergyModel* permettant de créer des liens entre les IP et les modèles de consommation développés dans le chapitre précédent. La figure 6.15 illustre le concept *EnergyModel* lié à celui de *HardwareImplementation*. Les modèles de la consommation sont classifiés dans des fichiers par type de composant. Pour créer les liens entre ces fichiers et le code du système généré, un attribut `filePath` à été ajouté à la méta-classe *EnergyModel*. De plus, les modèles de consommation nécessitent les occurrences des activités pertinentes dans chaque composant pour calculer la dissipation à chaque intervalle de temps ou bien à la fin de la simulation. Pour pouvoir lire les valeurs de ces occurrences, nous avons besoin des noms des compteurs insérés dans le composant. Le deuxième attribut `activitiesList` permet de récupérer la liste ordonnée de ces compteurs. De plus, un modèle de consommation nécessite les paramètres d'architecture du composant associé tels que la taille, l'associativité, stratégie de remplacement pour un cache. Ces paramètres seront déduits des spécifications de l'IP auquel le modèle de consommation est lié. Il en est de même pour la technologie de fabrication adoptée, ce paramètre sera déduit de l'attribut `technologyName` de l'*AbstractImplementation* contenant l'IP.

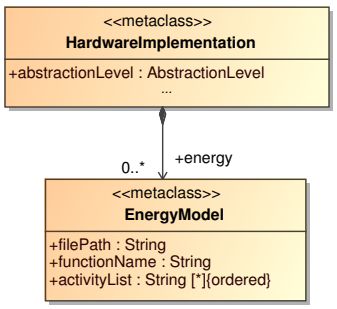


FIG. 6.15 – *Le concept EnergyModel lié à celui de HardwareImplementation*

6.3.8 Le Concept *Characterizable et Specializable*

Afin de permettre au concepteur du système d'ajouter ces propres attributs aux concepts *Implementation* et *PortImplementation*, nous avons défini le concept *Parameter*. Avec ces trois champs `name`, `type`, et `value` il est possible de créer de nouveaux attributs. Le nom du paramètre permet de le différencier des autres. Le champ type indique bien sûr le type de ce paramètre. Enfin, la valeur est spécifiée sous forme de chaîne de caractères. Généralement, les nouveaux attributs ajoutés sont fortement liés à la description de l'IP. Ils sont utiles soit dans la phase de transformations entre modèles, soit dans la phase de compilation. Suivant

l'objectif d'utilisation, nous faisons la distinction entre les attributs. Pour cela deux méta-classes *Characterizable* et *Specializable* sont définies. Ces deux dernières peuvent contenir 0 ou plus *Parameter* [6].

Via la méta-classe *Characterizable*, des paramètres appelés *characteristics* peuvent être créés. Ce type de paramètre a pour but de passer une information particulière à une transformation. A titre d'exemple, nous trouvons la caractéristique `className` dans la figure 6.14. Suivant la valeur d'une *characteristic* un modèle de sortie généré peut changer de valeur. En réalité, nous avons la possibilité de placer ce type de paramètres directement sur le méta-modèle Gaspard mais ils seront utiles que pour la génération d'une cible particulière. A titre d'exemple, au niveau PVT, nous avons besoin d'ajouter un attribut pour les ports matériels afin de spécifier le type d'interface supportée ; bloquant ou non-bloquant. Ce nouvel attribut n'est pas utile pour les autres niveaux d'abstraction, c'est pour cela il n'est pas placé dans le méta-modèle de base.

Via la classe *Specializable*, des paramètres appelés *specializations* peuvent être créés. Ce type de paramètre concerne une implémentation et ces ports correspondants. Ils sont transmis directement au code de l'IP. Lors de la compilation, le code peut utiliser ces informations pour obtenir une version spécialisée de l'IP telle que la spécialisation `selectedFrequency` dans la figure 6.14. En outre, ce type de paramètre est utilisé pour spécifier les informations temporelles des IP matériels telles que les temps d'accès et de cycle d'une mémoire, les latences d'un réseau d'interconnexion, etc.

La manière avec laquelle les *specializations* sont passées au code de l'IP varie en fonction du langage de programmation. Pour chaque langage possible dans Gaspard, il y a une manière définie. En C, cela ce fait par l'utilisation des variables du pré-processeur. Par exemple une *specialization* nommée ACCESS_TIME et ayant pour valeur 20 sera équivalent à avoir ajouté au début du code de l'IP #define ACCESS_TIME 20. En C++, le passage de l'information utilise le mécanisme de template interne au langage. Pour des cas tels que celui-ci, où l'ordre de passage des valeurs importe, les *specializations* sont ordonnées (indiqué par ordered dans le méta-modèle).

6.4 GaspardLib, une bibliothèque de composants pour la modélisation de SoC

La modélisation de haut niveau des IP logiciels et matériels précédemment décrite a été concrétisée par le développement d'une bibliothèque nommée GaspardLib. Ce travail a été fait en collaboration avec Eric Piel [114]. Les composants matériels de SoCLib ainsi que ceux développés dans le cadre de cette thèse sont intégrés dans notre bibliothèque afin de construire l'architecture du système. Pour l'instant nos composants comprennent des processeurs, des caches, un réseau crossbar, un accélérateur matériel, des mémoires, etc. Ils sont tous orientés pour la génération de simulation en SystemC aux niveaux d'abstraction CABA et PVT. Cependant, de nouveaux composants peuvent être ajoutés à notre bibliothèque en respectant les spécifications de haut niveau. Nous allons donner maintenant un exemple concernant l'ajout d'un composant cache au niveau PVT à notre bibliothèque en se basant sur la description du composant. Cette description comporte deux fichiers ; cache.h contenant la définition de la classe SystemC et cache.cc contenant le code. Le texte ci-après montre la déclaration de ce composant. La modélisation dans GaspardLib correspondante est présentée dans la figure 6.16.

```
1  #ifndef MASTER_HEADER
2  #define MASTER_HEADER
3
4  #include "systemc.h"
5  #include "bus_types.h"
6  #include "basic_TLM_PA_initiator_port.h"
7
8  using basic_protocol::basic_timed_initiator_port;
9  using basic_protocol::basic_request_timed;
10 using basic_protocol::basic_response_timed;
11
12 class cache : public sc_module
13 {
14 public:
15     cache( sc_module_name module_name );
16     SC_HAS_PROCESS( cache );
17     basic_timed_initiator_port<ADDRESS_TYPE,DATA_TYPE>
   *initiator_port;
18
19     sc_port<
20       tlm_blocking_get_if <
21         basic_request_timed< ADDRESS_TYPE , DATA_TYPE >>
22     > slave_port;
23
24
25 private:
26   void run();
27   };
28
29 #endif
```

Au début, une fonctionnalité générique appelée *Cache-implementations* a été définie (figure 6.16). Elle est stéréotypée *AbstractCache* pour indiquer que les IP décrits définissent un cache. A l'intérieur de cette implémentation abstraite, nous avons défini l'implémentation en PVT (nommée *PVT-Cache*). Dans la description en SystemC, la classe s'appelle `cache` et le constructeur (du même nom) prend un argument qui est le nom de l'instance indiqué dans le modèle du Système. Cet argument est utile pour l'instanciation du composant dans la phase de génération de code. Il doit être spécifié par la *Characteristic* `className` comme illustré dans la figure 6.16. De plus, le nom de l'instance permet d'obtenir des rapports de performance et de consommation d'énergie ainsi qu'un débogage lisible par l'utilisateur. L'appel à SC_HAS_PROCESS() (ligne 16 dans le code) est spécifique à SystemC et indique que cette classe représente un composant SystemC. Cette caractéristique est spécifiée par l'attribut `language` du concept *HardwareImplementation*. Au moment de la compilation, les paramètres de l'architecture seront insérés de façon automatique dans le fichier cache.h en utilisant la directive #define telle que #define blocSize 128.

A partir de la description du composant, le nom et le type de chaque port sont identifiés afin qu'ils soient modélisés via le concept *PortImplementation*. Ces paramètres sont importants pour la connexion du composant au reste de l'architecture. Les deux fichiers décrivant le composant sont spécifiés en haut niveau par le concept *CodeFile*. Les spécifications temporelles du composant pour l'estimation des performances sont introduites via le stéréotype *Specialization*. Dans notre exemple, nous avons défini les deux paramètres `accessTime` et

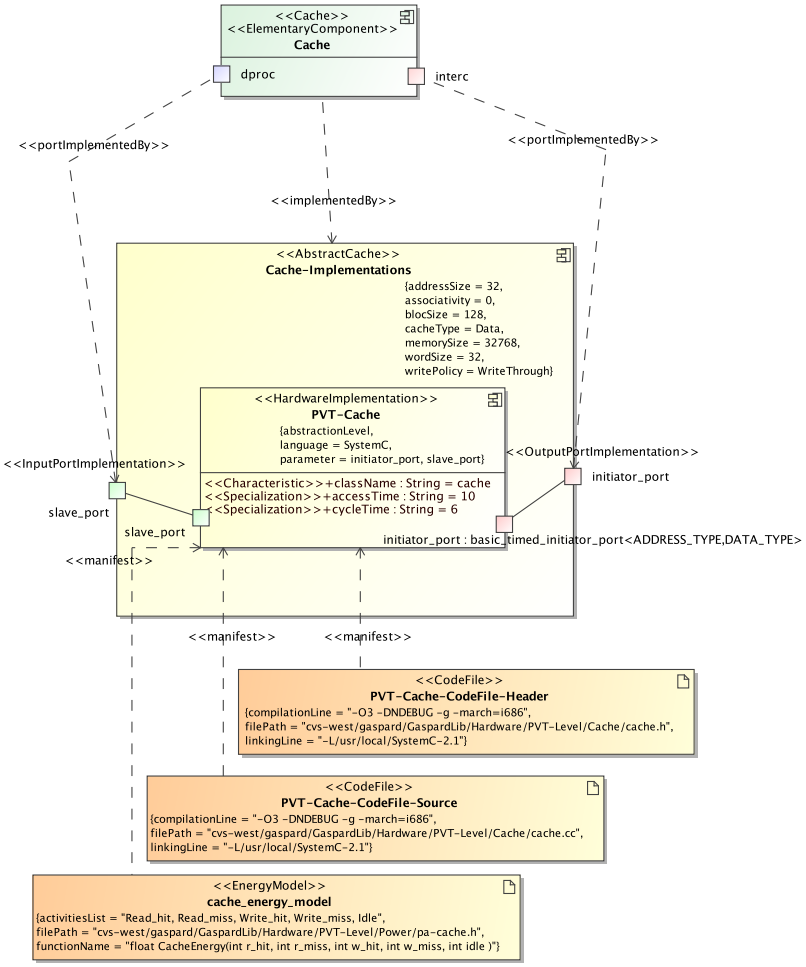


FIG. 6.16 – *Modéliser un nouveau IP cache dans GaspardLib*

`cycleTime` qui seront aussi insérés dans le fichier cache.h de façon automatique. Enfin, nous avons utilisé le concept *EnergyModel* pour associer un modèle de consommation d'énergie à notre composant cache au niveau PVT. En suivant les étapes précédentes, un IP peut être intégré facilement dans notre environnement de conception.

Dans notre bibliothèque, les IP logiciels sont réutilisés de différentes sources à savoir : la bibliothèque C++ GENIAL [72] (GENeric Image Array Library), de la bibliothèque C VSIPL [35] (Vector Signal Image Processing Library) et d'un ensemble de fonctions écrites en C par un partenaire industriel pour le traitement d'image. Parmi les fonctionnalités disponibles, nous trouvons la transformée de Fourier rapide (FFT), les filtres numériques, gradient d'une image en niveaux de gris, etc.

Lorsqu'il s'agit de la même fonctionnalité disponible dans plusieurs sources, le concept d'*AbstractImplementation* est utilisé pour rassembler les différents IP. L'ajout d'un nouvel IP se passe en deux étapes. La première consiste à créer un adaptateur autour de l'appel de l'IP pour qu'il puisse être compatible avec le protocole d'appel standard de Gaspard. La seconde étape consiste à modéliser l'interface de l'adaptateur créé à l'aide du paquetage *Deployment*

du méta-modèle.

A titre d'exemple, la fonctionnalité de calcul de FFT est disponible via l'usage soit de la bibliothèque GENIAL soit de la bibliothèque VSIPL. La figure 6.17 présente la modélisation de la fonctionnalité de FFT dans GaspardLib. L'*AbstractSoftwareImplementation* indique la fonctionnalité dans son ensemble : c'est un IP logiciel avec une entrée et une sortie. Elle contient deux *SoftwareImplementations* qui correspondent à la fonctionnalité dans les bibliothèques VSIPL et GENIAL. Nous trouvons ainsi la description des adaptateurs correspondante à chaque implémentation wrap_vsipl_fft.c et wrap_genial_fft.c codés en C. Le *CodeFile* indique l'emplacement du fichier contenant l'adaptateur. Nous considérons ici que l'utilisateur s'est déjà chargé de compiler la bibliothèque VSIPL ou GENIAL. L'attribut `linkingLine` permettra lors de la liaison du code logiciel d'incorporer la bibliothèque.

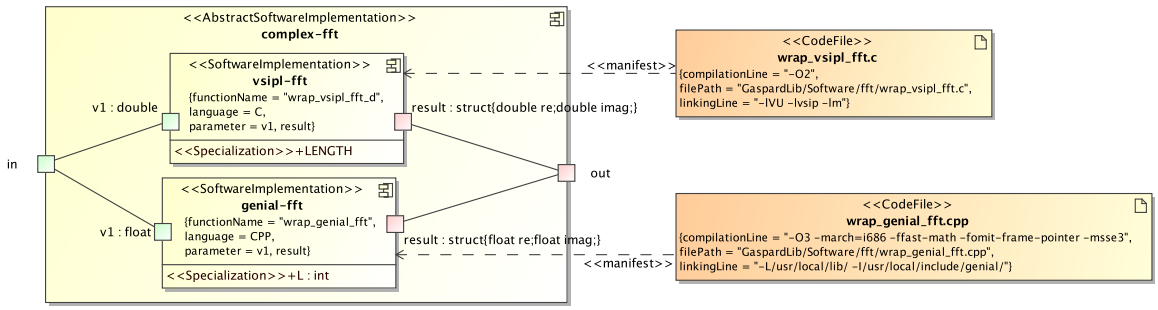


FIG. 6.17 – *Exemple d'IP logiciel de la bibliothèque Gaspard*

6.5 Chaîne de compilation

Au cours des sections précédentes, nous avons introduit d'abord le méta-modèle de base pour la modélisation des MPSoC dans l'environnement Gaspard. Ensuite, nous avons présenté notre contribution au méta-modèle *Deployment* afin que le modèle de haut niveau puisse contenir les informations nécessaires à la génération d'un code exécutable. Dans ce qui suit, nous allons nous attacher à l'obtention de ce code de manière automatique. Cette étape que nous appelons *compilation* est basée sur un pilier fondamental de l'IDM : les transformations de modèles. Le passage du modèle d'origine au résultat final (code exécutable) ne se fait pas en une seule étape. Afin de diviser les difficultés du passage de l'un à l'autre, des modèles intermédiaires sont introduits afin de diminuer la complexité des transformations correspondantes. Ainsi pour parvenir à la cible de compilation, une chaîne de transformations est utilisée. Dans notre environnement Gaspard, plusieurs plateformes cibles sont visées telles qu'une simulation SystemC, une validation formelle en langage synchrone, une version fonctionnelle en OpenMP et une autre pour l'obtention d'accélérateurs matériels en VHDL. La figure 6.18 donne un aperçu sur les chaînes de transformation correspondantes à chaque cible. Cette figure met en valeur un des avantages de l'IDM : la réutilisation des transformations. En effet, il est facile d'assembler les transformations les unes après les autres, ce qui permet de réutiliser les premières transformations pour générer différentes cibles. Dans le cadre de cette thèse, nous allons détailler la chaîne de transformations développée pour la génération du code SystemC à différents niveaux d'abstraction. Comme nous l'avons décrit

dans [5], pour arriver au code SystemC à partir d'un modèle Gaspard, trois transformations sont appliquées à savoir *Polyhedron*, *Loop* et *SystemC*.

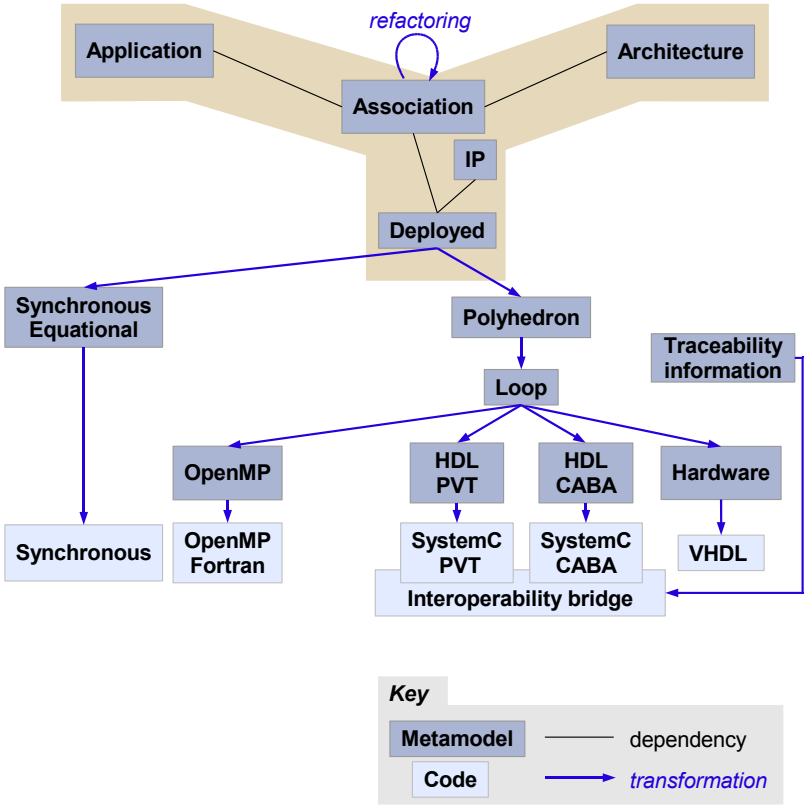


FIG. 6.18 – *Environnement de conception Gaspard*

6.5.1 Méta-modèle Polyhedron

Le méta-modèle Polyhedron est développé dans la thèse d'Eric Piel [114]. Le but de ce méta-modèle est de transformer la notion d'association dans le modèle de haut niveau. Les tâches sont représentées directement sur les processeurs et les tableaux de données sont spécifiés sur les mémoires. Pour représenter sans perte d'information la distribution des données ou des tâches, le concept mathématique de polyèdre entier paramétré est employé. L'usage de ces polyèdres permet d'exprimer l'espace de répétition lorsque la tâche est distribuée sur plusieurs processeurs. De cette caractéristique dérive le nom du méta-modèle. En plus de cela, les notions de déploiement ont été largement simplifiées, de manière à ce que chaque composant élémentaire soit lié directement à l'IP qui sera utilisé lors de la génération de code.

