Modèles de simulation pour la validation logicielle et l'exploration d'architectures des systèmes multiprocesseurs sur puce

P. Gerin

INSTITUT POLYTECHNIQUE DE GRENOBLE

Numéro attribué par la bibliothèque
978-2-84813-142-9

THÈSE

pour obtenir le grade de
DOCTEUR DE L'Institut Polytechnique de Grenoble
Spécialité : Micro et Nano Électronique

préparée au laboratoire **TIMA**
dans le cadre de **l'École Doctoral " Électronique, Électrotechnique, Automatique et Traitement du Signal "**

présentée et soutenue publiquement par

Patrice Gerin

le 30 novembre 2009

Titre :

Modèles de simulation pour la validation logicielle et l'exploration d'architectures des systèmes multiprocesseurs sur puce

---

sous la direction de Frédéric Pétrot

JURY

Pr. Florence Maraninchi	Présidente
Pr. Paul Feautrier	Rapporteur
Pr. Guy Bois	Rapporteur
Dr. Laurent Maillet-Contoz	Examinateur
Pr. Alain Greiner	Examinateur
Dr. Benoît Dupont de Dinechin	Invité
Pr. Frédéric Pétrot	Directeur

Remerciements

C'est un soir de novembre 2004, en pleine remise en question sur mon avenir professionnel, que Gabriela Nicolescu me suggère de contacter M. Ahmed Amine Jerraya, alors directeur de l'équipe SLS du laboratoire TIMA, afin d'entreprendre un travail de thèse. Je tiens donc à remercier en premier lieu Gabriela, pour m'avoir convaincu de reprendre mes études et Ahmed de m'avoir accueilli dans son groupe de recherche.

C'est finalement Frédéric Pétrot, ayant repris la direction du groupe SLS, qui m'aura encadré au cours de ces trois années de thèse. Je lui suis très sincèrement reconnaissant pour la qualité de son encadrement, ces nombreux conseils et tout le temps qu'il a consacré à ce travail.

Je remercie également l'ensemble des membres de mon jury de thèse, à commencer par Florence Maraninchi, professeur à Grenoble INP et responsable du groupe *synchrone* au laboratoire VERIMAG, qui m'a fait l'honneur de le présider. Merci à Paul Feautrier, professeur émérite à l'École Nationale Supérieure de Lyon et à Guy Bois, professeur titulaire à l'École Polytechnique de Montréal d'avoir accepté de rapporter sur mon travail. Merci également à Alain Greiner, professeur à l'Université Pierre et Marie Curie et directeur du département SoC au laboratoire LIP6, Laurent Maillet-Contoz, CAD Manager à STMicroelectronics et Benoît Dupont de Dinechin, architecte processeur et responsable d'équipe compilation chez Kalray d'avoir participé à mon jury de thèse en tant qu'examinateurs.

Merci à tout les membres et amis du groupe SLS avec qui j'ai passé ces quatres années. Je tiens tout particulièrement à adresser mes remerciement à Xavier Guérin et à Pierre Guironnet de Massas, pour les moments de travails acharnés, les nuits blanches passées au labo, les moments de déprime que nous avons pu traverser ensemble, mais aussi et surtout pour tout les instants de « rigolades » innoubliables, qu'ils soient assurés de mon amitié sincère. Un merci particulier également à Marius Gligor et ces sermons sur Windows (en vain ☺), à Nicolas Fournel pour ces discutions constructives ... ou pas !, à Quentin Meunier, Olivier Muller, Damien Hedde et Pierre-Henri Horrein. Bon courage à Mian Muhammad HAMAYUN qui a accepté de poursuivre ces travaux de recherche.

Merci à mes voisins de bureau, Alexandre Chureau, Marius Bonacciu, à mes voisin de couloir, Aimen et Lobna Bouchhima, Wassim Youssef, Iuliana Bacivarov, Katalin Popovicci, Amin Elmrabti, Lilia Sfaxi, Alexandre Chagoya-Garzon, Hao shen, Hui chen et à tout ceux que j'ai côtoyé.

Merci à Frédéric Rousseau et Paul Amblard, pour tous les conceils qu'ils m'ont

apportés lors de cette thèse et également lors de la rédaction de ce manuscrit. Je remercie également mes courageux correcteurs orthographiques, ma mère (que cette thèse aura certainement fait stresser autant que moi) et Corinne Molinari qui, sans vouloir remettre en cause leurs compétences dans la modélisation des systèmes multiprocesseurs sur puce, ont bien voulu corriger un document leur semblant parfois être écrit dans un dialecte encore inconnu !!!

Je finirai par une pensée toute particulière pour mon épouse Dorothée qui ma accompagné et supporté tout au long de cette thèse, à ma fille Lilou et à mon fils Nolan d'avoir du endurer mes absences.

Table des matières

Liste des tableaux

Table des figures

Chapitre 1

Introduction

De l'invention du circuit intégré à nos jours, l'industrie de la micro-électronique doit son succès essentiellement à la miniaturisation du transistor sur silicium. Pendant près de 40 ans, cette miniaturisation a été le principal facteur ayant permis de concevoir des systèmes intégrés toujours plus complexes. Les densités d'intégration et les procédés technologiques permettent aujourd'hui d'embarquer des systèmes complets sur une seule puce (mémoires, processeurs, périphériques, etc.), appelés *System On Chip (SoC)*.

Ces systèmes se retrouvent désormais dans la quasi totalité des appareillages électroniques, notamment dans les applications de communication ou grand public telles que la photo numérique, les consoles de jeux portables ou encore la téléphonie mobile. Ces applications ont par ailleurs des contraintes de plus en plus fortes en terme de puissance de calcul ou de consommation auxquelles les concepteurs doivent faire face, le tout accentué par des temps de mise sur le marché et de cycle de vie de plus en plus courts.

Cependant, la diminution des dimensions dans les systèmes intégrés s'accompagne de l'augmentation de certains facteurs tels que la puissance consommée ou le délai de propagation dans les interconnexions. Ces facteurs ne sont aujourd'hui plus négligeables et cette même miniaturisation qui permettait d'augmenter significativement et de manière relativement simple les performances s'essouffle. Face à ce constat et afin de répondre au mieux aux exigences du marché, de nouvelles solutions doivent être mises en œuvre telles que l'utilisation de nouveaux matériaux ou encore de nouvelles solutions architecturales.

La solution architecturale "la plus évidente" consiste à embarquer plusieurs processeurs sur une même puce pour former un *MultiProcessors System On Chip (MPSoC)*. Il existe deux types d'architectures multiprocesseurs :

Les architectures symétriques (dites SMP) mettent en œuvre des processeurs identiques partageant le même espace mémoire. Un système d'exploitation partagé contrôle tous les processeurs et permet l'exécution de n'importe quelle tâche ou application sur n'importe lequel d'entre eux. L'utilisation de plusieurs processeurs en parallèle fonctionnant à des fréquences plus basses permet d'atteindre les performances d'un seul processeur fonctionnant à une fréquence plus élevée tout en ayant une consommation d'énergie moindre.

Initialement issues du marché des machines haute performance (HPC), ces architectures ont d'abord été utilisées dans les applications réseau et sont maintenant introduites dans le marché des applications portables. Le processeur ARM Cortex^{TM}-A9 MPCore pouvant contenir jusqu'à quatre processeurs en est un exemple (figure 1.1).