La génération du polyèdre se fait en transformant les instances de composants applicatifs. A partir d'une instance, il est possible de retrouver sa *Shape*, son espace de répétition, et la *Distribution* qui lui est appliquée. Depuis cette *Distribution*, nous pouvons retrouver directement l'ensemble de processeurs sur lesquels la tâche est placée et indirectement on

retrouve sa *Shape*. Ce sont toutes les informations nécessaires à la création du polyèdre. Pour résumer, voici les informations disponibles :

- sourceTiler (O_{sw}, P_{sw}, F_{sw})
- repetitionSpace, patternShape
- targetTiler (O_{hw}, P_{hw}, F_{hw})
- espace de répétition de l'application (m_{sw})
- espace de répétition des processeurs (m_{hw})

L'usage du polyèdre doit permettre de déterminer les points $\overrightarrow{T_p}$, faisant partie de l'espace de répétition de l'application en fonction de l'indice du processeur $\overrightarrow{P}$, appartenant à l'espace de répétition des processeurs.

Une fois ces données posées, il suffit de fusionner l'ensemble des définitions concernant ces concepts : les définitions des *Tilers*, la définition de l'espace de répétition de la distribution, celle de la forme du motif, et enfin les définitions des espaces de répétitions de l'application et des processeurs. Nous obtenons alors le système d'équations et d'inéquations suivant :

$$\begin{cases} \overrightarrow{T_p} + m_{sw}.\overrightarrow{Q1} = O_{sw} + P_{sw}.\overrightarrow{Xq} + F_{sw}.\overrightarrow{Xd} \\ \overrightarrow{P} + m_{hw}.\overrightarrow{Q2} = O_{hw} + P_{hw}.\overrightarrow{Xq} + F_{hw}.\overrightarrow{Xd} \\ 0 \leq \overrightarrow{Xq} < repetitionSpace \\ 0 \leq \overrightarrow{Xd} < patternShape \\ 0 \leq \overrightarrow{P} < m_{sw} \\ 0 \leq \overrightarrow{T_p} < m_{hw} \end{cases}$$

Où les deux variables supplémentaires $Q1$ et $Q2$ permettent de représenter la notion de modulo des *Tilers*. Aucune contrainte ne portant sur ces variables, elles peuvent prendre n'importe quelles valeurs entières.

Toutes ces équations et inéquations sont affines. Une fois répliquées pour représenter chaque dimension des vecteurs, on obtient le polyèdre recherché. Notons au passage, que cette construction permet de démontrer qu'il est possible de trouver un polyèdre pour n'importe quelle *Distribution*.

Nous allons présenter maintenant un exemple de calcul de polyèdre correspondant à l'espace de répétition de la tâche décrite dans la figure 6.19. La distribution de cette tâche répétée 16×16 est répartie également sur chacun des quatre processeurs. La figure 6.20 illustre cette distribution en associant une couleur différente à chaque indice de tâche selon le processeur attribué.

En construisant le polyèdre selon la méthode définie précédemment on obtient ce sys-

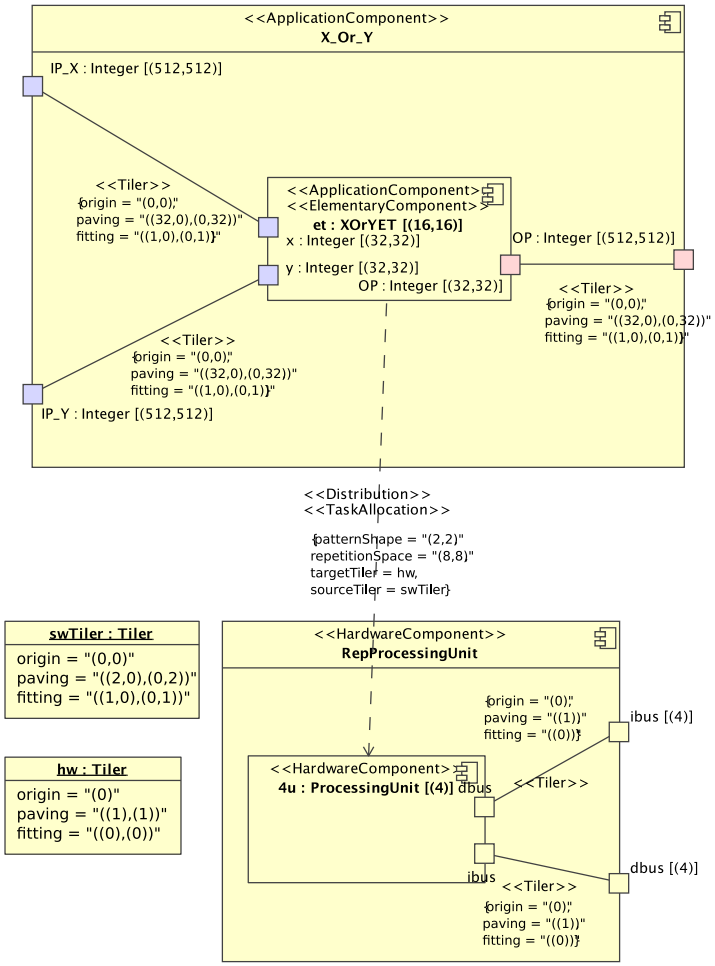


FIG. 6.19 – *Exemple de distribution d'une tâche répétée* 16×16 *sur quatre processeurs.*

tème :

$$\begin{cases} p_0 \leq 0,3 - p_0 \leq 0 \\ -4 * mh_0 - p_0 + 1 * q_0 + 1 * q_1 + 0 * d_0 + 0 * d_1 + 0 = 0 \\ -16 * ms_0 - x_0 + 2 * q_0 + 0 * q_1 + 1 * d_0 + 0 * d_1 + 0 = 0 \\ -16 * ms_1 - x_1 + 0 * q_0 + 2 * q_1 + 0 * d_0 + 1 * d_1 + 0 = 0 \\ q_0 \leq 0,7 - q_0 \leq 0 \\ q_1 \leq 0,7 - q_1 \leq 0 \\ d_0 \leq 0,1 - d_0 \leq 0 \\ d_1 \leq 0,1 - d_1 \leq 0 \\ x_0 \leq 0,15 - x_0 \leq 0 \\ x_1 \leq 0,15 - x_1 \leq 0 \end{cases}$$

Pour générer le modèle *Polyhedron* à partir du modèle *Deployed*, une transformation de modèles est utilisée. Cette étape nécessite la définition d'un ensemble de règles permettant de transformer les patrons d'entrée en patrons de sortie. Pour pouvoir exécuter les règles,

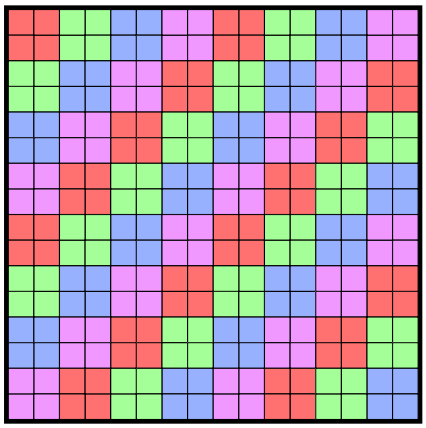


FIG. 6.20 – *Exemple de distribution d'une tâche répétée* 16×16 *sur quatre processeurs*

nous avons utilisé le moteur de transformation MoMoTE (Model to Model Transformation Engine) développé dans notre équipe. Cet outil est utilisé comme une API Java, et repose sur les bibliothèques Ecore et EMFT Query [43] pour manipuler les modèles et exprimer les patrons d'entrée et de sortie des règles. MoMoTE a l'avantage de permettre de mélanger les approches de programmation impérative et déclarative. En particulier, dans la partie logique d'une règle il est possible de faire appel à des bibliothèques ou des programmes extérieurs, vus comme des boîtes noires.

6.5.2 Méta-modèle Loop

Le méta-modèle *Loop* est le second méta-modèle intermédiaire. Développé par Julien Taillard, doctorant dans l'équipe, il est très proche du méta-modèle Polyhedron. Au lieu d'utiliser un polyèdre pour représenter la répétition des tâches, nous utilisons un *LoopStatement* pour exprimer cette répétition. Un *LoopStatement* correspond à la structure du pseudo-code qui, pour un indice de processeur donné, parcourt tous les indices de répétitions de la tâche associés. Cette représentation est très proche du code final généré. Dans notre chaîne de transformations, le passage d'une représentation à l'aide d'un polyèdre à celle sous forme de pseudo-code est effectué par l'outil externe CLooG [17].

La figure 6.21 présente le détail d'un LoopStatement. Il est constitué d'une suite ordonnée de Blocs, eux-mêmes pouvant contenir une suite ordonnée de Blocs. Un Bloc représente une instruction du pseudo-code, qui peut être soit un If (une condition), un For (une boucle), ou un *Assignement* (l'affectation d'une valeur à une variable). Chaque attribut de ces Blocs contient une expression arithmétique ou une expression de comparaison en pseudo-code. Les variables utilisées sont les mêmes que dans les polyèdres du méta-modèle Polyhedron. Chaque feuille de l'arborescence des Blocs correspond à des points du code où les variables x représentent l'indice d'une répétition de la tâche.

Le pseudo-code de la figure 6.22 est généré à partir du modèle *Polyhedron* en utilisant une transformation de modèles.

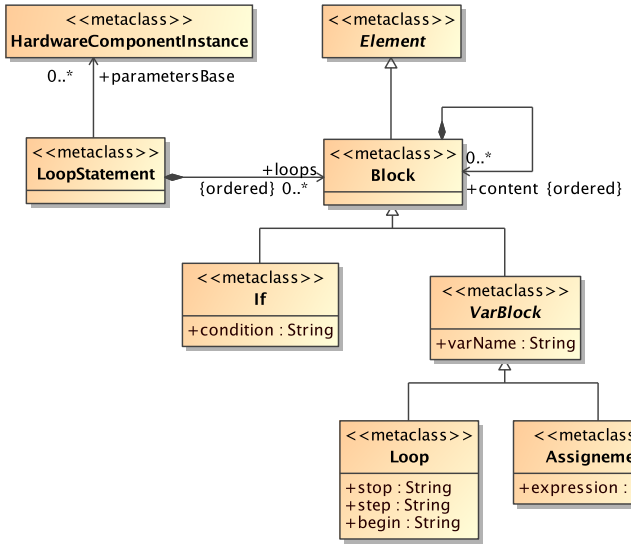


FIG. 6.21 – *Les concepts utilisés par le méta-modèle Loop pour représenter la structure de code correspondante à un polyèdre.*

6.6 Génération de code SystemC

Après le passage des deux transformations *Polyhedron* et *Loop*, nous considérons que le modèle du système MPSoC est suffisamment proche pour la génération du code SystemC. Pour cela, nous avons développé une dernière transformation pour atteindre notre objectif. Cette transformation est de type modèle-vers-texte, elle consiste à générer du code pour les deux parties matérielle et logicielle. Les étapes à suivre au cours de cette transformation dépendent du niveau d'abstraction projeté. Pour le niveau d'abstraction PVT-PA, nous avons besoin de générer les processeurs qui réalisent les tâches de l'application avant d'instancier les composants de l'architecture. Pour les autres niveaux d'abstraction, l'architecture peut être instanciée en premier. En second lieu, le code de l'application sera généré, compilé, ensuite placé dans les mémoires d'instructions et de données.

La partie de cette transformation générant le code de l'application a été réalisée par Eric Piel. Elle est décrite en détail dans sa thèse [114], nous ne la traiterons que brièvement dans notre document. Tandis que la génération de code de la partie matérielle sera présentée en détail. Avant tous, nous allons présenter l'outil de transformation du modèle *Loop* vers le code SystemC.

6.6.1 Moteur de transformation modèle-vers-texte

Selon les principes de l'IDM, pour effectuer la transformation d'un modèle en texte il faut que le modèle soit aussi proche que possible des concepts présents dans le texte généré. Dans notre travail, cette recommandation a été prise en considération. De ce fait, la transformation du modèle *Loop* vers le code SystemC ne doit pas modifier les concepts mais uniquement les traduire *littéralement* en un-vers-un.

La plupart des moteurs de transformations existants supportent ce type de transfor-

```
for (d1=0; d1<1; d1++){
 for (q1=0; q1<7; q1++){
  for (x1=0; x1<7; x1++){
   for (d0=MAX(0,-14); d0<MIN(1,15); d0++){
    q0 = -q1+4*mh0+p0
    if (MOD(d1+2*q1-x1,16) == 0){
        ms1 = (d1+2*q1-x1)/16
        x0 = -2*q1+d0+8*mh0+2*p0
        ! Call to the task
    }
   }
  }
 }
}
```

FIG. 6.22 – *Pseudo-code généré à partir du modèle Polyhedron*

mation. Probablement, le fait qu'elle soit de type un-vers-un la rend facile à implémenter. L'équipe a choisi JET [44] (Java Emitter Templates) pour la transformation modèle-vers-texte. En plus de son intégration facile avec l'environnement Eclipse, il permet de mixer la notion de patron au langage Java ce qui lui offre une grande possibilité d'expression. Nous n'utilisons pas directement JET, mais via l'utilisation d'une couche supplémentaire qui effectue le lien avec Ecore, la technologie de modèle d'Eclipse. Développée par l'équipe, cette couche est nommée MoCodE (Model to Code Engine), elle est spécialement dédiée à la génération de code source depuis des modèles.

Le principe de base de MoCodE est d'associer à chaque type d'élément du méta-modèle d'entrée un patron de code source. Au départ de l'exécution, seule la racine du modèle est transformée. C'est au code présent dans le patron de parcourir le modèle et de solliciter la transformation des éléments souhaitée (à l'aide de la fonction `generate`). Ainsi, au fur et à mesure, les patrons parcourent chacun un petit bout du modèle et appellent d'autres patrons. Le moteur de transformation détermine le patron correspondant à un élément en recherchant le patron ayant le même nom que le type de l'élément transformé. S'il n'existe pas de patron spécifique au type, alors le moteur remonte l'héritage de classe jusqu'à trouver un patron existant.

Par exemple si la génération d'un élément *SoftwareImplementation* est demandée, MoCodE recherchera un patron nommé `SoftwareImplementation`, puis cherchera un patron correspondant à son père, nommé `Implementation` et enfin à son grand-père, `NamedElement`. L'écriture du patron est inspirée des technologies web PHP[7] et JSP[8] : par défaut le texte présent dans le patron est directement écrit dans le fichier de sortie. Il existe des délimiteurs spéciaux (<% et %>) qui indiquent l'appel à du code Java. Lors de l'exécution, le texte généré par le code est inséré à l'emplacement du code. De plus, il est également possible d'ajouter à la transformation des bibliothèques de fonctions purement en Java qui pourront être appelées depuis n'importe quel patron.

Lors de la génération de code, il est souvent nécessaire de produire différents types de fichiers : les fichiers de code, les fichiers d'en-tête, les fichiers de compilation (tels que le

[7]http ://php.net/manual/
[8]http ://java.sun.com/products/jsp/

`Makefile`), etc. MoCodE permet de traiter chaque type de fichier comme une génération de code séparée, il suffit pour cela de placer les patrons dans des paquetages différents. Pour chaque paquetage, c'est-à-dire chaque type de fichiers de sortie, la génération est entièrement relancée en partant de la racine du modèle.

Par exemple, le patron suivant permet de traiter les composants de type *Hardware* du méta-modèle *Loop* en partant de la racine du modèle :

```
1  <%@ jet package="generated.inc"
2  imports="java.util.* org.eclipse.emf.ecore.EObject mocode.engine.*
3           gaspard2.metamodel.deployedloop.* gensysc.*"
4  class="SharedMemoryLoopModel"
5
6  mocode.engine.Engine engine =
   ((mocode.engine.Argument) argument).getEngine();
7  gaspard2.metamodel.deployedloop.SharedMemoryLoopModel element
   =(gaspard2.metamodel.deployedloop.SharedMemoryLoopModel)
   ((mocode.engine.Argument) argument).getElement();
8  %>
9
10 <% for (Component c : (List<Component>) element.getComponents())
11
12 {
13 // Only the Hardware, not the subtypes
14 if(c.eClass().isSuperTypeOf(Deployedloop.eINSTANCE.getHardware())) {
15 %>
16          <%=engine.generate(c)%>
17 <%
18          }
19 }%>
```

Dans cet exemple, les 4 premières lignes indiquent que le nom du patron est *SharedMemoryLoopModel*, il permet de récupérer la racine du modèle, et fait partie du paquetage `generated.inc` qui correspond à la génération de fichiers `.h`. Les lignes 6 et 7 suivantes permettent de récupérer les arguments passés au patron. Le premier argument est toujours l'élément sur lequel le patron travaille. La ligne 10 permet de récupérer tous les éléments de type *Component* dans le modèle. La condition `if` dans la ligne 14 permet de traiter que les instances de type *Hardware*.

6.6.2 Génération du code de la partie matérielle

En commençant à partir de la racine du modèle *Loop*, la génération de code de la partie matérielle opère seulement sur les composants stéréotypés *HardwareComponent*. Jusqu'à cette phase de conception, nous avons gardé la représentation compacte de notre système. De ce fait, l'architecture matérielle contient des instances de composants à différentes structures élémentaires, composées et répétitives. Nous avons défini un patron pour chaque structure afin qu'elle soit traitée différemment par le moteur MoCodE. Ce dernier se charge de générer les fichiers de code (`.cc`) et d'en-tête (`.h`) nécessaires. En se basant sur l'architecture matérielle de la figure 6.23 comme exemple, nous détaillons la génération de code pour chaque structure de composant.

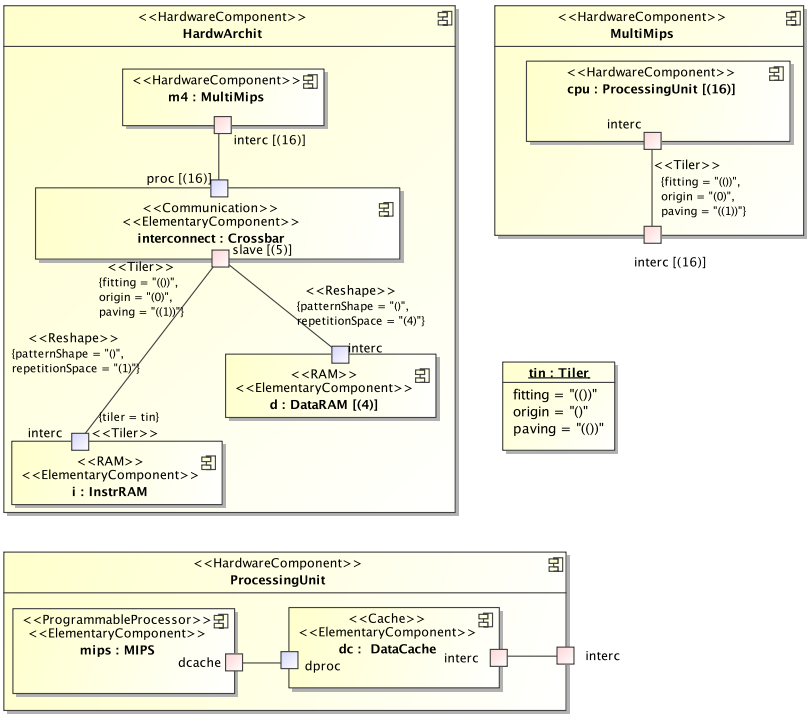


FIG. 6.23 – *Architecture matérielle du MPSoC en modèle Gaspard*

6.6.2.1 structure élémentaire

Pour un composant de structure élémentaire (stéréotypé *ElementaryComponent*), nous avons besoin de transmettre directement au code de l'IP les informations liées à l'architecture et à l'estimation de performance. MoCodE se réfère tout d'abord sur l'attribut `filePath` du *CodeFile* pour éditer le fichier de description de l'IP. Il récupère ensuite les attributs de l'*AbstractHardwareComponent* ainsi que ceux stéréotypés *specializations* à partir du modèle déployé. Enfin, des directives de pré-processeur correspondant à ces attributs et leurs valeurs seront insérées dans le code. Prenons comme exemple le composant élémentaire *dc* instance de *DataCache*. Le déploiement de ce composant a été présenté dans la figure 6.16 (page 154). MoCodE récupère le chemin d'accès du fichier décrivant l'IP afin de l'éditer (`cvs-west/gaspard/GaspardLib/Hardware/PVT-Level/Cache/cache.h`). Ensuite, il ajoute l'ensemble des directives à l'en-tête du fichier qui sera sauvegardé sous un nouveau nom, celui qui est spécifié dans le modèle (`DataCache.h`). Le texte ci-après montre les directives ajoutées dans le code du composant.

```
1 #define addressSize 32
2 #define associativity 0
3 #define blocSize 128
4 #define cacheType Data
5 #define memorySize 32768
6 #define wordSize 32
7 #define accessTime 10
8 #define cycleTime 6
```

6.6.2.2 structure composée ou répétitive

Les composants de structure composée et répétitive dans l'architecture sont réellement inexistants. Ils sont considérés comme des composants pères contenant plusieurs sous-composants. Ces derniers peuvent avoir eux-mêmes différentes structures, ainsi la hiérarchie de l'architecture est obtenue. Les composants de structure composée et répétitive seront générés automatiquement par MoCodE via la création d'un fichier de code et un autre d'en-tête. Dans ce qui suit nous détaillons les étapes nécessaires pour obtenir chaque type de fichier. Pour générer les fichiers d'en-tête, nous précédons comme suit :

1. Créer un fichier d'en-tête dont le nom est celui du composant père. A titre d'exemple pour le composant *HardwArchit*, le fichier `HardwArchit.h` est créé.
2. Inclure les instances des sous-composants dans le fichier d'en-tête du composant père en utilisant la directive `include`. Pour le composant *HardwArchit*, au début du fichier `HardwArchit.h`, 4 fichiers seront inclus à savoir : `MultiMips.h`, `Crossbar.h`, `DataRam.h` et `InstRam.h` représentant respectivement un composant composé de plusieurs unités de calcul, le réseau d'interconnexion, la mémoire de données et la mémoire d'instructions. Cette dernière n'est pas utilisée dans le cas du niveau PVT-PA. Le fichier `systemc.h` doit être toujours inclue puisque nous utilisons la bibliothèque SystemC. Le texte ci-après montre les fichiers inclus dans le code du composant.

```
#include "systemc.h"
#include "Crossbar.h"
#include "MultiMips.h"
#include "DataRam.h"
#include "InstRam.h"
```

3. Définir une nouvelle classe de type SC_MODULE dont le nom est celui du composant père. Pour le composant *HardwArchit*, la classe est définie par le code suivant :

```
class HardwArchit:
public sc_module
{
public:
// Constructeur de la classe
HardwArchit(sc_module_name module_name, int index = 0);
...
};
```

4. Déclarer les instances des sous-composants contenus dans le composant père. Deux paramètres sont nécessaires pour chaque sous-composant : le nom de la classe et la *shape*. Dans le cas d'une structure élémentaire, le nom de la classe est déduit à partir de l'attribut `className` de l'IP. Ceci est le cas des sous-composants *DataRAM*, *InstRAM* et *Crossbar* de *HardwArchit*. A l'opposé, dans le cas d'une structure composée ou répétitive, le nom de la classe est celui du modèle du composant de haut niveau. Ceci est le cas du sous-composant *MultiMips*. En respectant le standard C++, les sous-composants de structure répétitive sont déclarés sous forme de tableau de pointeurs. La dimension de ce dernier est déduite à partir de la *shape* du sous-composant.

```
MultiMips  *MultiMips_pointer;
mem_data  *DataRAM_pointer[4];
mem_inst  *InstRAM_pointer;
simple_crossbar<16,5>  *Crossbar_pointer;
```

5. Déclarer les ports du composant père. Cette étape nécessite la spécification des paramètres suivants pour chaque port : le nom, la *shape*, la direction et le type. Les trois premiers paramètres sont déduits directement à partir du modèle. Pour déterminer le type du port d'un composant père, il faut tout d'abord descendre dans la hiérarchie jusqu'à retrouver le port de l'instance de structure élémentaire auquel il est relié. Ensuite, le type de ce port final est attribué au port du composant père. Ce paramètre est déduit à partir du modèle déployé via l'attribut `type` du concept *PortImplementation*. Dans notre exemple de la figure 6.23, le composant *HardwArchit* ne possède pas de port car il représente le composant principal englobant l'architecture. En prenant l'exemple du composant *MultiMips*, nous remarquons qu'à travers deux niveaux de hiérarchie son port *interc* est relié au port *interc* du composant élémentaire *DataCache*. Donc, le premier port possède le même type que le second et il est déclaré comme suit :

```
// Declaration au niveau PVT
basic_initiator_port<ADDRESS_TYPE,DATA_TYPE> *interc[16];
// Declaration au niveau CABA
ADVANCED_VCI_INITIATOR<32,4,0>  *interc[16];
```

6. Déclarer les liens de communication décrits à l'intérieur du composant père. Ces liens permettent de relier les ports du composant père et les ports instances des sous-composants. Ils sont représentés dans notre modèle Gaspard en utilisant le concept de *Connector*. Ce dernier peut avoir une structure simple ou bien stéréotypé *Tiler* ou *Reshape*. Dans les deux derniers cas, nous nous basons sur les spécifications du *patternShape* et *repetitionSpace* dans le modèle afin de déterminer la dimension du connecteur. Pour la déclaration des liens de communication, nous avons besoin aussi de spécifier le type des connecteurs. Ce paramètre est déduit à partir du type du port d'entrée et celui de sortie liés par le connecteur. C'est à ce niveau que notre transformation vérifie l'interopérabilité entre les ports et décide de la nécessité d'utiliser des adaptateurs de communication. L'exemple suivant montre l'ensemble des connecteurs à déclarer pour le composant *HardwArchit* en prenant une simulation au niveau PVT comme cible. Nous utilisons des canaux de communication (*Channels*) bidirectionnels pour relier les différents ports.

```
typedef tlm_transport_channel<
                    basic_request_timed<ADDRESS_TYPE, DATA_TYPE>,
                    basic_response_timed<DATA_TYPE>
                    > channel_type;
channel_type *Reshape_91a829[16];
channel_type *Reshape_b73194[4];
channel_type *Reshape_1180cb[1];
```

Pour générer les fichiers de code, nous précédons comme suit :

1. Créer un fichier de code dont le nom est celui du composant père et inclure le fichier d'en-tête correspondant. A titre d'exemple pour le composant *HardwArchit*, le fichier `HardwArchit.cc` est créé et le fichier `HardwArchit.h` décrit précédemment est inclue.
2. Instancier les sous-composants contenus dans le composant père. Cette étape se fait par l'usage d'un `new` suivi du nom de la classe. Les composants de structure répétitive sont déroulés selon l'espace de répétition défini par la *shape*. Le texte ci-après montre l'instanciation des sous-composants de *HardwArchit*.

```
HardwArchit::HardwArchit( sc_module_name module_name, int index):
sc_module(module_name)
{
MultiMips_pointer = new MultiMips("MultiMips");
for(int i=0; i<4; i++){
DataRAM_pointer[i] = new mem_data("mem_data");}
InstRAM_pointer = new mem_inst("mem_inst");
Crossbar_pointer = new simple_crossbar<16,5> ("crossbar");
...
}
```

3. Instancier les ports du composant père. De façon similaire aux sous-composants, les ports des composants ainsi que les connecteurs sont déclarés sous forme d'objets dans notre code générés. Par la suite, nous utilisons l'instruction `new` suivi du type pour instancier ces éléments. Au cours de cette instanciation, un espace de répétition est défini pour les ports possédant une *shape*. L'exemple suivant montre l'instanciation du port *interc* de *MultiMips* au niveau PVT.

```
for(int i=0; i<16; i++){
interc[i]= new basic_initiator_port<ADDRESS_TYPE,DATA_TYPE>("port");
}
```

4. Instancier les connecteurs de façon similaire aux ports. Le texte suivant montre l'instanciation des trois connecteurs dans le composant *HardwArchit* en tenant compte de l'espace de répétition de chacun.

```
        for(int j=0; j<16;j++){
                Reshape_91a829[j]= new channel_type("reshape");
        }
        for(int j=0; j<4;j++){
                Reshape_b73194[j]= new channel_type("reshape");
        }
        for(int j=0; j<1;j++){
                Reshape_1180cb[j]= new channel_type("reshape");
        }
```

5. Établir les connexions entre les différents ports à l'intérieur du composant père. Cette connexion est de type un-vers-un pour une structure de connecteur simple. Cependant, pour des connecteurs stéréotypés *Tiler* ou *Reshape*, un calcul des coordonnées de chaque indice de port d'entrée pour retrouver son correspondant en sortie est nécessaire. Ce calcul se base principalement sur l'équation 6.3 présentée dans la page 138. Le code généré du *Tiler* est constitué d'abord de la déclaration sous forme de constantes des caractéristiques du *Tiler* (origine, pavage, ajustage). Cette déclaration est conditionnée par le fait que la caractéristique est différente de la valeur nulle. Ensuite une boucle d'itération parcourant l'espace de répétition (*repetitionSpace*) et la *shape* du port (*patternShape*) est insérée dans le code. Elle permet de calculer pour chaque indice du port de départ son correspond à l'arrivée. Dans le cas d'un *Reshape*, le code généré correspond au traitement de deux *Tilers*, un (*Tiler_in* et un *Tiler_out*) comme il le montre le texte suivant. Dans cet exemple, nous traitons le calcul du *reshape* connectant le banc mémoires *DataRam* au *Crossbar*.

```
// Variables generales
```

```
int reptitionSpaceI_Reshape[1];
int arrayInI_Reshape[1];
int arrayOutI_Reshape[1];
int repetitionSpace_Reshape[1] = {4} ;
int arrayIn_Reshape[1] = {4} ;
int arrayOut_Reshape[1] = {5} ;

// Caracteristiques du Tiler_In
int originMatrix_In_Reshape[1]={0};
int pavingMatrix_In_Reshape[1][1] = {{1}};
// Caracteristiques du Tiler_Out
int originMatrix_Out_Reshape[1]={0};
int pavingMatrix_Out_Reshape[1][1] = {{1}};

// Boucle d'iteration parcourant l'espace de repetition
for(reptitionSpaceI_Reshape[0]=0; reptitionSpaceI_Reshape[0]<
repetitionSpace_Reshape[0]; reptitionSpaceI_Reshape[0]++){
    // Calcul du Tiler_In
    for(int i = 0 ; i < 1 ; i++) {
        // Origin
        arrayInI_Reshape[i] = originMatrix_In_Reshape[i];
        // Paving
        for(int j = 0 ; j < 1 ; j++)
         arrayInI_Reshape[i]+= reptitionSpaceI_Reshape[j] *
         pavingMatrix_In_Reshape[i][j];
                }
    // Calcul du Tiler_In
        for(int i = 0 ; i < 1 ; i++) {
        // Origin
        arrayOutI_Reshape[i] = originMatrix_Out_Reshape[i];
        // Paving
        for(int j = 0 ; j < 1 ; j++)
         arrayOutI_Reshape[i]+= reptitionSpaceI_Reshape[j] *
         pavingMatrix_Out_Reshape[i][j];
                }
// Connexion des ports via les connecteurs
Crossbar_pointer->slave[arrayInI_Reshape[0]]->bind
            (Reshape[reptitionSpaceI_Reshape[0]]->target_export);
DataRAM_pointer->interc[arrayOutI_Reshape[0]]->bind
            (Reshape[reptitionSpaceI_Reshape[0]]->slave_export);
}
```

6.6.3 Génération du code de la partie logicielle

La génération du code de la partie logicielle diffère d'un niveau d'abstraction à un autre. Néanmoins, cette différence ne concerne que la réutilisation du code généré dans la simulation logicielle/matérielle. Au niveau PVT-PA, tout le code s'exécutant sur un processeur est contenu dans une seule classe, c'est-à-dire dans un fichier source et un fichier d'en-tête. Cette classe contient la description SystemC du module processeur, les tâches exécutées par le processeur, et l'ordonnanceur dynamique des tâches comme il a été décrit dans le chapitre 4. L'intégration de ce module processeur avec le reste des composants de l'architecture nous

permet une simulation logicielle/matérielle de notre système. Cependant, dans les autres niveaux plus bas, le code de la partie logicielle représente l'ensemble des tâches à exécuter pour chaque processeur. A ces niveaux, les variables de synchronisation entre les tâches sont déclarées sous forme logicielle de façon différente du premier cas où elles sont déclarées au niveau du simulateur. De plus, le code généré sera compilé afin qu'il soit exécuté par les processeurs à partir de la mémoire.

La génération des tâches se fait par l'appel récursif du patron traitant les *ApplicationComponents*. Ce patron permet de générer les boucles de l'espace de répétition des tâches dépendant de l'index du processeur. La génération commence du niveau de hiérarchie le plus haut pour arriver au niveau le plus bas, et donc l'appel des IP. A ce niveau, nous générons tout d'abord du code permettant le calcul d'adresses des données avant et après l'instanciation de la fonction. Ce calcul est basé principalement sur les caractéristiques des *Tilers*. Les transferts de données sont exprimés de manière explicite en utilisant des méthodes de communication de type *read()* et *write()*. Ensuite, notre transformation insère la déclaration conforme au protocole d'appel à l'IP : le nom de la fonction à appeler et le type de chaque paramètre. Ces informations seront déduites du modèle de l'IP déployé décrit dans une *SoftwareImplementation*. En plus de cela, le patron génère également la création de chacune des synchronisations utilisées par les tâches.

6.6.4 Création d'un Makefile

Afin de permettre à l'utilisateur de facilement compiler et exécuter le simulateur, il est nécessaire de générer en plus des fichiers sources un fichier Makefile. Ce fichier, usuel dans le cadre de développement UNIX, permet d'automatiser entièrement la compilation et l'édition de liens des fichiers du simulateur. Après l'exécution de la commande, l'utilisateur peut lancer la simulation en exécutant le programme généré appelé TLMRun.

Dans la génération, une grande partie du fichier Makefile est statique : les commandes permettant de compiler chaque type fichier supporté (actuellement SystemC, C++, et C), les commandes pour effectuer l'édition de lien, et la liste des fichiers à compiler. Dans l'exemple présenté ci-après cela correspond aux définitions entre la ligne 10 et la fin. Le reste du fichier généré permet de compiler les fichiers sources des IP. Pour chaque fichier l'attribut `compilationLine` du *CodeFile* le représentant est utilisé pour compléter les arguments passés au compilateur. Les `linkingLines` de tous les *CodeFiles* du modèle sont concaténées et passées en argument de l'éditeur de lien. Voici un exemple simplifié de fichier généré :

```
1  ## Ajouter les lignes de compilation pour chaque fichier
2  cache.o : CFLAGS_CODE =-O3 -g -march=i686
3  ...
4  LDFLAGS_CODE = -lnetpbm
5
6  ## Chemin d'acces des bibliotheques
7  LIBS = -L$(SYSTEMC)/lib-$(TARGET_ARCH) -L$(TLM_PA_LIB)
8  CFLAGS = -g -Wall $(INCDIR) $(CFLAGS_CODE)
9  LDFLAGS = $(LDFLAGS_CODE) -lsystemc  -lpthread
10
11 ## Tous les fichiers sources ont besoin d'etre compiles
12 SRCS = $(wildcard ./*.cc) $(wildcard ./*.c)
```

```
13 OBJS = $(patsubst %.c,%.o,$(SRCS:%.cc=%.o))
14
15 all:    Make.dep TLMrun
16
17 %.o: %.c
18         $(GX) $(CFLAGS) -c $< -o $@
19
20 %.o: %.cc
21         $(GXX) $(CFLAGS) -c $< -o $@
22
23 TLMrun: $(OBJS)
24         $(GXX) -o TLMrun $(LIBS) $(OBJS) $(LDFLAGS)
```

6.7 Conclusion

Dans ce chapitre nous avons présenté l'utilisation de l'IDM pour résoudre la complexité de conception des systèmes MPSoC. En partant du modèle Gaspard avec les spécifications du déploiement, des transformations de modèles successives permettent d'obtenir une simulation à différents niveaux d'abstraction. La compilation est faite en passant à travers trois transformations successives qui ont chacune pour but de se rapprocher de plus en plus du code de simulation. Les deux premières transformations sont communes aux autres chaînes de transformations de l'environnement Gaspard. Elles sont reliées par le méta-modèle intermédiaire *Polyhedron*. Ceci est considéré comme un avantage essentiel dans l'approche IDM. A partir de là, une dernière transformation, de modèle-vers-texte, est capable de générer l'ensemble de fichiers complets, directement compilables pour produire le simulateur exécutable.