Même si ces architectures ne sont pas toujours optimales, elles permettent d'atteindre des performances intéressantes, en fonction du nombre de processeurs et de la capacité du logiciel à être parallélisé, tout en conservant une certaine simplicité de programmation.

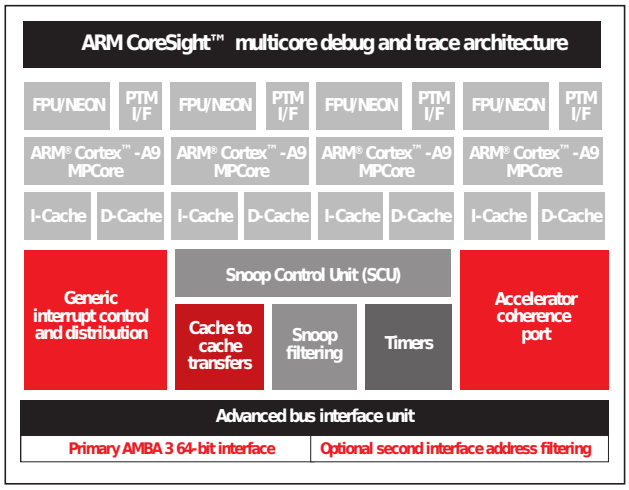


FIGURE 1.1 – ARM-Cortex^{TM}-A9 MPCore

Les architectures asymétriques ou hétérogènes (dites AMP) permettent d'abaisser encore les niveaux de puissance consommée tout en augmentant les performances par rapport aux architectures symétriques grâce à l'utilisation de processeurs spécialisés dans le type de tâche à réaliser. On retrouve très fréquemment dans ce type d'architecture l'association de processeurs généralistes (GPP) pour réaliser les tâches de contrôle avec des processeurs de signal (DSP) pour les tâches de calcul intensif.

La figure 1.2 représente l'architecture du SoC DaVinci TMS32DM365^{TM} de Texas Instruments, dédiée aux systèmes multimédias sur laquelle on retrouve deux processeurs ARM Cortex^{TM}-A9 MPCore et plusieurs processeurs spécialisés enfouis dans les blocs d'accélération matérielle et de traitement d'image.

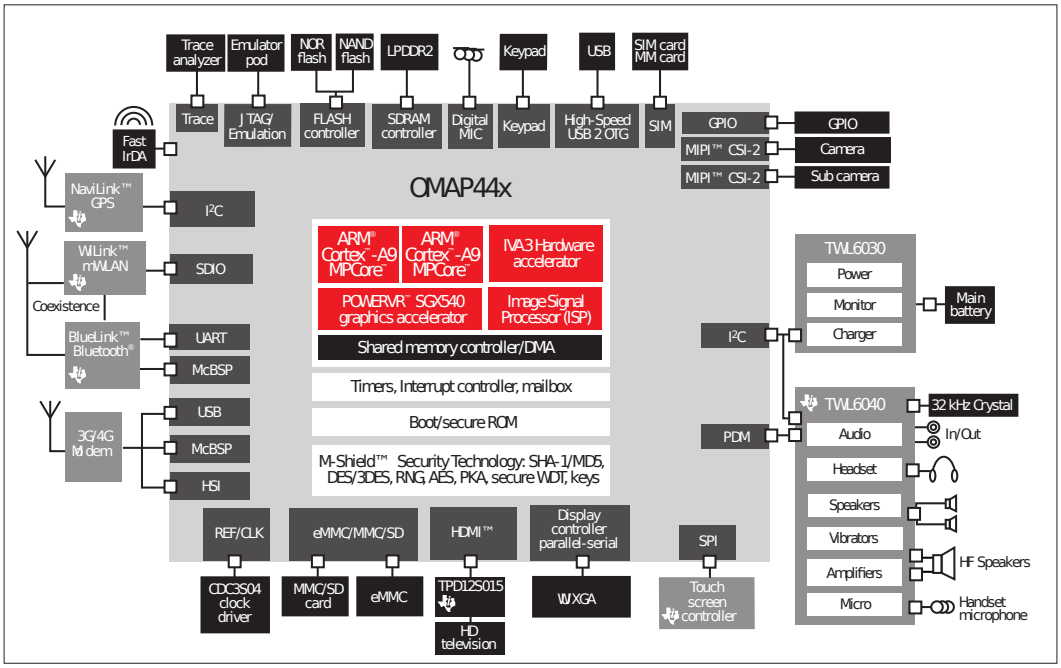
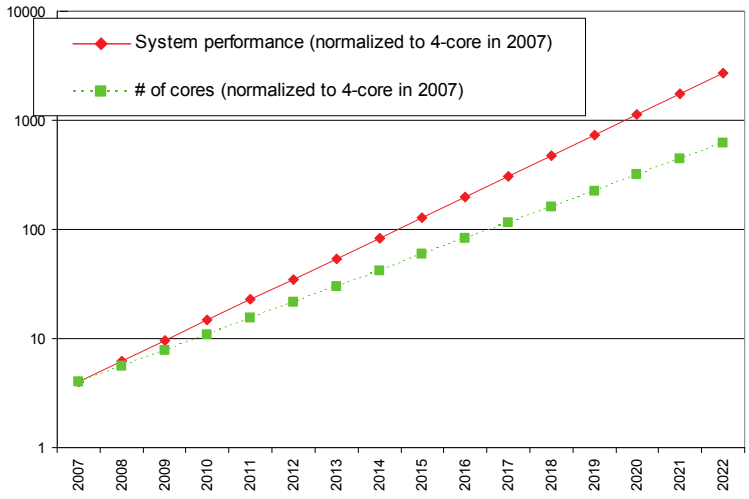
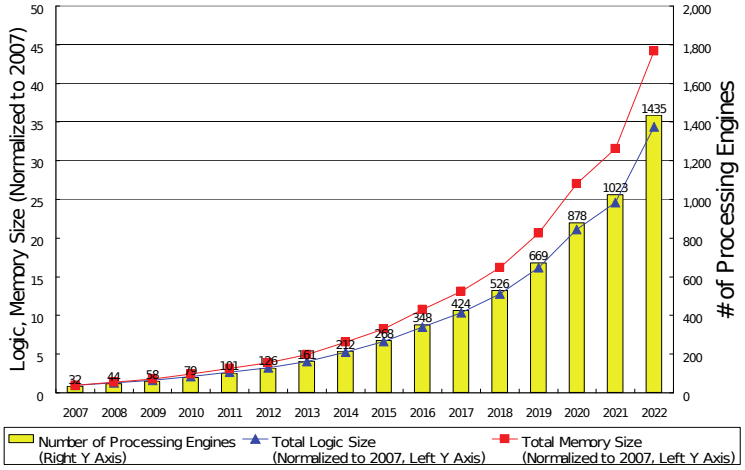


FIGURE 1.2 – MPSoC Texas Instruments DaVinci TMS320DM365^{TM}

Le fait de réaliser de plus en plus de fonctions par logiciel plutôt que d'utiliser des composants matériels dédiés a certes un coût en terme de performance mais qui est compensé par la flexibilité offerte, aussi bien durant le flot de conception du système que pendant sa durée de vie (correction d'erreurs, évolutions de fonctionnalités). Ainsi, la tendance actuelle consiste à embarquer de plus en plus de processeurs et les prévisions stratégiques du rapport ITRS montrent que ce nombre devrait continuer d'augmenter fortement dans le futur [ITR07]. Les figures 1.3(a) et 1.3(b) qui en sont tirées représentent l'estimation de cette tendance pour les applications réseau et grand public portables.