Dans le cadre des systèmes embarqués où la technologie évolue très vite, la flexibilité du compilateur qui s'adapte à cette tendance est considérée comme un atout important. Dans l'approche IDM, le processus de transformation de modèles souffre encore d'une technologie trop jeune. Actuellement, nous sommes en face d'un seul standard pour l'écriture de transformations : QVT [101] et les moteurs de transformations existants ne sont pas pleinement matures. De ce fait, notre équipe s'est chargée à développer ses propres moteurs de transformation et de génération de code (MoMoTE et MoCodE). Par ailleurs, la spécification séparée des méta-modèles est considérée comme une limitation de l'approche IDM. Pour deux méta-modèles très proches, le développeur doit les spécifier entièrement séparément, ce qui contraint à devoir écrire de nombreuses règles de un-vers-un. La transformation de *Polyhedron* à *Loop* en est un exemple. Une amélioration de la méthodologie serait de pouvoir spécifier un méta-modèle intermédiaire uniquement par la redéfinition d'un sous-ensemble de méta-modèle de référence. Dans le chapitre suivant nous allons, à l'aide d'une étude de cas, valider le fonctionnement de la chaîne de transformations via la génération automatique d'une simulation au niveau d'abstraction PVT-PA.

Chapitre 7

Étude de cas et validation expérimentale

7.1 Introduction

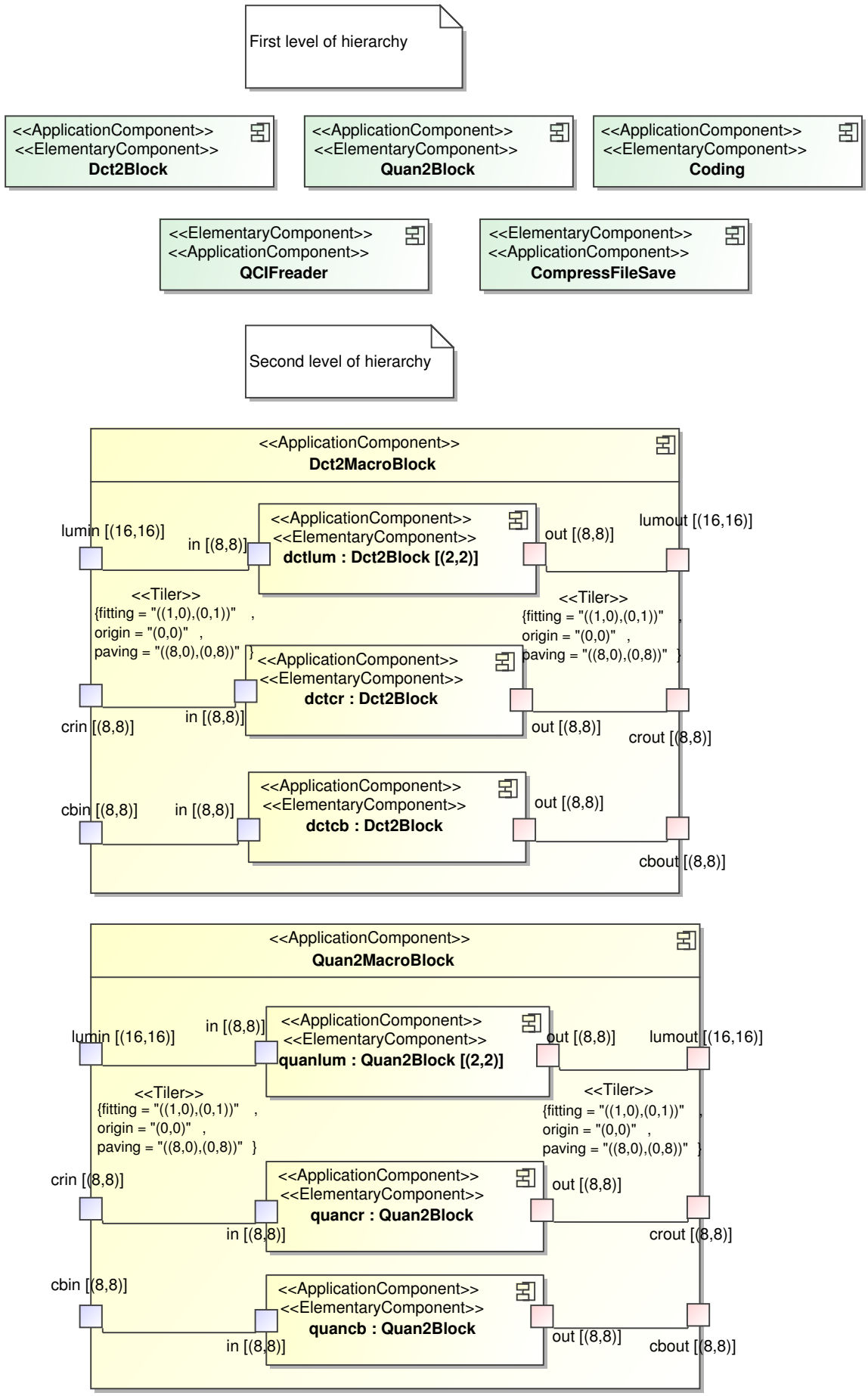
Ce chapitre offre l'occasion pour le lecteur d'avoir une idée plus précise sur l'organisation globale de la conception des systèmes sur puce à l'aide de l'environnement Gaspard. En l'occurrence, nous allons exposer une étude de cas à l'aide de l'application codeur H.263. Notre objectif est de valider de manière expérimentale les différentes contributions et leur intégration dans notre environnement. Dans un premier temps, nous modélisons le système MPSoC entier : application, architecture et association. Après avoir choisi un niveau de simulation (PVT-PA) pour co-simuler notre système, nous procédons à une étape de déploiement afin de spécifier les composants abstraits du haut niveau. En outre, toutes les caractéristiques temporelles des composants ainsi que les modèles de consommation correspondants sont spécifiés à cette étape. Ensuite, des transformations de modèles seront appliquées au modèle déployé afin de générer du code SystemC. Les résultats de la simulation seront comparés avec ceux du niveau CABA en termes d'accélération et de précision d'estimation. Enfin, nous ferons varier des paramètres du modèle MPSoC dans l'objectif d'une évaluation rapide de l'espace de solutions architecturales.

7.2 Codeur H.263 sur MPSoC

7.2.1 Le modèle d'application

Dans cette étude de cas nous avons adopté la même application codeur H.263 qui a été introduite dans le chapitre 4. Nous rappelons ici que notre application est constituée principalement de 3 tâches : la transformée en cosinus discrète, la quantification et le codage. Au cours de cette étude, nous optons pour une solution purement logicielle pour implémenter ces tâches. En réalité, nous ne possédons pas aujourd'hui une sémantique précise afin d'exprimer le placement de calcul d'adresses sur des accélérateurs matériels tel que le contrôleur DMA. Les séquences traitées sont de format QCIF (176×144 pixels). Dans l'algorithme de codage, les données manipulées sont structurées sous forme de macroblocs qui représentent un espace de 16×16 pixels d'une image vidéo. Chaque macrobloc contient trois composantes : luminance (Y), chrominance bleue (Cb) et chrominance rouge (Cr). Un macrobloc contient 6 blocs de 8×8 valeurs : 4 blocs contiennent les valeurs de la luminance, un bloc contient les valeurs de la chrominance bleue et un bloc contient les valeurs de la chrominance rouge. Afin d'avoir une simulation bornée dans le temps, nous avons choisi d'exécuter le codeur H.263 sur une séquence vidéo de 400 images. En utilisant notre profil Gaspard, nous avons modélisé notre application en partant tout d'abord des modèles des tâches élémentaires. Ensuite, nous exprimons le parallélisme de données ou celui de tâches à chaque niveau de hiérarchie jusqu'à atteindre le niveau le plus haut. Ce dernier représente l'application de base *VideoSequence* contenant une répétition de 400 de la tâche *QCIF2H263* correspondant au traitement de 400 images (figure 7.1).

Dans la figure 7.1, le premier niveau de hiérarchie représente les tâches élémentaires de l'application. En plus des 3 tâches de base, nous modélisons aussi les deux tâches *QCIFReader* et *CompressFileSave*. La tâche *QCIFReader* se charge de lire une séquence vidéo au format QCIF image par image. Pour chacune, elle produit trois tableaux correspondant aux trois composantes Y, Cb et Cr. Dans la réalité, un périphérique d'entrée/sortie se charge de l'acquisition des données depuis une caméra. Quant à la tâche *CompressFileSave*, elle se charge de sauvegarder le flux des macroblocs compressés.



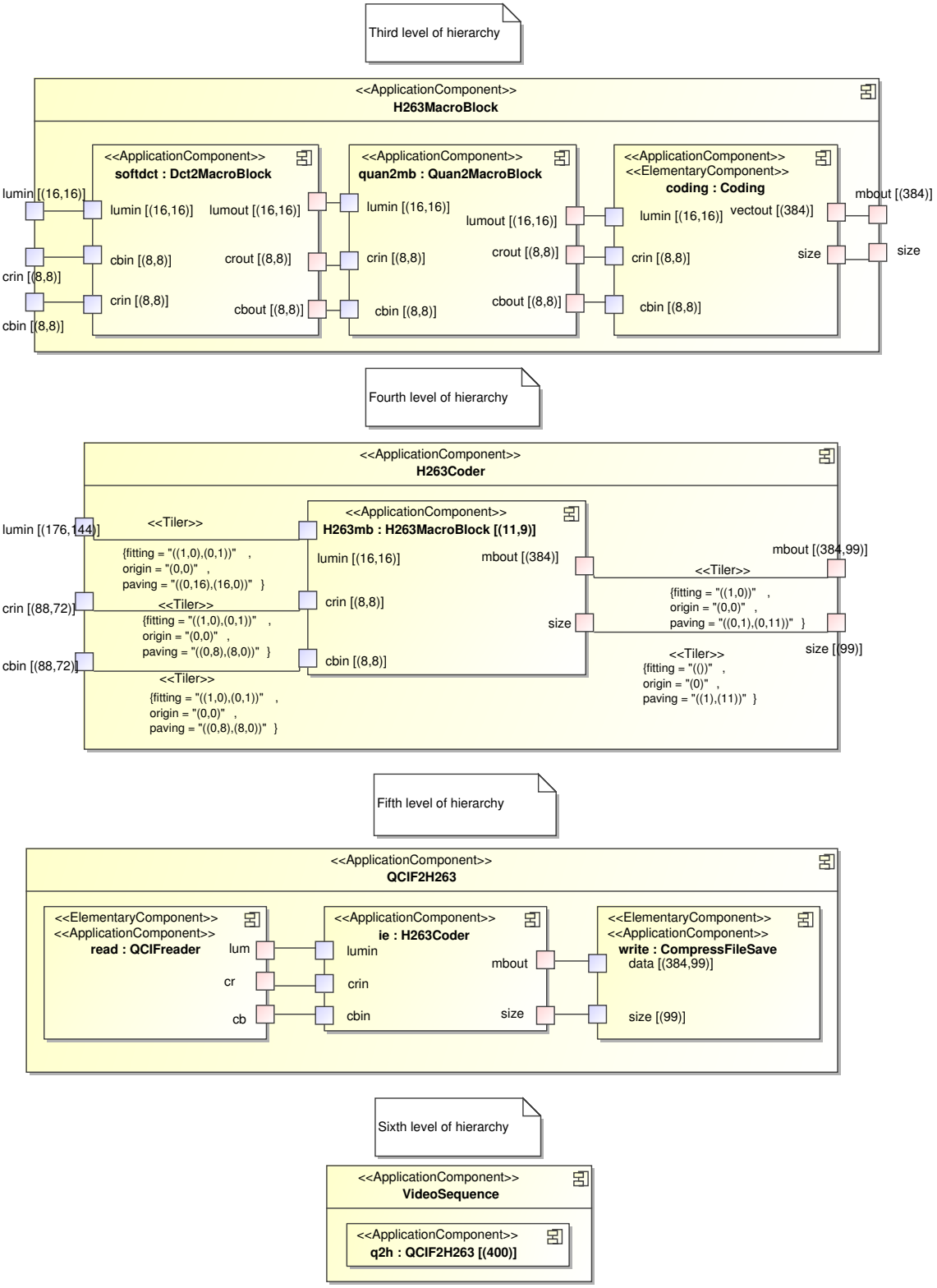


FIG. 7.1 – *Le codeur H.263 modélisé avec le profil Gaspard*

Les deux tâches élémentaires *Dct2Block* et *Quan2Block* présentant respectivement la TCD et la quantification sont appliquées à des blocs de données de 8×8. Or, dans notre application les données sont traitées sous forme de macroblocs. En l'occurrence, les deux tâches *Dct2MacroBlock* et *Quan2MacroBlock* sont modélisées permettant ainsi un deuxième niveau de hiérarchie. Chacune est composée de trois sous-tâches élémentaires représentant le traitement sur les composantes : luminance (répété 4 fois avec une *shape* de (2,2)), la chrominance rouge (une fois) et la chrominance bleue (une fois). Afin de traiter la luminance, des *Tilers* sont utilisés pour découper le bloc 16×16 en quatre motifs de 8×8 en entrée puis de reconstituer le bloc complet en sortie.

Le composant H263MacroBlock correspond à l'algorithme tel que nous l'avons décrit plus haut mais appliqué à un seul macrobloc : un appel à la DCT (*Dct2MacroBlock*), puis à la quantification (Quan2MacroBlock), et enfin au codage (Coding). Ce modèle présente bien un graphe de dépendance de tâches.

A un niveau de hiérarchie plus haut, la tâche de codage *H263Coder* est modélisée. Elle représente réellement tout l'algorithme pour coder une image. Elle contient la sous-tâche répétée *H263MacroBlock*. La répétition consiste à travailler sur chacun des 11×9 macroblocs composant une image de format QCIF. Comme on peut le voir par la taille des ports d'entrée de *H263Coder*, une image QCIF correspond à 176×144 pixels de la luminance et 88×72 pixels des chrominances. Le découpage de l'image d'entrée sur des macroblocs se fait à l'aide des *Tilers*. Ce modèle présente bien du parallélisme de données.

A titre d'exemple le *Tiler* pour la chrominance rouge a un ajustage de ((1,0),(0,1)) indiquant une tuile compacte : un avancement d'un élément dans la première (respectivement la deuxième) dimension du motif lui correspond également un avancement d'un élément dans la première (respectivement la deuxième) dimension du tableau. Le pavage est de ((8,0),(0,8)) : pour lire le motif suivant dans la première dimension de la répétition il faudra se décaler de 8 pixels dans la première dimension du tableau, de même pour la seconde dimension. Cela est illustré dans la figure 7.2. Enfin, pour des motifs de taille 8×8 et un espace de répétition de 11×9, nous allons obtenir l'ensemble des 88×72 éléments du tableau. Le composant *QCIF2H263* représente un graphe de dépendance entre les tâches : *QCIFReader* produisant les tableaux d'entrées, *H263Coder* et *CompressFileSave* sauvegardant les résultats. L'itération de la tâche *QCIF2H263* sur un espace unidimensionnel de 400 permet d'obtenir notre application appelée *VideoSequence*.

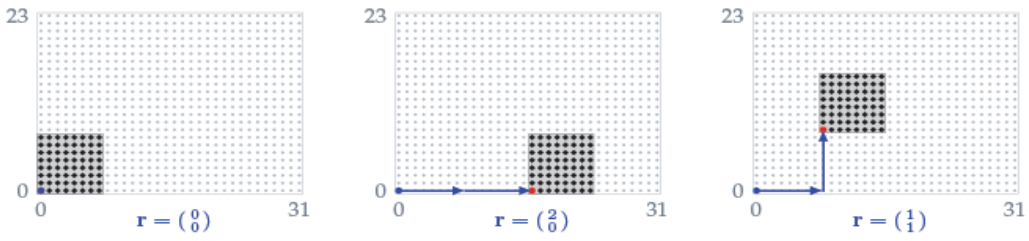


FIG. 7.2 – *Vue partielle des lecture des motifs à partir du tableau de chrominance*

7.2.2 Le modèle d'architecture

En utilisant le profil Gaspard, nous modélisons une architecture MPSoC représentant un support d'exécution de notre codeur H.263. L'architecture matérielle contenue dans le composant *HardwArchit* est constituée de plusieurs unités de calcul (*MultiMips*), un ou plusieurs bancs mémoires de données (*MultiBankMemory*), une mémoire d'instructions et un réseau d'interconnexion. Dans la figure 7.3, nous avons fixé le nombre de processeurs à 8 et le nombre de bancs mémoires à 4. Néanmoins, ces deux valeurs seront variées ensuite pour explorer différentes alternatives de l'espace de solutions architecturales. Dans cette modélisation, nous profitons du mécanisme de factorisation introduit par le méta-modèle Gaspard afin de présenter de façon compacte les composants répétés. A titre d'exemple, le composant *MultiMips* contient un sous-composant *ProcessingUnit* répété 8 fois. Ce dernier est lui même composé de deux sous-composants. Il contient un processeur nommé *mips* et un cache nommé *c* qui fait l'interface entre le processeur et le réseau d'interconnexion. Il en est de même pour le composant *MultiBankMemory* qui contient un sous-composant *DataRAM* répété 2×2 fois. Dans notre architecture ces différents composants sont connectés via le réseau d'interconnexion *Crossbar*. Ce dernier possède deux ports multiples. Le premier est de type *In* (proc[(8)] dans la figure 7.3) permettant de connecter les composants ayant des ports initiateurs tels que les processeurs. La multiplicité de ce port est déterminée par le nombre de composants actifs dans l'architecture. De façon similaire, le deuxième port du réseau d'interconnexion est de type *Out* (slave[(5)] dans la figure 7.3), auquel on peut connecter les composants ayant des ports cibles tels que les circuits mémoires. Pour relier les différents ports, des connecteurs de différentes topologies sont utilisés. Nous utilisons des connecteurs simples pour relier des ports non répétés. Dans le cas inverse (ports multiples), des *Reshapes* sont modélisés afin d'exprimer la distribution du port de départ sur celui d'arrivée.

7.2.3 Le modèle d'association

Une fois les modèles d'architecture matérielle et d'application définis, nous procédons une phase d'association. La figure 7.4 présente l'association que nous avons retenue. Tout d'abord, nous présentons deux instances de type *MainInstance* correspondant au composant principal de l'architecture *SoCHw* et celui de l'application *SoCSw*. Au niveau de hiérarchie le plus haut, la racine de l'application est placée via une *DataAllocation* sur le composant *MultiBankMemory*. Cela exprime que tous les tableaux de l'application seront contenus dans ces bancs mémoires. Il en est de même, la racine de l'application est également placée via une *TaskAllocation* sur le composant *MultiMips*, indiquant que toute l'application sera exécutée sur cette partie de l'architecture.

Pour spécifier le placement précis des différentes tâches de l'application sur les huit processeurs, nous avons représenté la structure interne des composants applicatifs *QCIF2H263* et *H263Coder* ainsi que celle du composant *MultiMips*. Les deux tâches élémentaires correspondant aux composants *CompressFileSave* et *QCIFreader* sont placées respectivement sur le septième et le huitième processeur. Elles sont mises sur des processeurs différents dans le but de répartir la charge de calcul le plus équitablement possible. Pour ce placement nous utilisons des *Distributions*, puisqu'elles sont nécessaires pour une allocation qui met en jeu au moins une instance répétée (*ProcessingUnit[(8)]*). Nous détaillons celle qui permet de placer *QCIFreader* sur le huitième processeur. Cette distribution a un espace de répétition de (1) et une forme de motif nulle (un tableau de zéro dimension, un seul point) car la tâche n'est

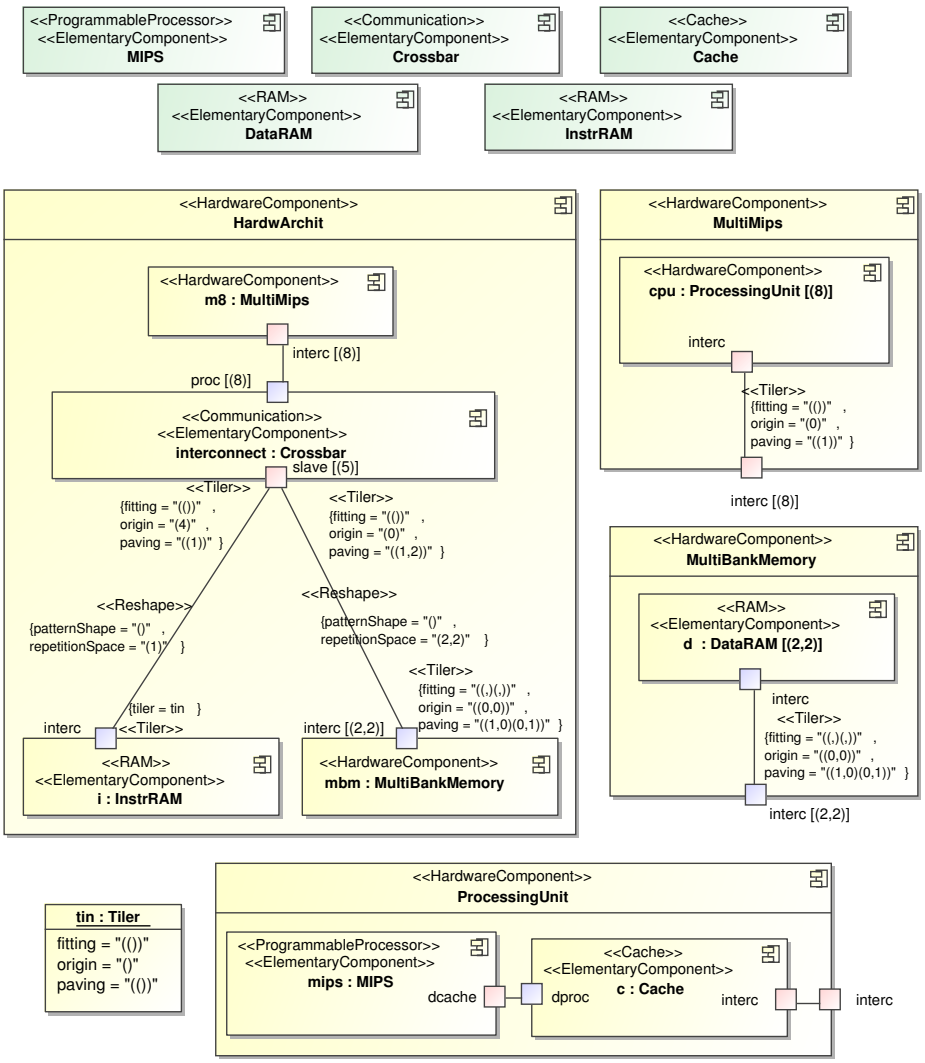


FIG. 7.3 – *Architecture MPSoC modélisée avec le profil Gaspard*

pas répétée. Elle utilise en entrée le *Tiler* `one-point` et en sortie le *Tiler* `one-seventh`. Le fonctionnement de cette distribution consiste à prendre un élément dans le tableau contenant les tâches à l'aide du *Tiler* `one-point`, il n'en existe de toute façon qu'un seul. Ensuite à l'aide du second *Tiler*, cet élément sera placé sur un élément du tableau de processeur (parmi les huit disponibles). Le numéro du processeur est spécifié par l'origine du *Tiler* qui vaut (7) dans note cas.

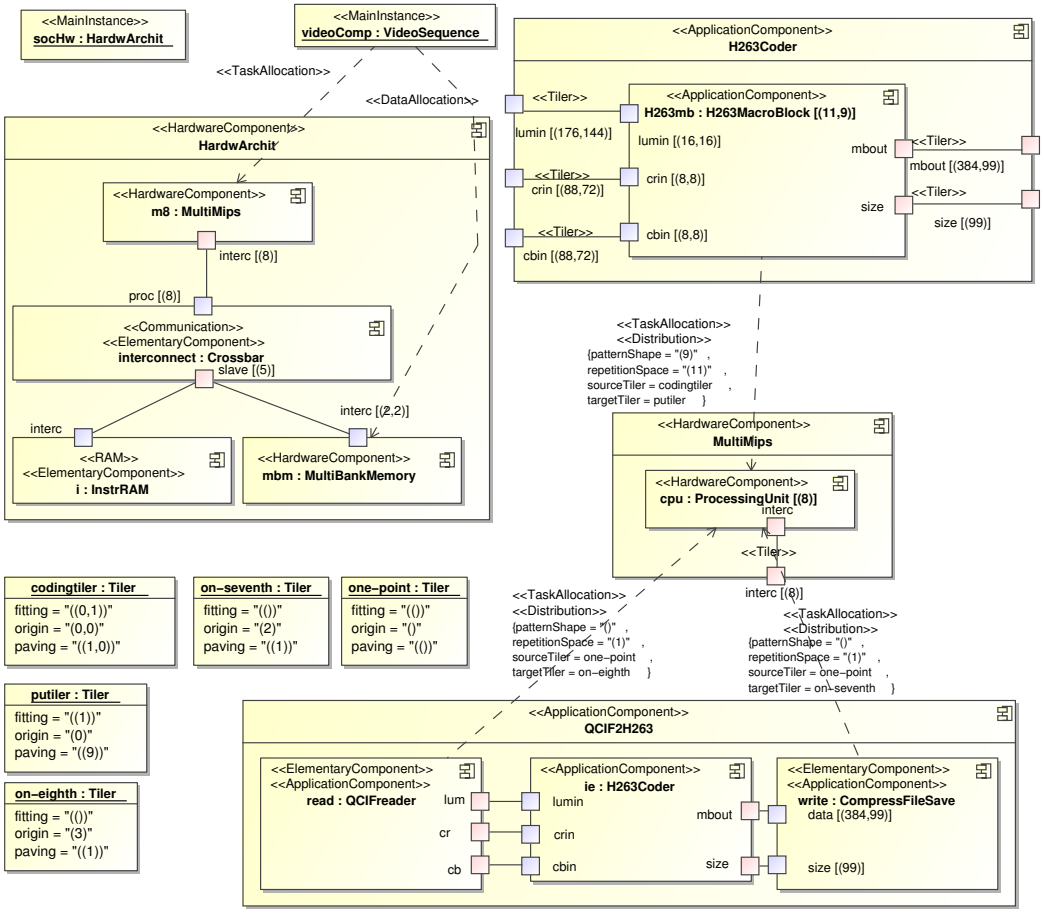


FIG. 7.4 – *Association de l'application H.263 sur une architecture à huit processeurs*

La dernière distribution permet de répartir le codage sur les différents processeurs. Nous avons choisi de le faire à grain fin en répartissant le traitement de chaque image sur les huit processeurs. Nous utilisons le parallélisme de données spécifié dans la tâche *H263Coder*. En effet, chaque image de 176×144 pixels est traitée sous la forme de 11×9 macroblocs. Nous partageons ces 99 sous-tâches sur les huit processeurs permettant ainsi à chacun de coder différents macroblocs. Néanmoins 99 n'étant pas divisible par 8, la répartition n'est pas parfaitement équitable.

Notre distribution considère que le tableau de 11×9 tâches est organisé sous forme linéaire (tableau de 99 éléments). Le tableau de processeurs ne contient que 8 éléments et grâce à l'usage du modulo, tous les éléments placés après le huitième processeur sont de nouveau placés à partir du premier processeur. Cette distribution étant schématisée dans la figure 7.5, nous présentons seulement les deux premières répétitions. Un motif de 9 éléments

est lu puis placé sur les huit processeurs. Le premier processeur reçoit les deux éléments 1 et 9 alors que les autres processeurs reçoivent chacun un seul élément. Lors de la répétition du deuxième motif, il est lu depuis la deuxième ligne et placé sur les processeurs à partir de l'indice 1 (9 modulo 8). Le deuxième processeur reçoit donc le premier élément, ainsi que le 9, les autres processeurs reçoivent chacun un seul élément, etc. Sur cette distribution, nous définissons une *patternShape* de 9 exprimant que le motif lu est de 9 éléments. Par ailleurs, un *repetitionSpace* de 11 est défini signifiant que le processus va être appliqué 11 fois.

Le *Tiler* `codingtiler` exprime d'une part que la lecture du motif dans le tableau de tâches est obtenue en se décalant d'un élément le long de la deuxième dimension, et d'autre part il exprime que le déplacement du motif à chaque répétition se fait en se décalant d'un élément le long de la première dimension. Le *Tiler* `putiler` spécifie le placement de ces motifs dans le tableau de processeurs. Il indique d'une part que le motif est écrit en se décalant d'un élément le long de la seule dimension, et d'autre part que chaque motif est écrit l'un après l'autre, c'est-à-dire en se décalant de 9 éléments. Il est intéressant de noter que cette distribution peut s'adapter à un nombre de processeurs quelconque. Elle permet une meilleure répartition des 99 tâches sur tous les processeurs.

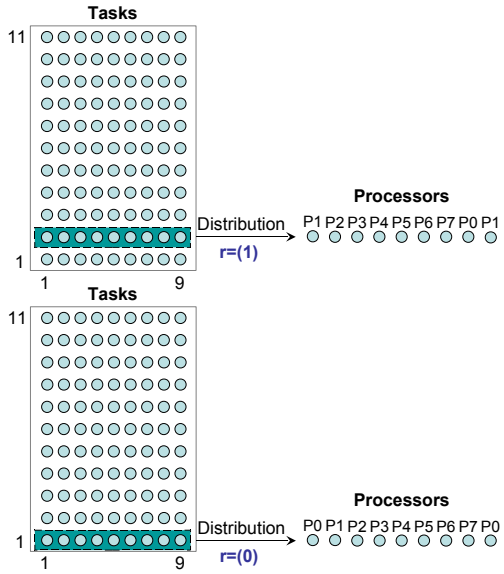


FIG. 7.5 – *Distribution des 11×9 tâches sur les huit processeurs*

7.3 Le déploiement

Le passage de l'état contemplatif de notre modèle MPSoC à un état productif ou exécutable nécessite tout d'abord la définition d'une cible de compilation. Dans cette étude de cas, nous nous intéressons à la génération automatique du code SystemC pour la co-simulation logicielle/matérielle au niveau PVT-PA. Afin d'atteindre notre objectif, une phase de déploiement est nécessaire. Elle permet d'associer les composants élémentaires de haut niveau à des IPs réels intégrés dans la bibliothèque GasparLib (chapitre 6). A chaque tâche élémentaire de notre Codeur H.263, lui correspond une fonction logicielle étant disponible dans la bibliothèque GasparLib. Le lien entre les composants modélisés en haut niveau et ces fonc-

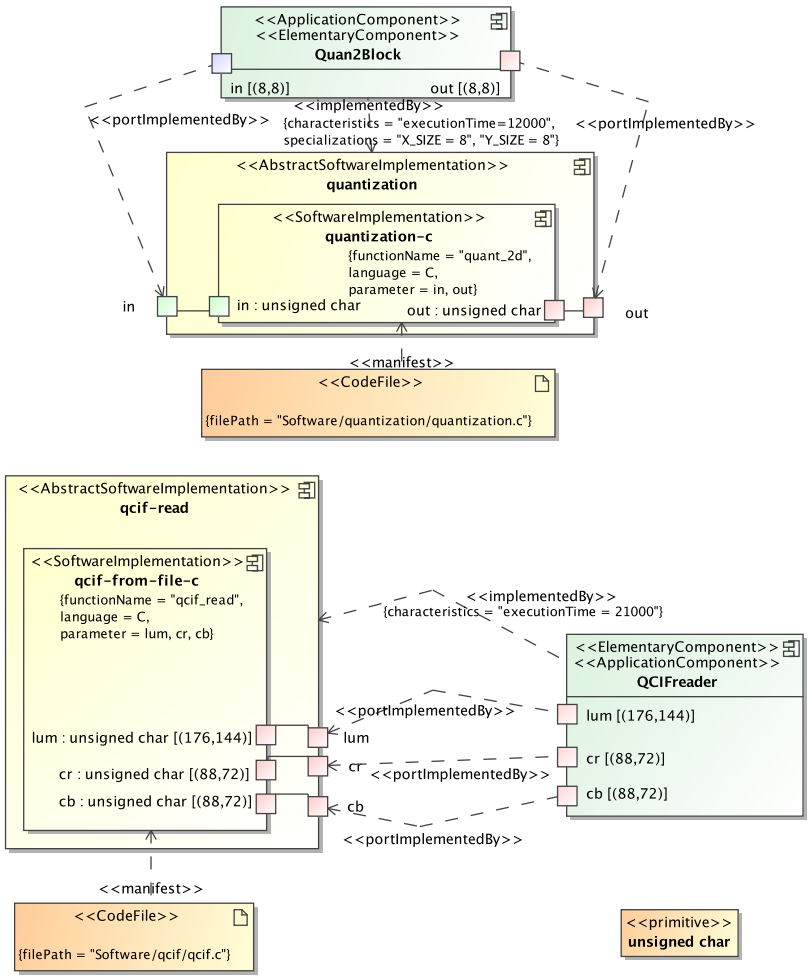


FIG. 7.6 – *Déploiement des tâches élémentaires Quan2Block et QCIFReader*

tions sera réalisé via les différents concepts introduits par le méta-modèle de déploiement. La figure 7.6 présente une partie du déploiement des tâches élémentaires. *Quan2Block* est déployée sur l'implémentation abstraite *quantization*, qui ne contient qu'une seule implémentation d'IP : `quantization-c`. Cette implémentation utilise un fichier (`quantization.c`) et fait appel à la fonction `quant_2d()`. Le type de données traitées par cette fonction (`unsigned char`) est spécifié via le type des ports de l'implémentation. Nous retrouvons sa définition comme type *primitive* en bas à droite dans la figure 7.6. Afin de spécifier la taille des tableaux de données manipulés par la fonction `quant_2d()`, nous utilisons la dépendance *ImplementedBy* qui porte deux spécialisations : X_SIZE et Y_SIZE. Chacun de ces paramètres est fixé à 8. Il faut également noter la spécification de la caractéristique `executionTime` qui permettra à la simulation PVT-PA d'estimer le temps d'exécution de l'application. Le temps d'exécution de la fonction `quant_2d()` a été mesuré avec une simulation au niveau CABA. De façon similaire, la tâche élémentaire *QCIFReader* est déployée sur l'implémentation abstraite *qcif-read*. Comme le standard QCIF correspond à une taille d'image spécifique de 176×144 pixels, l'implémentation de l'IP *qcif-from-file-c* possède des ports de taille fixe.

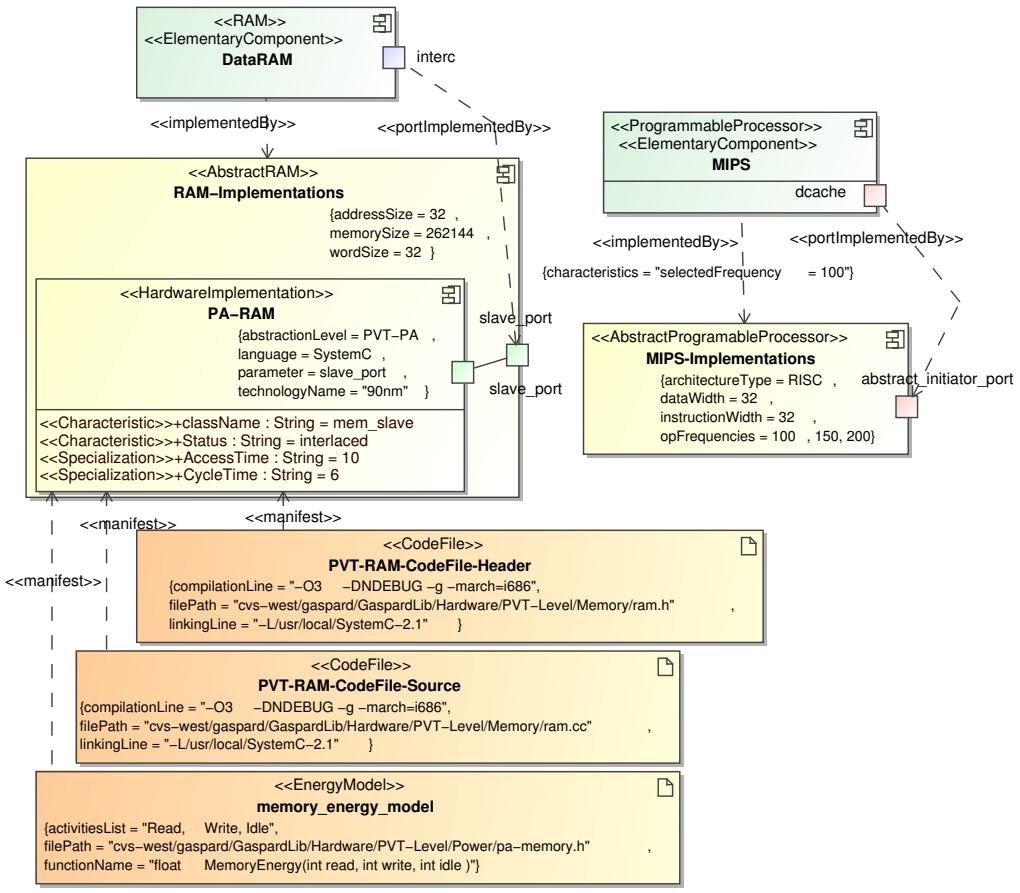


FIG. 7.7 – *Déploiement des composants DataRAM et MIPS*

Les composants matériels de notre architecture sont déployés sur des IP décrits au niveau PVT-PA. Nous avons pu utiliser nos IP définis dans la bibliothèque GaspardLib. La figure 7.7 présente le déploiement des composants *DataRAM* et *MIPS*. Le MIPS a été déployé sur une implémentation abstraite de processeur MIPS. La caractéristique `selectedFrequency=100` a été ajoutée afin de fixer la vitesse d'exécution du processeur. Notons que dans le cadre de la génération vers une simulation PVT-PA, l'implémentation du processeur n'est utile que pour connaître l'interface des ports, l'IP (ISS) n'est pas utilisé à ce niveau d'abstraction. Le composant *DataRAM* a été déployé sur une implémentation de mémoire (*PA-RAM*) décrite au niveau PVT-PA. Sur cette implémentation, nous mettons l'accent sur les spécifications nécessaires pour l'estimation des performances et de la consommation d'énergie. En l'occurrence, les caractéristiques `AccessTime` et `CycleTime` sont définis afin d'estimer le temps d'exécution lors d'une opération de lecture ou d'écriture. La caractéristique `Status` permet de spécifier le type de la mémoire utilisée : partagée, privée ou entrelacée. Le type `interlaced` a été ajouté afin d'être utilisé dans le cas où plusieurs bancs mémoires sont définis dans l'architecture. Cela permet aux processeurs de réaliser des accès mémoires simultanés et par conséquent d'améliorer les performances du système. L'artifact *Memory-Energy-Model* définit le modèle de consommation d'énergie du composant mémoire ayant comme activités pertinentes les

accès en lecture et en écriture ainsi que l'état de repos. Afin d'évaluer les coûts énergétiques de ces activités, le modèle de consommation a recours aux paramètres `memorySize`, `wordSize` et `technologyName` qui sont définis dans l'implémentation abstraite du composant mémoire.