(a) SoC dédiés aux applications réseaux



(b) SoC dédiés aux applications grand public portables

FIGURE 1.3 – Évolution du nombre de processeurs dans SoC futur (source [ITR07])

Cette dominance du logiciel dans les architectures MPSoC, pouvant représenter jusqu'à 80% du coût total de conception, nous contraint à commencer la validation et l'intégration avec le matériel dès les premières étapes des flots de conception, bien avant de disposer des premiers prototypes matériels, afin de minimiser les risques d'erreurs et les temps de mise sur le marché. Dans la version mise à jour en 2008 de son rapport [ITR08], l'ITRS confirme le logiciel comme étant « *une partie à part entière des systèmes micro électroniques, et la productivité de conception du logiciel comme un facteur clé dans la productivité de l'ensemble d'un produit* ».

La simulation est la principale méthode permettant la validation et l'exploration d'architectures dans la conception des systèmes intégrés. Elle a représenté un axe important de recherche depuis les débuts de la micro-électronique, puisqu'elle est le seul moyen de valider un circuit ne pouvant pas être décrit analytiquement ou dont la complexité ne permet pas aux concepteurs de le faire, soit en pratique, tous les systèmes intégrés.

Historiquement, cette simulation ne concerne que le matériel. L'évolution des langages de description de matériel (HDL) vers des niveaux plus abstraits a permis de répondre à des besoins fondamentaux pour les concepteurs, tels que la réduction des temps de conception, la vitesse de simulation ou encore la réutilisabilité. On est ainsi passé de la simulation au niveau portes logiques du système à la simulation RTL (Register Transfer Level) pour arriver ensuite aux langages VHDL ou Verilog permettant de couvrir différents niveaux de description du système en vue de sa simulation ou de sa synthèse.

Même s'il est possible d'exécuter un programme sur le modèle VHDL d'un processeur et de son environnement, ce programme n'est en fait qu'une séquence de vecteurs de tests permettant de valider le comportement matériel du composant, en aucun cas il ne permet la validation d'une application logicielle complète (vitesse de simulation faible, pas d'outils de débogage adaptés au logiciel).

Les outils de co-simulation ont permis de répondre au besoin de simulation du logiciel en séparant la modélisation du processeur de celle du matériel environnant. Celui-ci peut alors être modélisé par un simulateur plus abstrait, réalisé dans un langage non dédié au matériel et donc beaucoup plus performant. Ces simulateurs de processeurs sont appelés des ISS (pour Instruction Set Simulator). Un ISS est la mise en œuvre par du logiciel de l'algorithme d'interprétation des instructions d'un processeur, algorithme qui dans le processeur réel est évidemment mis en œuvre par du matériel. L'environnement matériel du processeur reste décrit en langage HDL. Les deux simulateurs s'exécutent de manière conjointe et communiquent par des mécanismes dédiés (typiquement des IPC) afin de permettre au logiciel simulé sur un ISS d'interagir avec le matériel.

L'utilisation de langages de programmation de logiciel tel que le C/C++ pour la description du matériel a permis de franchir un nouveau cap dans la validation des systèmes intégrés. Ces solutions permettent notamment d'atteindre des vitesses de simulation beaucoup plus élevées grâce à l'utilisation d'un seul environnement pour la simulation des ISS et de leurs périphériques. SystemC et SpecC en sont deux exemples.

Des approches plus récentes ont tout simplement pour objectif de supprimer l'utilisation d'ISS, dans le but d'atteindre des performances de simulation toujours plus élevées. Partant du principe que toute l'architecture matérielle est modélisée dans un langage tel que SystemC, c'est à dire en langage C++, il ne reste qu'un pas à franchir pour compiler également le code logiciel avec l'architecture matérielle, lui aussi développé en langage C/C++ ou dans un langage pouvant s'interfacer facilement avec celui-ci, pour se passer de l'utilisation d'ISS. L'exécution de code logiciel directement sur le processeur de la station de travail est beaucoup plus rapide qu'une interprétation faite par un ISS, mais il est alors beaucoup plus difficile de prendre en compte les aspects dépendant de l'architecture matérielle réelle dans la simulation.

Dans ce type d'approche, nous parlerons d'*exécution native* de logiciel. Le processeur de la station de travail sera appelé *processeur hôte*, et le ou les processeurs de l'architecture matérielle réelle seront appelés *processeurs cible*.

Ceci dresse le contexte de cette thèse dans laquelle nous nous attacherons à contribuer à la mise au point de nouveaux modèles de simulation, basés sur l'exécution native du logiciel, permettant (1) la validation fonctionnelle du logiciel s'exécutant sur des architectures MPSoC, ce qui nécessite une modélisation fine du matériel ainsi que du logiciel qui le pilote et (2) l'exploration d'architectures nécessaire à la recherche d'une solution architecturale optimale, ce qui requiert une vitesse de simulation importante tout en conservant une précision acceptable.

Dans le chapitre qui va suivre, nous présenterons le contexte de cette thèse de manière plus précise ainsi que la problématique liée à l'exécution native du logiciel. Nous en tirerons alors une liste de questions auxquelles nous nous attacherons à apporter des réponses au cours de ce document, en commençant par l'état de l'art des solutions déjà apportées et qui seront discutées dans le chapitre 3.

Les chapitres 4 et 5 présentent les deux principales contributions de cette thèse. La première est une méthode de conception de plateforme de simulation, basée sur l'exécution native du logiciel et permettant, malgré le niveau d'abstraction utilisé, l'exécution de la quasi totalité du logiciel final (système d'exploitation, pilotes de périphériques, application, etc.) sur des modèles réalistes de l'architecture matérielle du système. La seconde contribution est une technique d'instrumentation permettant l'annotation du logiciel s'exécutant sur une plateforme de simulation native. Le chapitre 6 présente deux exemples d'applications des techniques proposées dans ce travail, l'une permettant d'estimer sur la plateforme native les performances temporelles du logiciel s'il s'était réellement exécuté sur la plateforme finale et l'autre permettant de profiler le logiciel multi-tâches, exécuté sur des architectures multiprocesseurs.

Dans les expérimentations du chapitre 7, nous mettrons en évidence l'efficacité et les limitations des techniques proposées dans les contributions permettant de modéliser de manière précise les détails architecturaux et de les prendre en compte lors de l'exécution native du logiciel. Nous conclurons enfin ce manuscrit par le bilan de cette thèse, et une ouverture vers les perspectives envisageables à ce travail de recherche.

Chapitre 2

Problématique

Sommaire

De nombreux travaux de recherche ont mis en évidence la nécessité d'utiliser des modèles de simulation à différents niveaux d'abstraction, apportant chacun des réponses à un problème de conception spécifique. Un des objectifs de ce chapitre est de présenter le niveau d'abstraction auquel nous nous intéressons et de le positionner par rapport aux niveaux existants. Nous définirons ensuite la problématique liée à l'utilisation de tels modèles de simulation pour finalement dresser une liste de questions auxquelles cette thèse devra répondre.

Afin de mieux positionner ce travail dans son contexte, nous commencerons par présenter l'architecture générique des MPSoC considérés dans ce travail, ainsi que les différents concepts de base nécessaires à sa compréhension.

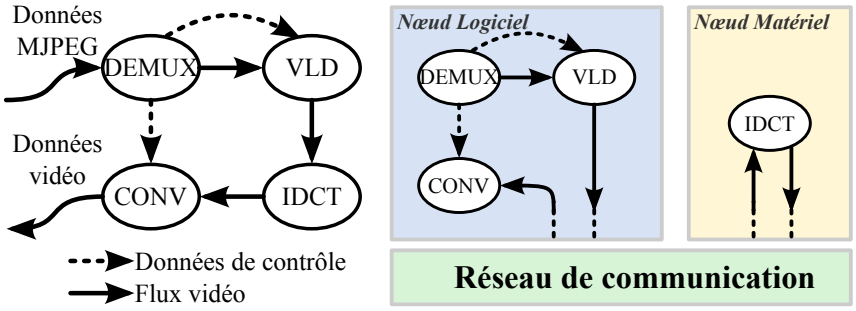
2.1 Contexte

2.1.1 Architecture MPSoC générique

Dans cette thèse, un système multiprocesseur peut-être considéré d'une manière générale comme un ensemble de nœuds de calcul, pouvant être matériels ou logiciels, auxquels sont affectées statiquement ou dynamiquement les tâches à réaliser en fonction de différents critères tels que les performances de traitement requises ou encore la puissance consommée.

Un nœud de calcul est dit matériel lorsqu'il ne présente (tout du moins d'un point de vue externe) aucune capacité de programmation. Ils correspondent généralement à des accélérateurs dédiés pour des fonctions critiques en temps d'exécution. Un nœud de calcul est dit logiciel lorsque les tâches qui lui sont affectées sont réalisées par des éléments programmables.

La figure 2.1 donne un exemple de répartition des tâches d'une application de décodage vidéo MJPEG dont la spécification simplifiée est donnée figure 2.1(a). Dans cet exemple, la répartition a été faite sur un nœud logiciel et un nœud matériel interconnectés par un réseau de communication. La tâche *IDCT*[1] du MJPEG, qui peut représenter plus de la moitié du temps d'exécution de l'application lorsqu'elle est réalisée par logiciel, est ici affectée à un nœud matériel. Les autres tâches, aux vues des contraintes temporelles de l'application, sont réalisées sur un nœud logiciel.



(a) Spécification de l'application MJPEG

(b) Exemple d'architecture MPSoC

FIGURE 2.1 – Répartition des tâches de l'application MJPEG sur une architecture MPSoC

Cette représentation des architectures MPSoC sous forme de nœuds de calcul permet de modéliser de manière générique tout type d'architectures (SMP et AMP).

2.1.2 Architecture d'un nœud logiciel

Un nœud logiciel offre tout l'environnement nécessaire à l'exécution des tâches qui lui ont été attribuées. La figure 2.2 représente l'architecture du nœud logiciel. Il est lui même composé d'une partie matérielle appelée *sous-système processeur* et d'une partie logicielle appelée dans ce travail *pile logicielle*.

1. Inverse Discrete Cosine Transform /Transformée en Cosinus Discrète Inverse

2.1.2.1 Le sous-système processeur

Le sous-système processeur regroupe l'ensemble des composants « classiques » permettant l'exécution du logiciel, tel que les mémoires, contrôleurs d'interruptions, DMA et les périphériques d'entrées/sorties vers le réseau d'interconnexion (figure 2.2).

Notre définition du nœud logiciel est telle que tous les processeurs qui le composent sont identiques, ou a minima, supportent le même jeu d'instructions et exécutent la même pile logicielle. Ce type d'architecture est dit SMP (Symmetric Multiple Processor). Les architectures hétérogènes peuvent être modélisées par l'utilisation de plusieurs nœuds logiciels.

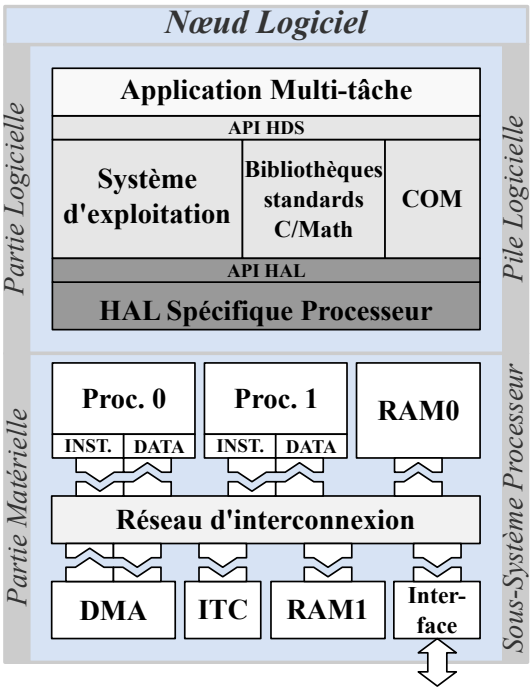


FIGURE 2.2 – Architecture d'un nœud logiciel

2.1.2.2 La pile logicielle

Face à la complexité croissante des systèmes embarqués, l'organisation du logiciel en couches, telle qu'elle est utilisée dans les systèmes informatiques classiques [Tan84], est une solution couramment employée afin de simplifier son développement. Cet empilement de couches logicielles représenté sur la figure 2.2 est appelé « *pile logicielle* » :

- La couche applicative contient une ou plusieurs tâches réalisant la partie fonctionnelle du logiciel embarqué qui s'exécutent de manière concurrente en vrai ou faux parallélisme suivant le nombre de processeurs disponibles dans le sous-système.
- La couche contenant le système d'exploitation (SE) et les différentes bibliothèques est généralement appelée HDS pour Hardware Dependent Software [JBP06, DGM09]. Elle est dépendante du matériel dans le sens où elle est dédiée à un sous-système processeur donné, mais son implémentation doit rester autant que possible indépendante de celui-ci.
- La couche d'abstraction du matériel, appelée HAL pour Hardware Abstraction Layer, utilisée dans le logiciel embarqué sur des SoC [YJ03], est là encore issue de techniques classiquement employées dans la conception de systèmes d'exploitation.

Cette organisation en couche peut cependant s'avérer inefficace dans des systèmes très spécifiques à base de DSP par exemple, qui n'embarquent généralement pas de systèmes d'exploitation complexes. Elle est en revanche indispensable dès lors que l'on souhaite exécuter une application multi-tâches et offre une flexibilité en terme de conception et de développement non négligeable.

2.1.3 Terminologie

2.1.3.1 Plateformes

Le terme plateforme est employé ici pour désigner l'ensemble du matériel, permettant l'exécution du logiciel (processeurs et périphériques). Une plateforme peut être réelle (prototype à base de FPGA, SoC final) ou virtuelle, permettant de modéliser et de simuler le matériel avec le logiciel s'exécutant sur celui-ci. Dans ce document, toutes les plateformes sont virtuelles, nous les désignerons également par *plateformes de simulation*.