7.4 Génération de code à l'aide de l'environnement Gaspard

Jusqu'à cette phase de conception, nous avons illustré la description du MPSoC qui se fait via la création d'un modèle UML en utilisant le profil Gaspard. Cette modélisation a été réalisée à l'aide de l'outil MagicDraw à partir duquel nous exportons le modèle vers le format UML Ecore. Dans la pratique, ce format sera adopté par la chaîne de transformations de Gaspard jusqu'à arriver à la phase de génération de code. La figure 7.8 illustre notre environnement Gaspard intégré comme un plug-in dans Eclipse. Le panneau de gauche présente les différents modèles utilisés dans la chaîne de transformation : UML, Gaspard, Polyhedron, Loop, et le code source SystemC/PA. Le panneau de droite présente le contenu de chaque modèle tandis que les boutons de la barre d'outils en haut permettent d'appliquer une transformation spécifique à un modèle.

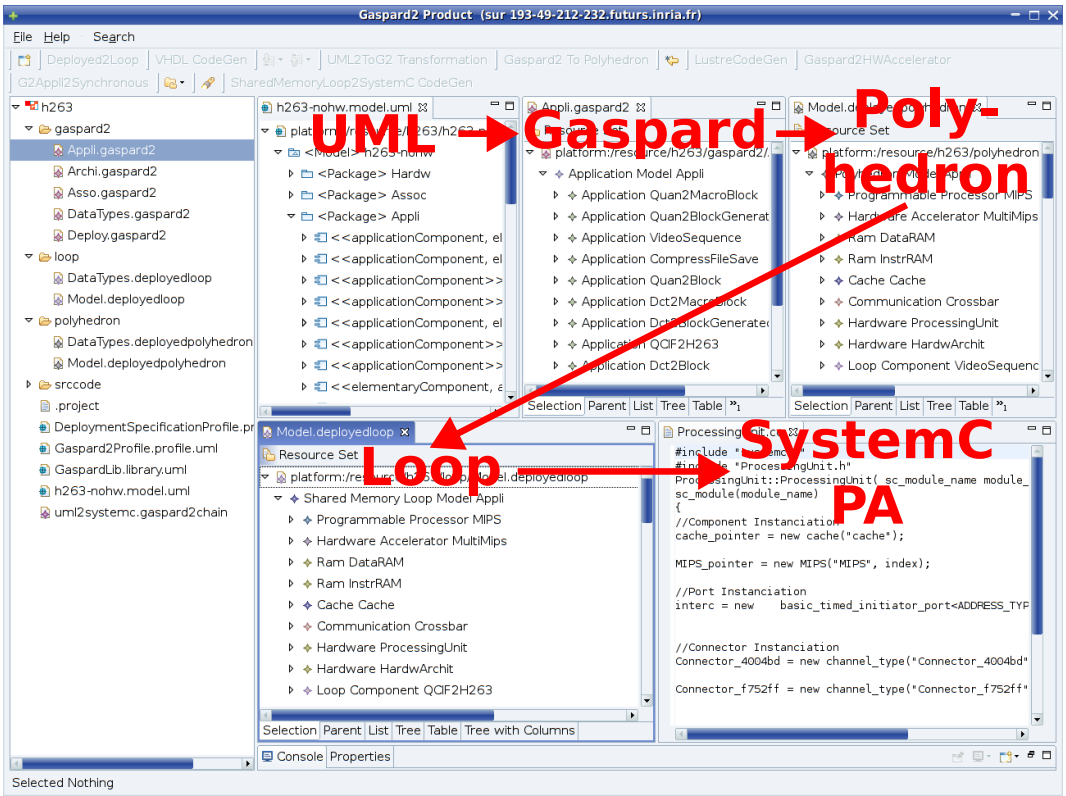


FIG. 7.8 – *La chaîne de transformations de modèles intégrée dans un plug-in Eclipse*

Pour utiliser notre environnement, il faut tout d'abord créer un projet et y insérer le modèle MPSoC. Comme une deuxième étape, l'utilisateur doit choisir l'une des chaînes de compilation disponibles. Il y existe actuellement quatre permettant la génération des codes suivants : langage synchrone, VHDL, OpenMP/Fortran, et SystemC/PA. C'est cette dernière cible qu'il faut choisir dans notre étude de cas. Au fur et à mesure qu'une transformation est

exécutée, son modèle correspondant est généré automatiquement. Sur la capture d'écran, les modèles sont visibles dans le panneau de droite qui permet d'afficher le contenu de chacun sous une forme arborescente textuelle ou les fichiers de code générés.

A la fin de l'exécution de la chaîne vers SystemC/PA, nous obtenons un répertoire contenant l'ensemble des fichiers nécessaires à la compilation du simulateur. Dans cet exemple, nous avons obtenu une quinzaine de fichiers. Une fois ces fichiers générés, le concepteur peut compiler le simulateur en tapant la commande `make` dans le répertoire puis exécuter la simulation à l'aide de la commande `./TLMrun`. Les délais d'exécution de chaque processeur ainsi que le temps d'exécution total sont inclus dans un fichier journal généré à la fin de la simulation. Concernant la consommation d'énergie, nous avons réussi à récupérer les occurrences des activités pertinentes de chaque composant élémentaire. Par la suite, la consommation totale peut être calculée de façon statique à la fin de la simulation. Lors de la génération du code, nous avons rencontré un problème pour calculer la consommation totale de façon dynamique. Il s'agit de faire remonter les occurrences des activités via plusieurs niveaux de hiérarchie jusqu'à arriver à celui le plus haut. Une solution à ce problème nous paraîssant intéressante est la mise en plat et le déroulement de l'architecture avant d'arriver à la phase de génération de code.

7.5 Exploration de l'espace architectural

Afin de démontrer l'usage, l'efficacité et la flexibilité de notre environnement de conception MPSoC développé, nous nous focalisons à explorer quelques paramètres de l'espace de solutions architecturales. En particulier, nous allons faire varier le nombre de processeurs et le nombre de bancs mémoires dans l'architecture. L'objectif est de démontrer qu'avec une simple modification du modèle de haut niveau, nous sommes capables de régénérer le code SystemC et de mesurer de nouveau les performances. Il est intéressant de dire que la modification du modèle MPSoC se fait manuellement. L'automatisation de cette phase fait l'objectif de nos futurs travaux.

7.5.1 Variation du nombre de processeurs

Dans une première phase de notre exploration, nous avons varié le nombre de processeurs de l'architecture (figure 7.3, page 178) de 4 à 8, 12 et 16. La modification dans le modèle de départ est mineure, elle consiste à changer la multiplicité du composant *ProcessingUnit* et des ports des composants *MultiMips* et *Crossbar*. L'association n'a pas besoin d'être modifiée car nous l'avons écrite de manière à ce qu'elle puisse s'adapter avec n'importe quel nombre de processeurs. Une fois le modèle modifié, la phase d'exportation vers le format UML Ecore est refaite et la chaîne de transformations est exécutée de nouveau. Après la génération du code, la simulation est exécutée afin d'obtenir les nouveaux résultats.

Afin de vérifier les résultats de simulation de PVT-PA, nous avons mesuré la précision d'estimation de performance et l'accélération de la simulation. Ces métriques sont rapportées en comparaison avec le niveau CABA qui n'a pas été généré automatiquement. Malheureusement, la chaîne de transformations vers cette cible n'est pas encore achevée dans notre environnement Gaspard. La figure 7.9 illustre les temps d'exécution en cycles obtenus en fonction du nombre de processeurs pour les deux niveaux d'abstraction. Notons que les résultats correspondent à la simulation du codage d'une seule image de la séquence vi-

déo. Les résultats de simulation montrent une erreur d'estimation dans PVT-TA qui varie entre 15% et 30%. Comme nous l'avons déjà expliqué dans le chapitre 4, cette erreur est due principalement à deux causes. La première est liée à notre supposition de négliger les accès mémoires qui correspondent aux variables locales des fonctions et des variables de synchronisation. Par ailleurs, la politique *write-through* adoptée par le cache augmente ce type d'accès. La deuxième cause correspond à l'erreur de détection des contentions dans le réseau d'interconnexion.

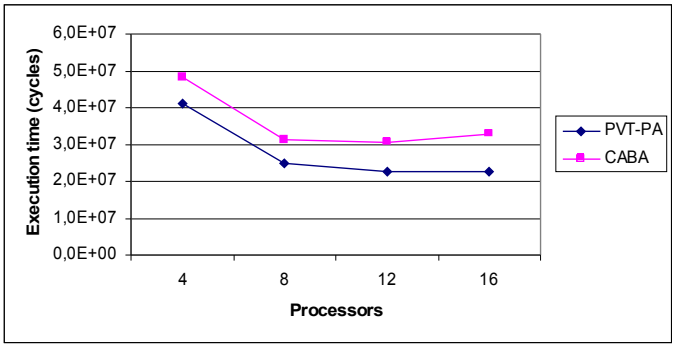


FIG. 7.9 – *Temps d'exécution en fonction du nombre de processeurs pour les niveaux PVT-PA et CABA*

Bien que l'erreur d'estimation absolue entre PVT-PA et CABA paraisse assez importante, la simulation à ces deux niveaux permet de retirer les mêmes conclusions. L'augmentation du nombre de processeurs diminue le temps d'exécution jusqu'à 12 processeurs. Avec 16 processeurs, la tendance est inversée et le temps d'exécution augmente. Lors de l'exploration de l'espace de solutions architecturales, l'erreur absolue ne possède pas trop d'importance. En effet, ce qui est intéressant est que les deux niveaux d'abstraction permettent d'obtenir la même classification des configurations entre elles. Ainsi, malgré l'erreur d'estimation de la simulation PVT-PA, ce dernier nous permet de prendre les bonnes décisions vis-à-vis de l'exploration.

La figure 7.10 présente le facteur d'accélération obtenu entre une simulation PVT-PA et une simulation CABA en fonction du nombre de processeurs. Ce facteur augmente de 11 pour 4 processeurs jusqu'à 25 pour 16 processeurs. Il est plus important à chaque fois où le nombre de processeurs augmente traduisant ainsi la particularité du niveau PVT-PA à accélérer la simulation des composants processeurs. Une analyse précise de la trace produite par le simulateur SystemC nous montre que seulement 0,9% du temps a été dédié à la simulation des processeurs. Par ailleurs, la simulation PVT-PA réduit également le coût du noyau du simulateur SystemC car elle évite l'ordonnancement entre chaque instruction simulée. Cela démontre donc l'avantage majeur de simuler les MPSoC à haut niveau d'abstraction. En faisant varier le nombre de processeurs, nous avons noté que le passage de 12 à 16 processeurs a diminué les performances. Ceci est dû principalement au nombre très important de contentions qui bloquent le réseau. Nous allons tenter de pallier ce phénomène en faisant varier le nombre de bancs mémoires.

7.5.2 Variation du nombre de bancs mémoires

En remarquant que les contentions étaient particulièrement élevées pour un MPSoC de 16 processeurs, nous avons décidé d'augmenter le nombre de bancs mémoires et de le faire

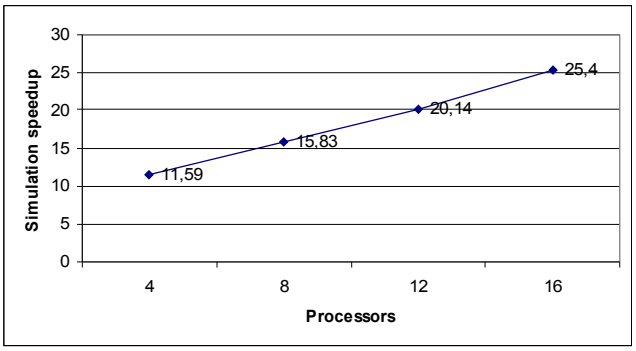


FIG. 7.10 – *Accélération de la simulation en fonction du nombre de processeurs entre les niveaux PVT-PA et CABA*

varier de 1 à 2 puis 4. La modification dans le modèle de départ consiste à modifier la multiplicité du composant *DataRAM* et des ports des composants *MultiBankMemory* et *Crossbar*. Par ailleurs, les allocations de données sur les bancs mémoires, spécifiées via les concepts d'association, doivent être aussi modifiées. Il est intéressant de noter que la spécification de la distribution n'est pas utile lorsqu'il s'agit d'un seul banc mémoire. Dans le cas inverse, nous avons besoin de répartir chacun des cinq tableaux intermédiaires de *QCIF2H263* sur les différents bancs mémoires. Pour ce faire, deux solutions existent :

- La première solution consiste à utiliser des mémoires ayant des adresses physiques différentes. Dans ce cas, nous initialisons la caractéristique `Status` de la mémoire à *shared*(figure 7.7) et l'espace d'adressage réservé sera divisé équitablement sur les différents bancs mémoires. Les données d'un même tableau peuvent être distribuées sur des adresses physiques non-contiguës. Ce cas suppose que les processeurs peuvent prendre en considération des spécifications de la distribution afin de calculer les adresses des données. Nous considérons que cette supposition complexifie la phase de génération de code. Pour cela, elle ne sera pas adoptée actuellement dans notre travail.
- La deuxième solution consiste à utiliser des mémoires avec des adresses entrelacées. Dans le modèle de déploiement, la caractéristique `Status` de la mémoire est fixée à *interlaced* et un seul espace d'adressage est attribué à l'ensemble de bancs mémoires. Dans ce cas, les données seront partagées de façon équitable sur ces bancs. Afin d'identifier le numéro du banc dans lequel une donnée particulière est stockée, nous avons recours à une solution matérielle qui consiste à utiliser les bits de poids faibles de l'adresse. Cette solution a été implémentée au niveau du réseau d'interconnexion.

Parmi les attraits du niveau PVT-PA, il offre la possibilité d'observer les contentions dans le réseau d'interconnexion. Nous avons changé la multiplicité du composant mémoire et des ports dans le modèle. La figure 7.11 illustre les contentions d'accès à la mémoire de données en fonction du nombre de bancs mémoires utilisés. Chaque point d'une courbe correspond au nombre de contentions détectées par le crossbar pour accéder aux différents bancs dans un intervalle de 10ms (1000 cycles). Il est clairement mis en évidence que l'augmentation du nombre de bancs mémoires permet de réduire les contentions. L'usage de 4 bancs mémoires au lieu d'un seul permet de réduire les contentions moyennant de 500 contentions/10ms à moins de 100.

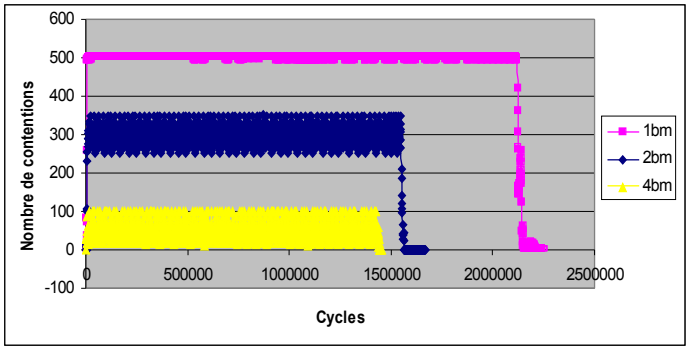
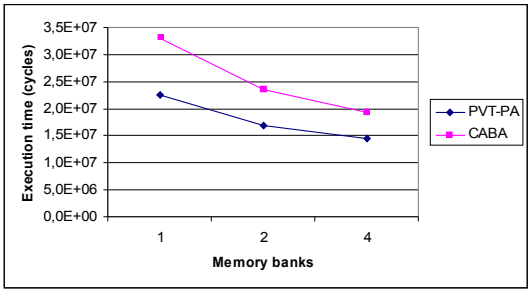
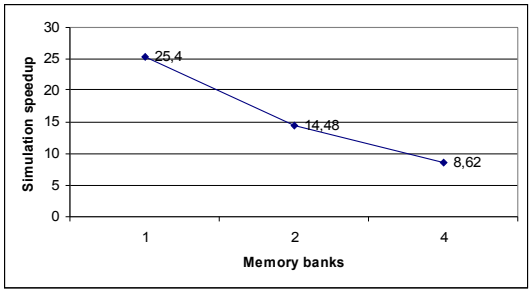


FIG. 7.11 – *Contentions dans le réseau d'interconnexion en fonction du nombre de bancs mémoires*

Les figures 7.12(a)et 7.12(b) présentent respectivement les graphes du temps d'exécution et du facteur d'accélération en fonction du nombre de bancs mémoires. Nous notons de nouveau que la simulation PVT-PA permet de classer correctement les différentes configurations. En effet, plus le nombre de bancs mémoires augmente, plus l'exécution est rapide vu la réduction des contentions. Par ailleurs, le facteur d'accélération de la simulation diminue avec l'ajout de nouveaux bancs mémoires. Cela s'explique par le fait que la simulation des processeurs devient de moins en moins dominante dans la simulation totale avec l'usage d'un nombre important de modules mémoires.