2.1.3.2 Processeur cible vs processeur hôte

Le processeur cible désigne le type de processeur qui sera finalement embarqué dans le SoC. Il s'agit généralement d'un processeur dédié au domaine des systèmes embarqués. Le processeur hôte correspond au processeur de la machine sur laquelle sont exécutées les plateformes de simulation. Son architecture est généralement différente de celles des processeurs cibles. Un processeur cible peut être simulé dans une plateforme virtuelle, s'exécutant sur le processeur hôte.

2.1.3.3 Exécution vs exécution native

Sans précision particulière, le terme « *exécution* » désigne l'exécution du logiciel de l'application sur le processeur cible. On parle d'exécution native lorsque celle-ci est directement exécutée par le processeur de la machine hôte (après re-compilation du code source). L'exécution du logiciel sur un processeur cible simulé par un ISS n'est pas une exécution native.

2.2 Langages de description des modèles de simulation

Les approches basées sur les langages de programmation de logiciel pour la description de composants matériels fournissent l'abstraction nécessaire pour faciliter la conception, l'exploration d'architecture ou encore la validation des MPSoC rendue quasiment impossible sur des modèles au niveau transfert de registre (RTL).

Depuis maintenant plusieurs décennies, les concepteurs modélisent leurs systèmes en utilisant des langages tels que C/C++. Les premières approches ont rapidement montré leurs inconvénients liés principalement à une sémantique inadaptée au domaine matériel, notamment pour la modélisation du parallélisme, des informations structurelles (interface/comportement) ou encore de la notion de temps. Plusieurs travaux de recherches ont proposé des langages basés sur le C/C++ ou des extensions, disposant de la sémantique nécessaire à la modélisation du matériel, afin d'en permettre la validation et/ou la synthèse [KD90, GL97, DM99, VKS00].

Cette dernière décennie a vue l'émergence de langages de description de matériel basés sur le C++, tels que SystemC [SYSa], ou sur un langage inspiré de celui tel que SpecC

[GZD+00]. SystemC est devenu aujourd'hui un langage incontournable dans la conception des systèmes sur puce qui s'étend maintenant également aux systèmes sur carte [RGC+09]. Les plateformes de simulation présentées dans cette thèse sont toutes réalisées en SystemC.

2.2.1 SystemC

SystemC [SYSa] est un langage de description de matériel au même titre que VHDL ou Verilog. Bien que l'on parle de « *langage* », SystemC est en fait une bibliothèque de classe C++ contenant tous les éléments structurels permettant de modéliser des systèmes électroniques, ainsi qu'un moteur de simulation évènementiel (appelé également noyau de simulation). Un modèle SystemC est donc un programme exécutable et une simulation SystemC consiste tout simplement à exécuter ce programme.

2.2.2 Organisation structurelle de SystemC

La bibliothèque SystemC est constituée d'un ensemble de classe C++ de base permettant de modéliser structurellement une architecture sous forme de composants. Les principaux éléments de SystemC sont les modules, les interfaces, les canaux et les ports.

- **Le module** : c'est l'élément de base de SystemC à partir duquel tout composant est modélisé. Son rôle est uniquement structurel et il ne possède aucune sémantique d'exécution. Des canaux de communications permettent de connecter des modules entre eux par l'intermédiaire de ports.
- **L'interface** : La notion d'interface est primordiale en SystemC puisqu'elle garantit l'indépendance entre l'implémentation d'un mécanisme de communication et sa définition. Cette notion correspond d'ailleurs exactement à celle utilisée en programmation logicielle et elle est mise en œuvre de la même manière. Ainsi une interface est une classe C++ dérivée de la classe SystemC `sc_interface`, et dans laquelle on ne trouve que la déclaration de méthodes virtuelles pures. Ces méthodes sont en quelque sorte la signature de l'interface de communication.
- **Le canal** : c'est une classe C++ qui dérive d'une ou plusieurs interfaces et qui en définit les méthodes, fournissant ainsi le mécanisme de communication.
- **Les ports** : ils permettent aux canaux et aux modules de fournir et d'accéder à ces mécanismes de communication. On distingue donc deux types de ports SystemC. Une interface de communication est fournie au travers d'un port SystemC (objet C++) de type `sc_export` et elle est requise par l'intermédiaire d'un port de type `sc_port`.

La partie supérieure de la figure 2.3 représente le diagramme de classe UML simplifié de deux interfaces. Une interface `Itf_write`, déclarant une méthode virtuelle pure `write()` et une autre `Itf_read`, déclarant de la même manière une méthode `read()`.

La partie inférieure de la figure 2.3 représente quant à elle la vue structurelle d'une architecture SystemC composée de deux modules interconnectés par un canal de communication. Le canal `Canal_A` dérive des deux interfaces et implémente le comportement des deux méthodes `write()` et `read()`. Les ports du canal sont des instances d'objet `sc_export` dont le type template[2] est celui des interfaces qu'ils fournissent. Le canal contient donc un port de type `sc_export<Itf_write>` par lequel un module pourra accéder à la mé-

2. Un template est un patron définissant un modèle de fonction ou de classe dont certaines parties sont des paramètres (type traité, taille maximale)

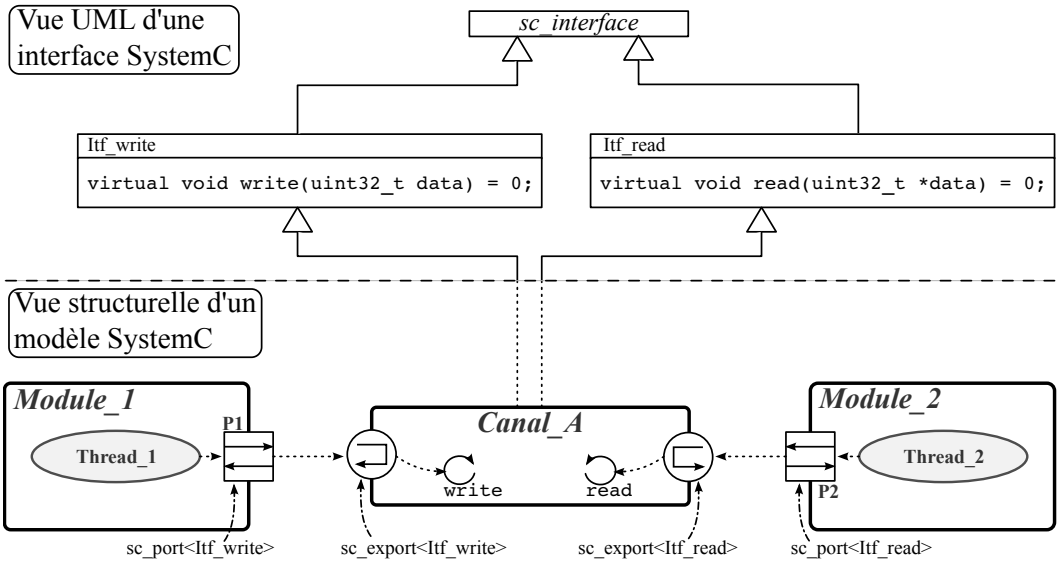


FIGURE 2.3 – Éléments structurels de SystemC et interfaces de communication

thode `write()`, et un autre de type `sc_export<Itf_read>` par lequel on pourra accéder à la méthode `read()`.

Un module est une classe C++ dérivée de la classe de base `sc_module`. Les ports d'un module sont des instances d'objets C++ `sc_port` dont le type template est celui des interfaces qu'ils requièrent. Le module `Module_1` contient donc un port de type `sc_port<Itf_write()>`, et le module `Module_2` un port de type `sc_port<Itf_read()`.

La construction d'une architecture SystemC (appelée élaboration) se déroule en deux phases :

- L'instanciation : elle correspond à la création des différents modules et canaux constituant l'architecture.
- La connexion : elle consiste à relier les ports des modules avec ceux des canaux correspondants.

2.2.3 Éléments fonctionnels de SystemC

Les éléments fonctionnels de SystemC se trouvent soit dans les modules, soit dans les canaux. Dans les canaux, ces éléments correspondent directement aux méthodes réalisant le comportement des interfaces.

Dans les modules, les éléments fonctionnels permettant de modéliser le comportement sont les processus. Ce sont des méthodes membres des modules, dont l'exécution est prise en charge par le moteur de simulation de SystemC. Les processus d'un module peuvent communiquer avec ceux d'un autre module par l'intermédiaire des ports et des canaux. Pour cela, la méthode de communication implémentée dans un canal est directement appelée sur le port du module correspondant. Dans la figure 2.3, le processus `thread_1` pourra appeler la méthode `write()` directement sur l'instance du port `P1` de la manière suivante : `P1->write(valeur)`.

La synchronisation se fait par l'utilisation d'évènements ainsi que la fonction de mise en veille `wait()`. Lorsqu'un processus appelle cette fonction, il est mis en veille pour une durée déterminée ou jusqu'à ce qu'un évènement auquel il est sensible (par exemple une communication dans un canal) soit notifié. L'ordonnanceur du moteur de simulation de SystemC se charge d'appeler les processus en fonction des évènements auxquels ils sont

sensibles et ne préempte pas leurs exécutions. Ainsi, un processus est exécuté jusqu'à ce qu'il redonne lui-même le contrôle au moteur de simulation grâce à la fonction `wait()`.

Entre deux appels à la fonction `wait()`, le code du processus est exécuté de manière atomique, c'est à dire en temps nul du point de vue de la simulation. Seuls les appels à la fonction `wait()` permettent de faire évoluer le temps. En d'autres termes, tout évènements matériel ne pourra être pris en compte que lors d'appels à cette fonction, servant ainsi de point de synchronisation.

2.2.4 Abstraction en SystemC

À partir de ces éléments structurels de base, il est possible en SystemC de modéliser des systèmes électroniques à différents niveaux d'abstraction. Un canal peut permettre de représenter par exemple un fil ou un groupe de fils. On peut ainsi modéliser très précisément les connexions entre deux composants matériels comme le montre la figure 2.4(a). Sur cette figure, plusieurs signaux (données, adresses et contrôles) sont utilisés pour connecter l'interface d'une mémoire avec un autre composant. Il existe pour cela un type prédéfini en SystemC nommé `sc_signal`, et permettant de modéliser des fils ou des bus de différentes tailles. Pour effectuer des transferts, les deux composants devront modéliser précisément le protocole de communication, généralement sous forme de machine à états.

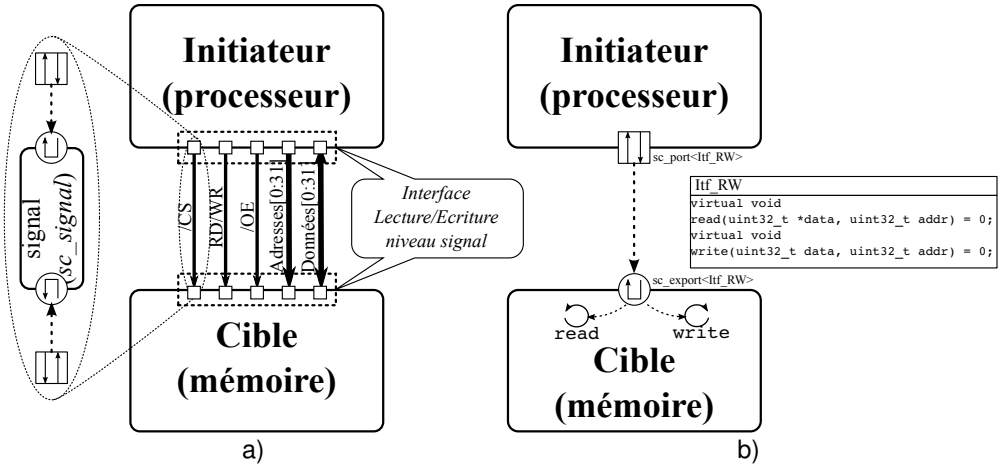


FIGURE 2.4 – Modélisation TLM en SystemC

Cet ensemble de signaux peut également être vu comme une seule entité permettant à ces deux composants de communiquer. Il est alors possible de modéliser tous ces signaux par une seule connexion rendant le même service sous forme d'une méthode d'interface plus généraliste. Le protocole de communication n'est alors plus modélisé par des signaux, mais directement par des appels de fonctions (figure 2.4(b)).

En général, l'interface n'est plus modélisée par un canal de communication, mais directement par le module fournissant le service. Dans cet exemple, le module mémoire dérive de l'interface `RW` et implémente les méthodes `read()` et `write()`. Lors de l'élaboration de l'architecture, le port du composant mémoire est directement connecté au port du composant processeur. Ce principe de modélisation est appelé TLM pour Transaction Level Modeling.

De manière complètement anecdotique, un signal SystemC peut lui-même être vu comme la modélisation TLM d'un fil électrique. On utilise en effet le même principe d'interfaces C++ et d'appel de méthode, pour l'affectation d'un potentiel sur un signal électrique et la lecture de celui-ci.

2.3 Niveaux d'abstractions et modèles de simulation classiques

L'utilisation de plusieurs niveaux d'abstraction permettant de passer graduellement d'une spécification à une implantation finale est indispensable pour la conception de systèmes tels que les MPSoC. L'utilisation de langages de programmation issus du domaine logiciel pour l'implantation des modèles de simulation matériels a été un facteur clé ayant permis la définition des différents (pour ne pas dire multiples) niveaux d'abstraction existants aujourd'hui. A chaque niveau d'abstraction est associé un modèle de simulation permettant de valider tout ou partie de l'application avec plus ou moins de précision architecturale et temporelle.

La figure 2.5 donne les principaux niveaux d'abstraction ainsi qu'une représentation des modèles de simulation correspondants. Dans la suite, nous présentons rapidement ces principaux niveaux et modèles, le but n'étant pas de faire une étude détaillée, mais de positionner le modèle de simulation proposé dans ce travail parmi les modèles existants.