(a) Temps d'exécution en fonction du nombre de bancs mémoires



(b) Accélération de la simulation en fonction du nombre de bancs mémoires

FIG. 7.12 – *Comparaisons entre les simulations PVT-PA et celui CABA en fonction du nombre de bancs mémoires*

7.6 Conclusion

Dans ce chapitre une étude de cas de l'usage de l'environnement Gaspard pour la modélisation d'un MPSoC a été exposée. Cette étude a été menée avec un exemple d'architecture MPSoC à mémoire partagée et l'application de codage H.263. D'une part, cela fut l'occasion de montrer l'usage des propositions faites aux chapitres 4, 5 et 6. Ce fut également l'opportunité de démontrer le fonctionnement de la transformation de génération de code développée au cours de cette thèse et d'esquisser l'usage du flot de conception Gaspard. Nous avons vu que grâce à ce flot il est particulièrement facile d'explorer l'espace de solutions architecturales. En effet, il suffit de modifier le modèle du MPSoC exprimé à l'aide de concepts de

haut niveau et automatiquement la simulation peut être régénérée. Enfin, nous avons comparé la simulation au niveau PVT-PA à des simulations de plus bas niveau (CABA). Des facteurs d'accélération importants ont été mesurés et une comparaison fiable des différentes alternatives a été démontrée.

Chapitre 8

Conclusion et perspectives

8.1 Bilan

Les travaux que nous venons de présenter s'inscrivent dans le cadre du développement de systèmes sur puce multiprocesseurs dédiés à des applications de traitement de signal intensif. Nous avons d'abord exposé les difficultés de la conception des MPSoC liées principalement à la co-simulation logicielle/matérielle et à l'estimation de performance et de la consommation d'énergie. Il est important de souligner que tout au long de notre démarche, nous avons prêté une grande attention aux méthodologies de conception basées sur les modèles afin de résoudre le problème de l'augmentation constante de la complexité de conception. De nos travaux de thèse il résulte la proposition d'une chaîne de compilation automatisée et entièrement dirigée par les modèles. Cette chaîne permet la génération du code SystemC à partir d'une modélisation de haut niveau. Sa mise en œuvre a nécessité l'utilisation de techniques de différents domaines tels que la co-simulation des MPSoC à des niveaux d'abstraction élevés, les techniques d'estimation de performance et de la consommation d'énergie et enfin l'Ingénierie Dirigée par les Modèles (IDM).

Nos travaux ont commencé par l'étude de la problématique du temps de simulation des MPSoC au niveau CABA. En se basant sur la théorie des "machines à états finis (FSM) communicantes synchrones", nous avons détaillé la description d'un certain nombre de modules en utilisant la bibliothèque de composants SoCLib. A travers cette étude, nous avons montré que l'effort de développement nécessaire au niveau CABA est considérable pour obtenir des estimations précises. Par ailleurs, les temps de simulation mesurés ne sont pas suffisamment réduits pour permettre une exploration architecturale rapide des MPSoC. Toutefois, une simulation préliminaire en CABA de certains composants s'avère nécessaire si des informations pertinentes sur les performances et la consommation d'énergie doivent être intégrées dans des niveaux d'abstraction plus élevés.

Pour résoudre cette contradiction, les premières contributions présentées dans cette thèse ont consisté à développer un environnement décrit au niveau PVT permettant l'accélération de la simulation des systèmes MPSoC. Cet environnement est composé de trois sous-niveaux PVT-PA, PVT-TA et PVT-EA offrant différents compromis entre l'accélération et la précision. Différents composants matériels ont été conçus pour implémenter les trois sous-niveaux. Afin d'obtenir une prédiction précise du temps d'exécution dans notre environnement, nous avons enrichi les trois sous-niveaux par des modèles de temps permettant une erreur d'estimation relative à la précision de description du système.

Par la suite, nous nous sommes intéressés à la consommation d'énergie dans les MPSoC. Pour cela, nous avons enrichi les simulateurs MPSoC décrits aux niveaux CABA et PVT par des modèles de consommation d'énergie. Nous avons proposé une méthodologie hybride d'estimation de la consommation basée sur des simulations de bas niveau et des modèles analytiques offrant un niveau acceptable de précision et de flexibilité. L'intégration du critère de la consommation dans notre travail nous a permis d'obtenir un environnement flexible pouvant être utilisé dans une exploration rapide et précise des systèmes MPSoC.

Après ces deux premières étapes, nous avons présenté dans la troisième étape de notre thèse, la mise en place d'une chaîne de compilation dans l'environnement Gaspard. Cette chaîne a pour objectif de générer un simulateur SystemC à partir d'un modèle de MPSoC. Cette chaîne respecte les recommandations de l'IDM, et est constituée de plusieurs transformations qui prennent en entrée et en sortie des modèles conformes à des méta-modèles prédéfinis. Notre contribution dans cette partie consiste en premier lieu à étendre le méta-modèle Gaspard via la proposition du méta-modèle déploiement afin de produire des mo-

dèles exécutables. Notre proposition offre trois avantages. Elle permet une augmentation de la productivité des concepteurs en autorisant l'usage de bibliothèques de composants, elle assure la représentation des fonctionnalités indépendamment de la cible de compilation et enfin permet l'expression des spécifications liées à l'estimation de performance et la consommation d'énergie. En second lieu, nous avons développé une transformation de type modèle-vers-texte qui permet de générer du code SystemC représentant le système à MPSoC à simuler.

Enfin, dans la dernière partie de notre thèse, nous avons illustré l'usage du flot complet de Gaspard et de la simulation au motif près avec un exemple de codeur H.263 placé sur un MPSoC à base de processeurs MIPS. Cela a permis de valider les transformations de modèles ainsi que la simulation à haut niveau d'abstraction. Cette étude de cas a été aussi l'occasion de présenter une exploration d'architecture. Dans ce chapitre, l'exploration a été appliquée aux choix du nombre de processeurs et du nombre de bancs mémoires optimaux pour notre application H.263.

8.2 Perspectives

Le travail effectué dans cette thèse peut être poursuivi suivant de nombreuses directions, nous en présentons ici quelques-unes :

Simulation des MPSoC plus complexes

Dans notre travail, nous nous sommes limités à des systèmes MPSoC à mémoires partagées intégrant un nombre modéré de processeurs ($\leq$16). Par ailleurs, les architectures de processeurs utilisés sont simples tels que le processeur MIPS R3000. Comme nous nous sommes focalisés sur un seul type de réseau d'interconnexion, le crossbar. Il serait intéressant d'étudier par la suite la problématique de co-simulation pour des architectures MPSoC plus complexes. Ces dernières sont composées de différents types de processeurs tels que les processeurs superscalaires ou VLIW (*Very Large Instruction Word*), des topologies de réseaux supportant un grand nombre de composants tels que le NoC, des mémoires distribuées, etc. En outre, des efforts sont nécessaires pour mettre en adéquation les modèles d'estimation de performance et de la consommation d'énergie développés dans cette thèse à la complexité des nouvelles architectures.

Automatisation de l'exploration de l'espace de conception

Dans le chapitre 6, nous avons exposé la façon avec laquelle notre flot de conception peut être intégré à l'environnement Gaspard dans l'objectif de faciliter l'exploration de l'espace de solutions architecturales. En effet, des modifications simples dans le modèle de départ suivi d'une phase de génération automatique du code permettent de tester différentes alternatives. A titre d'exemple, pour changer le nombre de processeurs ou le nombre de bancs mémoires, il suffit de modifier le paramètre correspondant dans le modèle de départ. De plus, en utilisant notre flot de conception il est possible de cibler différents niveaux d'abstraction. Actuellement toutes ces spécifications sont ajoutées manuellement par le concepteur. Il est ainsi possible de prévoir un outil d'exploration automatique autour de notre flot

de conception. Il permettra de parcourir l'espace de solutions en allant des niveaux d'abstraction les plus hauts vers les plus bas. A chaque niveau, on effectue des simulations et on choisit les meilleures alternatives selon les observations de performance et de la consommation d'énergie. Cette proposition nécessite d'une part la spécification des paramètres du modèle de SoC qui peuvent être modifiés pour parcourir l'espace de solutions. D'autre part, les critères multiples auxquels devra se conformer le système final doivent être aussi spécifiés (temps d'exécution de l'application, consommation d'énergie, taille physique, coût de fabrication, etc.).

Extension du modèle d'exécution

Les transformations de modèles que nous avons proposées ne permettent pas de prendre en compte toutes les architectures et les associations possibles. En particulier, deux types de répartitions intéressantes ne sont pas gérés par notre environnement actuel : le placement des données sur des mémoires distribuées et l'allocation des tâches entre des processeurs et des accélérateurs matériels. La chaîne de compilation développée pour l'instant ne prend en compte que des données placées sur des mémoires partagées centralisées et accessibles par tous les processeurs. Dans le cas des mémoires privées, il faut tenir compte du fait que certaines mémoires ne peuvent être accédées qu'à travers le processeur auquel elles sont reliées. Ce type d'architectures utilise un autre mode de programmation qui est le passage de messages (ou message passing). Plusieurs points doivent être pris en considération notamment la nécessité de doter le système d'un mécanisme pour permettre à un processeur de localiser l'information désirée et d'un mécanisme de routage et de gestion des paquets transmis dans chaque nœud du réseau.

Dans cette thèse, nous n'avons considéré que l'exécution logicielle des tâches sur plusieurs processeurs. Dans les travaux de Sébastien Le Beux, ancien doctorant de l'équipe, l'exécution des tâches est assurée par des accélérateurs matériels [74]. Cependant, il n'est pas encore possible d'allouer une partie de l'application de façon logicielle et une autre partie de façon matérielle. La recherche dans cette direction devra proposer un modèle d'exécution capable de synchroniser les tâches entre elles, et définissant une gestion du flot de données qui soit efficace aussi bien lors du traitement sur un processeur que sur un accélérateur matériel.

Bibliographie personnelle

[1] Rabie Ben Atitallah, Smail Niar, and Jean-Luc Dekeyser. MPSoC Power Estimation Framework at Transaction Level Modeling. In *The 19th International Conference on Microelectronics (ICM 2007)*, Cairo, Egypt, December 2007.

[2] Rabie Ben Atitallah, Smail Niar, Alain Greiner, Samy Meftali, and Jean-Luc Dekeyser. Estimating energy consumption for an MPSoC architectural exploration. In *Architecture of Computing Systems (ARCS'06)*, Frankfurt, Germany, March 2006.

[3] Rabie Ben Atitallah, Smail Niar, Samy Meftali, and Jean-Luc Dekeyser. An MPSoC performance estimation framework using transaction level modeling. In *The 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Daegu, Korea, August 2007.

[4] Rabie Ben Atitallah, Eric Piel, Smail Niar, Philippe Marquet, and Jean-Luc Dekeyser. Multilevel MPSoC simulation using an MDE approach. In *IEEE International SoC Conference (SoCC 2007)*, Hsinchu, Taiwan, September 2007.

[5] Rabie Ben Atitallah, Eric Piel, Julien Taillard, Smail Niar, and Jean-Luc Dekeyser. From High Level MPSoC description to SystemC Code Generation. In *International ModEasy'07 Workshop in conjunction with Forum on specification and Design Languages (FDL'07)*, Barcelona, Spain, September 2007.

[6] Rabie Ben Atitallah, Lossan Bonde, Smail Niar, Samy Meftali, and Jean-Luc Dekeyser. Multilevel MPSoC performance evaluation using MDE approach. In *The International Symposium on System-on-Chip 2006 (SOC 2006)*, Tampere, Finland, November 2006.

[7] Rabie Ben Atitallah, Pierre Boulet, Arnaud Cuccuru, Jean-Luc Dekeyser, Antoine Honoré, Ouassila Labbani, Sébastien Le Beux, Philippe Marquet, Éric Piel, Julien Taillard, and Huafeng Yu. Gaspard2 uml profile documentation. Technical Report 0342, September 2007.

[8] Rabie Ben Atitallah, Smail Niar, Samy Meftali, and Jean-Luc Dekeyser. A transaction level modeling based framework for MPSoC performance estimation. *Journal of Systems Architecture*, en deuxième révision, 2008.

[9] Hajer Chtioui, Rabie Ben Atitallah, Smail Niar, Mohamed Abid, and Jean-Luc Dekeyser. Gestion de la cohérence des caches dans les architectures MPSoC utilisant des NoC complexes. In *Rencontres francophones du Parallélisme (RenPar'18) / Symposium en Architecture de machines (SympA '2008) / Conférence Française sur les Systèmes d'Exploitation (CFSE '6)*, Freiburg, Switzerland, February 2008.

Bibliographie

[10] ALLIANCE home page. http://www-asim.lip6.fr/recherche/alliance/.

[11] Mentor home page. http://www.mentor.com.

[12] . SystemVerilog. http://www.systemverilog.org/, 2007.

[13] Altera Startix II Architecture. http://www.altera.com/products/devices/stratix2/st2-index.jsp.

[14] AMBA protocol. http://www.arm.com/products/solutions/AMBAHomePage.html.

[15] ARM Technologies Consortium. http://www.arm.com/products/CPUs/.

[16] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis : an integrated electronic system design environment. *IEEE Computer*, 36(4), Apr. 2003.

[17] C. Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, Sept. 2004.

[18] A. Bellaouar and M. I. Elmasry. *Low-Power Digital VLSI Design Circuits and Systems*. Kluwer Academic Publishers, 1995.

[19] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri. MPARM : Exploring the Multi-Processor SoC Design Space with SystemC. *J. VLSI Signal Process. Syst.*, 41(2) :169–182, 2005.

[20] L. Benini and G. D. Micheli. *Multi-processors systems-on-chips*, chapter Networks on chips : New paradigm for component-based MPSOC design, pages 49–80. Morgan Kaufmann, 2005.

[21] M. V. Biesbrouck, L. Eeckhout, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *ISPASS'04 : Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, Texas, USA, 2004.

[22] M. V. Biesbrouck, L. Eeckhout, and B. Calder. Considering all starting points for simultaneous multithreading simulation. In *ISPASS'06 : Proceedings of the 2006 IEEE International Symposium on Performance Analysis of Systems and Software*, Texas, USA, 2006.

[23] M. V. Biesbroucky, T. Sherwood, and B. Calder. A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation. *IEEE Int'l Symp. On Performance Analysis of Systems and Software*, 2004.

[24] T. Bjerregaard and S. Mahadevan. A survey of research and practices of Network-on-chip. *ACM Comput. Surv.*, 38(1), 2006.

[25] M. Blagojevic. *SOI mixed mode design techniques and case studies*. PhD thesis, Ecole Polytechnique Fédérale de Laussane, 2005.

[26] L. Bonde. *Transformations de Modèles et Interopérabilité dans la Conception de Systèmes Hétérogènes sur Puce à Base d'IP*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Université de Lille 1, 2006.

[27] P. Boulet. Array-OL revisited, multidimensional intensive signal processing specification. Technical Report RR-6113, Feb. 2007.

[28] D. Brooks, V. Tiwari, and M. Martonosi. Wattch : a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 83–94, 2000.

[29] A. Bunker and G. Gopalakrishnan. Formal specification of the Virtual Component Interface Standard in the Unified Modeling Language. `http://www.cs.utah.edu/research/techreports/2001/pdf/UUCS-01-007.pdf`.

[30] CACTI Home Page. `http://research.compaq.com/wrl/people/jouppi/CACTI`.

[31] L. Cai and D. Gajski. Transaction level modeling : an overview. In *CODES+ISSS'03*, New York, USA.

[32] Y. Cao, T. Sato, M. Orshansky, D. Sylvester, and C. Hu. New paradigm of predictive MOSFET and interconnect modeling for early circuit simulation. In *IEEE Custom Integrated Circuits Conference*, 2000.

[33] W. Cesário, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava. Component-based design approach for multicore socs. In *DAC'02 : Proceedings of the 39th conference on Design automation*, New Orleans, Louisiana, USA, 2002.

[34] A. P. Chandrakasan and R. W. Brodersen. *Low Power Digital CMOS Design*, chapter 3. Kluwer Academic Publishers, 1995.

[35] V. Consortium. Vector signal image processing library, 2002. `http://www.vsipl.org/`.

[36] Coreconnect protocol. `www.ibm.com/chips/products/coreconnect`.

[37] G. Cote, B. Erol, M. Gallant, and F. Kossentini. H.263+ : video coding at low bit rates. *IEEE Transactions on Circuits and Systems for Video Technology*, 8(7) :849–866, November 1998.

[38] COWARE. N2c. `http://www.coware.com/cowareN2C.html`.

[39] CoWare Inc, ConvergenSC. `http ://www.coware.com/products`.

[40] U. de Berkeley. Ptolemy project. `http://ptolemy.eecs.berkeley.edu`.

[41] A. Demeure, A. Lafarge, E. Boutillon, D. Rozzonelli, J.-C. Dufourd, and J.-L. Marro. Array-OL : Proposition d'un formalisme tableau pour le traitement de signal multidimensionnel. In *Gretsi*, Juan-Les-Pins, France, Sept. 1995.

[42] A. Donlin. Transaction level : flows and use models. In *CODES+ISSS'04*, Stockholm, Sweden.

[43] Eclipse Consortium. EMF. `http://www.eclipse.org/emf`, 2007.

[44] Eclipse Consortium. JET, Java Emitter Templates. `http://www.eclipse.org/modeling/m2t/?project=jet`, 2007.

[45] M. Eva. *SSADM Version 4 : A User's Guide*. McGraw-Hill Publishing Co, Apr. 1994.

[46] J.-M. Favre. Concepts fondamentaux de l'IDM. De l'ancienne egypte à l'ingénierie des langages. In *2èmes Journées sur l'Ingénierie Dirigée par les Modèles (IDM'06)*, Lille, France, 2006.

[47] A. Fraboulet, T. Risset, and A. Scherrer. Cycle accurate simulation model generation for SoC prototyping. In *SAMOS IV System Modeling, and Simulation*, 2004.

[48] F. Fummi, S. Martini, G. Perbellini, and M. Poncino. Native ISS-SystemC integration for the co-simulation of multi-processor SoC. In *Date'04 : Proceedings of the conference on Design, automation and test in Europe*, Paris, France.

[49] D. D. Gajski and R. H. Kuhn. Guest editor introduction : New VLSI-tools. *IEEE Computer*, 16(12) :11–14, 1983.

[50] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC :Specification Language and Methodology*. Kluwer, 2000.

[51] F. Ghenassia. *Transaction-Level Modeling with SystemC : TLM Concepts and Applications for Embedded Systems*. Springer, 2006.

[52] E. A. Gonzalez. BiCMOS technology processes, trends, and applications. Technical report, November 2004.

[53] GRACE++. *System Level Design Environment for Network-on-Chip (NoC) and Multi-Processor platform (MP-SoC) exploration*.

[54] M. Graphics. Lsim power analyst : Transistor-level simulation, 2004. `http://www.mentor.com/lsmpoweranalyst/datasheet.html`.

[55] M. Graphics. Seamless cve. `http://www.metorg.com/semless`.

[56] T. Grotker, S. Liao, and G. Martin. *System Design with SystemC*. Kluwer, 2003.