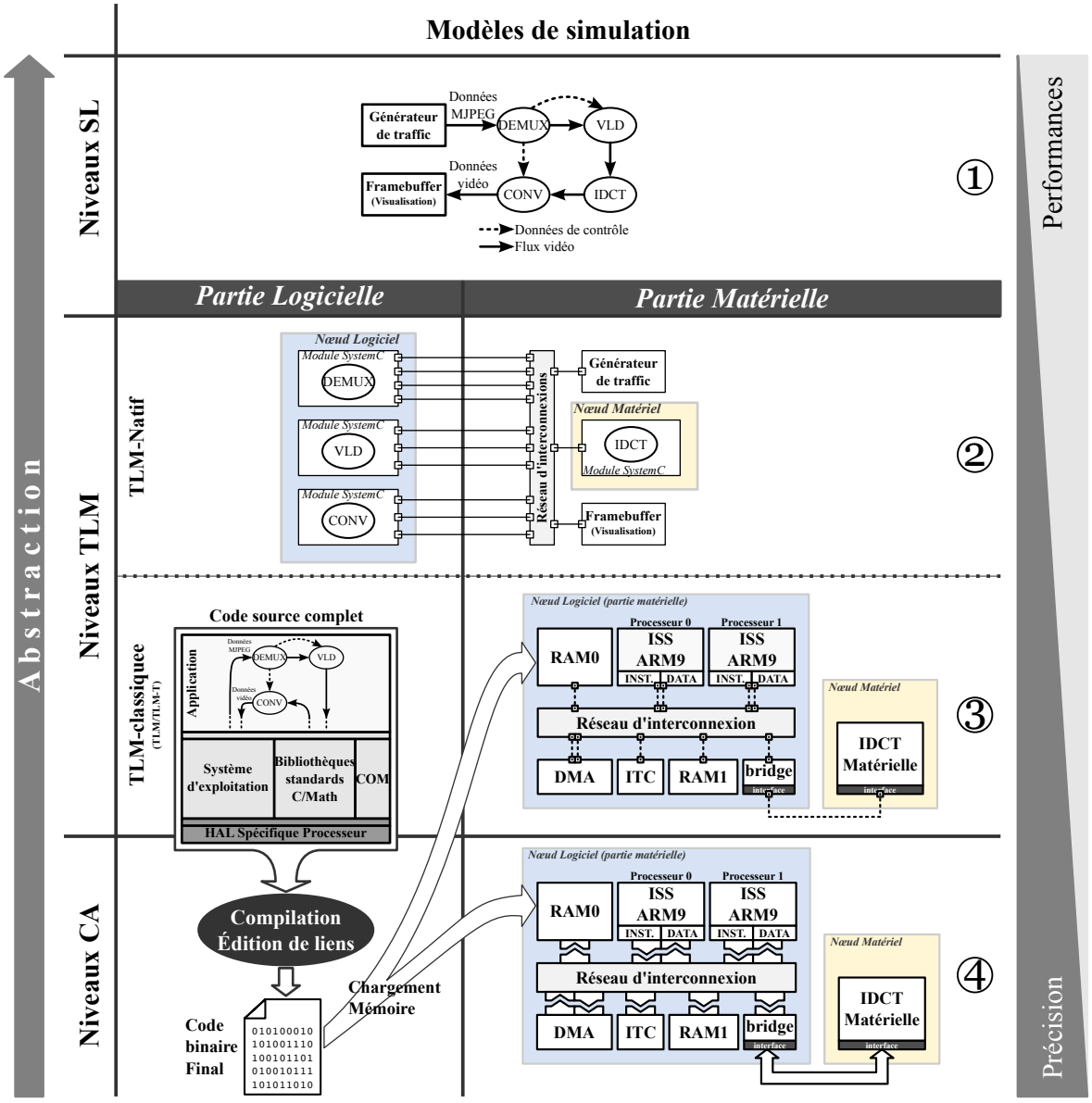


FIGURE 2.5 – Niveau d'abstraction et modèles de simulation

2.3.1 Le niveau système

Le premier niveau (le plus abstrait) ① sur la figure 2.5 est appelé SL pour (System Level) et correspond à une spécification exécutable de l'application. Le modèle de simulation correspondant peut être implémenté dans des langages tels que Simulink [PJ07, HHP+07], SystemC ou tout simplement en C/C++ [VKS00]. Il offre de très bonnes performances de simulation au prix d'une absence complète d'information temporelle, puisqu'il n'existe à ce niveau aucun détail de l'architecture sur laquelle l'application s'exécutera *in fine*. En particulier, on ne sait pas si une tâche est destinée à être réalisée en logiciel ou en matériel. Ce modèle permet néanmoins de valider certaines propriétés fonctionnelles de l'application et peut servir de modèle de référence.

2.3.2 Le niveau Cycle Accurate

Parmi les niveaux de simulation actuellement les mieux acceptés par les concepteurs de systèmes intégrés, on retrouve tout d'abord le niveau *Cycle Accurate* (CA), très précis par rapport à une plateforme réelle, au détriment des performances de simulation. Viennent ensuite les niveaux transactionnels (Transaction Level Modeling-TLM), dont l'objectif est d'offrir un compromis entre performance et précision de simulation.

L'objectif des modèles au niveau CA est de fournir une simulation relativement précise de l'ensemble du système. C'est le niveau le plus précis parmi ceux considérés dans ce travail (figure 2.5 ④). Il correspond à la dernière étape du flot de conception avant de passer au modèle RTL utilisé pour la synthèse du circuit final (non représenté ici). Pour ce faire, tous les composants matériels sont présents et implémentés de manière fidèle par rapport à leurs équivalents RTL, tout du moins au niveau de leurs interfaces où ils sont précis au bit et au cycle près. Les processeurs sont modélisés par des simulateurs de jeu d'instructions (ISS).

L'intégralité de la pile logicielle est compilée pour le processeur cible (nous utiliserons le terme *cross compilation* [3] par la suite), c'est à dire celui de l'architecture réelle. Son code binaire est chargé dans les mémoires simulées de la plateforme lors de l'initialisation du modèle, puis interprété et exécuté par les ISS durant la simulation. Ce principe est représenté sur la figure 2.6.

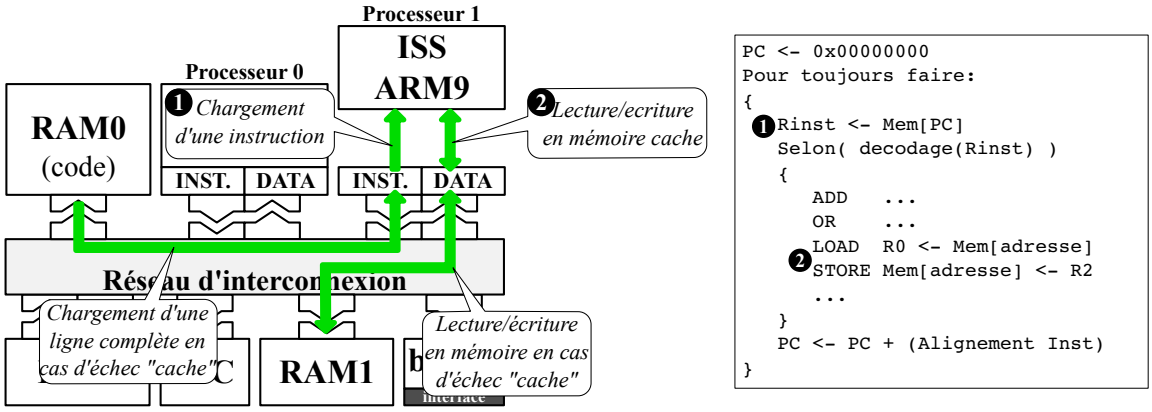


FIGURE 2.6 – Principe de fonctionnement d'un ISS dans son environnement matériel

3. La compilation croisée ou *cross compilation* définit le fait de compiler un programme pour un environnement matériel différent de celui sur lequel est effectuée la compilation

Une implémentation simplifiée consiste en une boucle infinie, dans laquelle on commence par charger le mot d'instruction pointé par le compteur programme (registre PC) dans le registre d'instruction du processeur simulé (❶ sur la figure). Une demande de lecture est faite sur le cache d'instruction (s'il y en a un) qui retournera directement la valeur du mot concerné s'il est présent, sinon, celui-ci ira le lire directement en mémoire. Le mot d'instruction est ensuite décodé puis exécuté en fonction de son type. S'il s'agit d'un accès en lecture ou écriture ❷ (`LOAD/STORE`), celui-ci sera également transféré vers le cache de données. Une fois l'instruction exécutée, le compteur programme est mis à jour et la même séquence est exécutée pour interpréter l'instruction suivante. Bien entendu, une réalisation d'ISS devra prendre en compte l'architecture interne du processeur (*pipeline*, cycles d'exécution des instructions, dépendances entre instructions, etc.) si l'on désire une précision absolue.