[57] A. Hekmatpour and K. Goodnow. DesignCon east 2005 : Standards-compliant IP design advantages, problems, and future directions. `http://www.iec.org/events/2005/designcon_east/pdf/1-tp1_hekmatpour.pdf`.

[58] J. Henkel. A low power hardware/software partitioning approach for core-based embedded systems. In *The 36th Design automation conference*, 1999.

[59] D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the Institute of Radio Engineers*, 40(9) :1098–1101, September 1952.

[60] N. Inglart and S. Niar. Rapid Performance and Power Consumption Estimation Methods for Embedded System Design. In *17th IEEE International Workshop on Rapid System Prototyping*, Crete, Greece.

[61] Intel. 3rd generation itanium : power budget. In *The 40th Design automation conference*, 2003.

[62] Intel StrongARM SA-1100 Microprocessor for Portable Applications. `http://netwinder.osuosl.org/pub/netwinder/docs/intel/datashts/27808706.pdf`.

[63] N. Julien, J. Laurent, E. Senn, and E. Martin. Power Estimation of a C Algorithm Based on the Functional-Level Power Analysis of a Digital Signal Processor. In *ISHPC '02 : Proceedings of the 4th International Symposium on High Performance Computing*, London, UK, 2002.

[64] N. Julien, J. Laurent, E. Senn, and E. Martin. Power consumption modeling and characterization of the TI C6201. *IEEE Micro*, 23(5), 2003.

[65] H. Kadima. *Conception orienté objet guidée par les modèles*. Dunod, 2005.

[66] D. Kim, S. Ha, and R. Gupta. CATS : cycle accurate transaction-driven simulation with multiple processor simulators. In *DATE'07 : Proceedings of the conference on Design, automation and test in Europe*, Nice, France, 2007.

[67] D. Kim, Y. Yi, and S. Ha. Trace-driven HW/SW cosimulation using virtual synchronization technique. In *The 38th Conference on Design Automation*, Las Vegas, USA.

[68] K. Kim and J.-S. Koh. An area efficient DCT architecture for MPEG-2 video encoder. *IEEE Transactions on Consumer Electronics*, 45(1) :62–67, February 1999.

[69] M. Lajolo, A. Raghunathan, and S. Dey. Efficient power co-estimation techniques for system-on-chip design. In *DATE'00 : Proceedings of the conference on Design, automation and test in Europe*, NY, USA, 2000.

[70] J. Laurent. *Estimation de la consommation dans la conception système des applications embarquées temps réels*. PhD thesis, Laboratoire d'Electronique des Systèmes TEmps Réel, Université de Bretagne Sud, 2002.

[71] J. Laurent, N. Julien, E. Senn, and E. Martin. Functional level power analysis : An efficient approach for modeling the power consumption of complex processors. In *DATE'04 : Proceedings of the conference on Design, automation and test in Europe*, DC, USA, 2004.

[72] P. LAURENT. GENeric Image Array Library. `http://www.ient.rwth-aachen.de/team/laurent/genial/genial.html`, 2007.

[73] L. Lavagno, A. Sangiovanni-Vincentelli, and E. Sentovich. *System-level synthesis*, chapter Models of computation for embedded system design, pages 45 – 102. Kluwer Academic Publishers, 1999.

[74] S. Le Beux. *Un flot de conception pour applications de traitement du signal systématique implémentées sur FPGA à base d'Ingénierie Dirigée par les Modèles*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Universitéde Lille 1, 2007.

[75] H. Lee, B. Lee, Y. Lee, and B. Kang. Optimized VLSI Design for Enhanced Image Downscaler. In *Second IEEE Asia Pacific Conference on ASIC*, 2000.

[76] J. Li and S.-L. Lu. Low power design of two-dimensional DCT. In *Ninth Annual IEEE International ASIC Conference and Exhibit*, 1996.

[77] Y. Li and J. Henkel. A framework for estimating and minimizing energy dissipation of embedded hw/sw systems. In *The 35th Design automation conference*, 1998.

[78] H. Lim, V. Piuri, and E. Swartzlander. A Serial-Parallel Architecture for Two-Dimensional Discrete Cosine and Inverse Discrete Cosine Transforms. *IEEE Transactions on Computers*, 49(12) :1297–1309, December 2000.

[79] R. P. Llopis and K. Goossens. The petrol approach to high-level power estimation. In *ISLPED '98 : Proceedings of the 1998 international symposium on Low power electronics and design*, CA, USA, 1998.

[80] C. Loeffler, A. Lieenberg, , and G. Moschytz. Practical fast 1-D DCT algorithms with 11 multiplications. In *International Conference on Acoustics, Speech, and Signal Processing*, 1989.

[81] M. Loghi and M. Poncino. Exploring Energy/Performance Tradeoffs in Shared Memory MPSoCs : Snoop-Based Cache Coherence vs. Software Solutions. In *Design, Automation and Test in Europe Conference and Exhibition*, 2005.

[82] Logiciel Quartus II d'Altera. `http://www.altera.com/products/software/products/quartus2/qts-index.html`.

[83] D. Maliniak. Design languages vie for system-level dominance. Technical report, Oct. 2001.

[84] M. Mamidipaka, K. Khouri, N. Dutt, and M. Abadir. IDAP : a tool for high-level power estimation of custom array structures. In *International Conference on Computer Aided Design*, 2003.

[85] M. Mamidipaka, K. Khouri, N. Dutt, and M. Abadir. Analytical models for leakage power estimation of memory array structures. In *international conference on Hardware/-software codesign and system synthesis*, 2004.

[86] S. Marc, K. Reiner, L.-P. Josep, U. Theo, and V. Mateo. Transistor Count and Chip-Space Estimation of SimpleScalar-based Microprocessor Models. In *Workshop on Complexity-Effective Design*, 2001.

[87] S. Meftali. *Exploration d'architectures et allocation/affectation mémoire dans les systèmes multiprocesseurs monopuce*. PhD thesis, Laboratoire Techniques de l'Informatique et de la Microélectronique pour l'Architecture des ordinateurs, Université de Joseph Fourier, 2002.

[88] T. Meyerowitz. Transaction Level Modeling Definitions and Approximations. `http://www.eecs.berkeley.edu/~alanmi/courses/2005_290A/reports/290a_modeling.pdf`.

[89] MIPS Technologies Consortium. `http://www.mips.com`.

[90] M. Miyama, S. Kamohara, M. Hiraki, K. Onozawa, and H. Kunitomo. Pre-silicon parameter generation methodology using BSIM3 for circuit performance-oriented device optimization. *IEEE Transaction on Semiconductor Manufacturing*, 14(2) :134–142, May 2001.

[91] S. Mohanty, V. K. Prasanna, S. Neema, and J. Davis. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. In *Conference on Languages, compilers and tools for embedded systems*, Berlin, Germany, 2002.

[92] MORPHEUS project. Multi-purpose Dynamically Reconfigurable Platform for Intensive Heterogeneous Processing. `http://www.morpheus-ist.org/`.

[93] M.Tawk, K. Z. Ibrahim, and S.Niar. Adaptive Sampling for Efficient MPSoC Architecture Simulation. *The 15th of the IEEE international Symposium on Modeling Analysis, and Simulation of Computer and Telecommunication Systems*, 2007.

[94] NEC Data sheet. `http://necel.com/memory/en/download/M11657EJCV0DS00.pdf`.

[95] G. Nicolescu. *Specification and validation for heterogeneous embedded systems*. PhD thesis, Laboratoire TIMA, UJF Grenoble, 2002.

[96] G. Nicolescu, K. Svarstad, W. O. Cesário, L. Gauthier, D. Lyonnard, S. Yoo, P. Coste, and A. A. Jerraya. Desiderata pour la spécification et la conception des systèmes électroniques. *Technique et Science Informatiques*, 21(3) :291–314, 2002.

[97] K. Nose and T. Sakurai. Analysis and future trend of short-circuit power. *Transactions on Computer-Aided design of integrated circuits and systems*, 19(9) :1023–1030, September 2000.

[98] S. Nussbaum and J. E. Smith. Statistical Simulation of Symmetric Multiprocessor Systems. In *35th Annual Simulation Symposium*, 2002.

[99] I. Ober, I. Ober, S. Graf, and D. Lesens. Projet OMEGA : un profil UML et un outil pour la modélisation et la validation de systèmes temps réel . 73 :33–38, juin 2005.

[100] Object Management Group, Inc., editor. *UML 2 Infrastructure (Final Adopted Specifcation)*. `http://www.omg.org/cgi-bin/doc?ptc/2003-09-15`, Sept. 2003.

[101] Object Management Group, Inc. MOF Query / Views / Transformations. `http://www.omg.org/docs/ptc/05-11-01.pdf`, Nov. 2005. OMG paper.

[102] Object Management Group, Inc., editor. *SysML v0.9*. `http://www.omg.org/cgi-bin/doc?ad/05-01-03`, Jan. 2005.

[103] Object Management Group, Inc., editor. *(UML) Profile for Schedulability, Performance, and Time Version 1.1*. `http://www.omg.org/technology/documents/formal/schedulability.htm`, Jan. 2005.

[104] U. of Berkeley (USA). Spice manual. `http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE/`.

[105] Open SystemC Initiative. SystemC. `http://www.systemc.org/`, 2007.

[106] I. M. Panades, A. Greiner, and A. Sheibanyrad. A low cost network-on-chip with guaranteed service well suited to the GALS approach. In *NanoNet*, 2009.

[107] P. R. Panda. *Designing Embedded Processors, A Low Power Perspective*, chapter Power Optimization Strategies Targeting the Memory Subsystem, pages 131 – 155. Springer, 2007.

[108] P. R. Panda and N. D. Dutt. Memory Architectures for Embedded Systems-On-Chip. In *9th International Conference High Performance Computing*, 2002.

[109] S. Pasricha, N. Dutt, and M. Ben-Romdhane. Fast exploration of bus-based on-chip communication architectures. In *CODES+ISSS'04*, Stockholm, Sweden.

[110] D. Patterson and J. Hennessy. *Computer Organization and Design : The Hardware/Software Interface, 3rd Edition*. Morgan Kaufmann, 2006.

[111] F. Petrot, A. Greiner, and P. Gomez. On Cache Coherency and Memory Consistency Issues in NoC Based Shared Memory Multiprocessor SoC Architectures. In *DSD '06 : Proceedings of the 9th EUROMICRO Conference on Digital System Design*, pages 53–60, 2006.

[112] F. Petrot, A. Greiner, and P. Gomez. On Cache Coherency and Memory Consistency Issues in NoC Based Shared Memory Multiprocessor SoC Architectures. In *DSD '06 : Proceedings of the 9th EUROMICRO Conference on Digital System Design*, 2006.

[113] Philips Electronic Design and Tools Group, Philips Research. DIESEL User Manual, version 2.5 edition, June 2001.

[114] E. Piel. *Ordonnancement de systèmes parallèles temps-réel, De la modélisation à la mise en œuvre par l'ingénierie dirigée par les modèles*. PhD thesis, „ France, Dec. 2007.

[115] C. Piguet. *Low-Power Electronics Design*. CRC Press, 2004.

[116] A. D. Pimentel. The artemis workbench for system-level performance evaluation of embedded systems. *Journal of Embedded Systems*, 1(7), 2005.

[117] R. E. C. Porto and L. V. Agostini. Project space exploration on the 2-D DCT architecture of a JPEG compressor directed to FPGA implementation. In *Design, Automation and Test in Europe Conference and Exhibition*, 2004.

[118] Power Analyzer home page. `www.eecs.umich.edu/panalyzer/`.

[119] S. project. An open platform for modelling and simulation of multi-processors system on chip. `http://soclib.lip6.fr/Home.html`.

[120] ProMarte partners. UML Profile for MARTE, Beta 1. `http://www.omg.org/cgi-bin/doc?ptc/2007-08-04`, Aug. 2007.

[121] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A UML 2.0 profile for SystemC : toward high-level SoC design. In *EMSOFT'05 : Proceedings of the 5th ACM international conference on Embedded software*, NJ, USA, 2005.

[122] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A model-driven design environment for embedded systems. In *DAC'06 : Proceedings of the 43rd annual conference on Design automation*, CA, USA, 2006.

[123] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand. Leakage current mechanisms and leakage reduction techniques in deep-submicrometer cmos circuits. *Proceedings of the IEEE*, 91(2) :305–327, February 2003.

[124] C. V. Schimpfe, S. Simon, and J. A. Nossek. High-level circuit modeling for power estimation. In *6th IEEE ICECS, International Conference on Electronics, Circuits and Systems*, 1999.

[125] E. Senn, J. Laurent, N. Julien, and E. Martin. SoftExplorer : estimation, characterization and optimization of the power and energy consumption at the algorithmic level. In *IEEE PATMOS*, Santorin, Grèce, 2004.

[126] M. Sgroi, L. Lavagno, and A. Sangiovanni-Vincentelli. Formal Models for Embedded System Design. *IEEE Design and Test of Computers*, 17(2) :14–27, June 2000.

[127] A. Sinha and A. Chandrakasan. Jouletrack - a web based tool for software energy profiling. In *Proceedings of the 38th DAC Conference*, 2000.

[128] A. Sinha and A. P. Chandrakasan. JouleTrack : a web based tool for software energy profiling. In *The 38th conference on Design automation*, NY, USA, 2001. ACM.

[129] SpecC Technology Open Consortium. SpecC. `http://www.specc.gr.jp/eng/index.html`, 2007.

[130] SPIRIT consortium. `www.spiritconsortium.org`.

[131] Synopsys. NanoSim datasheet (timemill and powermill) : Memory and mixed signal verification. `http://www.synopsys.com/products/mixedsignal/nanosim/nanosim_ds.pdf`.

[132] Synopsys. Power-Gate(TM) : a dynamic, simulation-based, gate-level power analysis tool. `Synopsys,http://www.synopsys.com/news/pubs/rsvp/fall97/rsvp_fall97_5.html`.

[133] H. Tardieu, A. Rochfeld, and R. Colletti. *La Méthode Merise : Principes et outils*. Editions d'Organisation, 1991.

[134] I. Tartalja and V. Milutinovi. Classifying Software-Based Cache Coherence Solutions. *IEEE Softw.*, 14(3) :90–101, 1997.

[135] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software : A first step towards software power minimization. In *Transactions on VLSI Systems*, 1994.

[136] M. Tomasevic and V. Milutinovic. Hardware Approaches to Cache Coherence in Shared-Memory Multiprocessors Part 2. *IEEE Micro*, 14(6) :61–66, 1994.

[137] A. Turier. *Etude, conception et caractérisation des mémoires CMOS, faible consommation, faible tension en technologies submicroniques*. PhD thesis, Université Pierre et Marie Curie, Paris, France, 2000.

[138] E. Viaud, F. Pecheux, and A. Greiner. An efficient TLM/T modeling and simulation environment based on parallel discrete event principles. In *Design, Automation and Test in Europe Conference and Exhibition*, 2006.

[139] K. Wakabayashi and T. Okamoto. C-Based SoC Design Flow and EDA Tools : An ASIC and System Vendor Perspective. *IEEE Computer on Computer-Aided Design of Integrated Circuits and Systems*, 19(12) :1507–1522, 2000.

[140] WEST Team LIFL, Lille, France. Graphical Array Specification for Parallel and Distributed Computing (GASPARD-2). `http://www.lifl.fr/west/gaspard/`, 2005.

[141] T. T. Ye, G. D. Micheli, and L. Benini. Analysis of power consumption on switch fabrics in network routers. In *The 39th Design Automation Conference*, 2002.

[142] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. Irwin. The Design and Use of SimplePower : A Cycle Accurate Energy Estimation Tool. In *Design Automation Conf*, June 2000.

Modèles et simulation des systèmes sur puce multiprocesseurs
Estimation des performances et de la consommation d'énergie

Résumé : La simulation des systèmes sur puce multiprocesseurs (MPSoC), dés les premières phases de conception, joue un rôle primordial puisqu'elle permet de réduire le temps d'arrivée sur le marché du produit final. Néanmoins, comme ces MPSoC deviennent de plus en plus complexes et hétérogènes, les méthodes conventionnelles de simulation de bas niveau ne sont plus adéquates. La solution proposée à travers cette thèse est l'intégration dans un seul environnement de plusieurs niveaux de simulation. Ceci permet l'évaluation des performances à un niveau précoce dans le flot de conception. L'environnement est utile dans l'exploration de l'espace des solutions architecturales et permet de converger rapidement vers le couple Architecture/Application le plus adéquat. Dans la première partie de cette thèse, nous présentons un outil de simulation performant et qui offre, à travers les trois niveaux qui le composent, différents compromis entre la vitesse de simulation et la précision de l'estimation des performances. Ces trois niveaux se différencient par les détails de l'architecture nécessaires à chacun et se basent sur le standard SystemC-TLM. Dans la deuxième étape, nous nous sommes intéressés à la consommation d'énergie dans les MPSoC. Pour cela, nous avons enrichi notre environnement de simulation par des modèles de consommation d'énergie flexibles et précis. Enfin dans la troisième étape de notre thèse, une chaîne de compilation basée sur la méthodologie Ingénierie Dirigée par les Modèles (IDM) est développée et intégrée à l'environnement Gaspard. Cette chaîne permet la génération automatique du code SystemC à partir d'une modélisation de haut niveau d'un MPSoC.

Mots clefs : Système sur puce multiprocesseur (MPSoC), accélération de la simulation, SystemC, CABA, TLM, estimation de performances, consommation d'énergie, ingénierie dirigée par les modèles, flot de conception.

Multiprocessor system-on-chip modeling and simulation
Performance and energy consumption estimation

Abstract: Multiprocessor system on chip (MPSoC) simulation in the first design steps has an important impact in reducing the time to market of the final product. However, MPSoC have become more and more complex and heterogeneous. Consequently, traditional approaches for system simulation at lower levels cannot adequately support the complexity needed for the design of future MPSoC. In this thesis, we propose a framework composed of several simulation levels. This enables early performance evaluation in the design flow. The proposed framework is useful for design space exploration and permits to find rapidly the most adequate Architecture/Application configuration. In the first part of this thesis, we present an efficient simulation tool composed of three levels that offer several performance/energy tradeoffs. The three levels are differentiated by the accuracy of architectural descriptions based on the SystemC-TLM standard. In the second part, we are interested by the MPSoC energy consumption. For this, we enhanced our simulation framework with flexible and accurate energy consumption models. Finally in the third part, a compilation chain based on a Model Driven Engineering (MDE) approach is developed and integrated in the Gaspard environment. This chain allows automatic SystemC code generation from high level MPSoC modeling.

Keywords: Multi-processor systems-on-chip (MPSoC), simulation speedup, SystemC, CABA, TLM, performance estimation, energy consumption,model driven engineering, design flow.