Tous les accès vers les mémoires (données et instructions) se font par l'interface avec le bus ou le réseau d'interconnexion en respectant précisément le protocole de communication. Les hiérarchies mémoire sont également modélisées de manière réaliste (notamment la gestion de cohérence entre les caches des systèmes multiprocesseurs), permettant de simuler finement le temps d'exécution du logiciel, en tenant compte de tous les aspects liés à l'architecture matérielle réelle.

Le prix à payer pour cette précision est une vitesse de simulation très faible, rendant ces modèles difficilement utilisables pour une validation complète de l'application ou du système d'exploitation. Ils restent cependant indispensables pour la validation fonctionnelle des composants matériels, tels que le comportement de leurs automates d'états finis, l'implémentation du logiciel bas niveau (vérification du programme de *bootstrap* et d'initialisation des périphériques par exemple) ou encore la validation des mécanismes de synchronisation dans les architectures multiprocesseurs nécessitant une précision temporelle fine.

2.3.3 Les niveaux transactionnels

La modélisation TLM permet d'augmenter les performances de simulation pour répondre aux besoins en terme de validation et d'exploration d'architecture. Elle met à profit l'utilisation des langages de programmation de logiciel telle que nous l'avons présentée précédemment afin d'abstraire les interfaces de communication des composants (généralement décrites au niveau signal) par des appels de fonction.

De précédents travaux de recherche [CG03, Don04, Ghe06, JBP06] ont défini plusieurs niveaux d'abstraction basés sur l'approche TLM, chacun ayant un rôle à jouer dans le raffinement d'un système du niveau SL au niveau CA. Dans la figure 2.5 nous n'avons pas cherché à représenter les différents modèles TLM en fonction de leurs niveaux d'abstraction mais plutôt en fonction de la technique utilisée pour traiter le code logiciel embarqué.

Il en ressort deux types d'approches, la première que nous pouvons qualifier de TLM « *classique* », où le logiciel reste interprété par un ISS comme sur les modèles au niveau CA et la seconde où le logiciel n'est plus interprété par le simulateur du processeur cible, mais directement intégré avec la description C/C++ du matériel, et donc exécuté par le processeur de la machine utilisée pour la simulation. On parle alors de simulation *native* et de modèle TLM *natif*.

2.3.3.1 Le niveau TLM classique

L'implémentation TLM *classique* est très proche du point de vue architecture d'un modèle au niveau CA. On peut voir cette similitude sur la figure 2.5, où seules les interfaces de communication changent entre les composants matériels du modèle de simulation au niveau CA ④ et ceux du niveau TLM ③. De la même manière, des ISS disposant d'interfaces de communication TLM sont utilisés pour modéliser les processeurs du système. Ainsi, le code binaire *cross compilé* peut directement être réutilisé sur la plateforme de simulation TLM.

La précision avec laquelle est décrite l'architecture du système et des composants ne se traduit pas directement par une précision temporelle de la simulation. En effet, les mécanismes de communication implémentés dans les interfaces des composants sont effectués par défaut sans prendre en compte le temps. On parle alors de modèles PV (pour Programmer's View). Afin d'obtenir une meilleure précision, des annotations temporelles sont ajoutées à la description des composants PV, donnant ainsi des modèles dits PVT (pour Programmer's View with Time) [CMMC08]. Ces annotations correspondent principalement à des latences, attribuées à chacune des interfaces de communication permettant ainsi d'introduire des délais dans les communications (typiquement les accès mémoires) et donc d'améliorer la précision temporelle des modèles de simulation.

Malgré l'amélioration des performances de simulation que procure cette abstraction, celle-ci peut s'avérer insuffisante pour permettre la validation des applications logicielles de plus en plus conséquentes, ou l'exploration d'architecture dans un espace de solutions de plus en plus grand. Les principaux facteurs limitant la vitesse de simulation des modèles TLM classiques sont d'une part liés au fait que toutes les communications dans le système (chargement d'instruction et accès aux données) sont modélisées et d'autre part liés à l'utilisation d'ISS pour l'interprétation et l'exécution des instructions du programme (qui reste un processus relativement lent).

2.3.3.2 Le niveau TLM natif

L'exécution native est une pratique très largement employée depuis longtemps par les concepteurs de logiciel embarqué. Elle consiste à compiler et à exécuter certaines parties d'un programme non pas pour le processeur embarqué (ou processeur cible) mais pour le processeur de la machine utilisée lors du développement ou de la simulation (le processeur hôte). Cette technique permet de valider le comportement fonctionnel du code compilé nativement sans attendre la disponibilité d'un premier prototype matériel.

La quantité de code pouvant être compilé nativement dépend fortement de sa dépendance au matériel. Ainsi, on peut idéalement exécuter la totalité d'une application reposant sur le standard POSIX aussi bien sur une machine hôte équipée d'un système Linux que sur un matériel final disposant du même système d'exploitation embarqué. Bien entendu, plus il y aura de logiciel dépendant du matériel, et moins l'on pourra exécuter de code nativement. Le plus souvent, elle n'est utilisée que pour valider le comportement fonctionnel d'un algorithme complexe ou du corps des tâches logicielles.

Ce concept simple peut très facilement être adapté à une modélisation SystemC. Nous illustrons ici ce principe. Sur la figure 2.5 ②, les tâches logicielles de l'application MJPEG sont encapsulées dans des modules SystemC. La partie logicielle ne contient pas de système d'exploitation, ni même de bibliothèque de communication et la partie matérielle ne modélise généralement que l'interconnexion entre les modules matériels et logiciels.

Le listing 2.1 donne le code de la tâche CONV[4] de l'application MJPEG (représentée figure 2.1) dans un module SystemC (lignes 1 à 4). Le corps de la tâche de conversion est découpé en trois phases :

1. Lecture des entrées (lignes 10 à 12)
2. Traitement des données (lignes 14 à 22)
3. Écriture des sorties (ligne 23)

Listing 2.1 Intégration de la tâche CONV dans un module SystemC

```
1  CONV::CONV(sc_module_name name) : sc_module(name)  /* Encapsulation      */
2  {                                                  /*     SystemC        */
3    SC_THREAD(threadConv);                           /* Instantiation      */
4  }                                                  /*     du thread      */
5
6  void CONV::threadConv(Channel *c[2]) {
7    uint8  MCU_Y[64], MCU_Cr[64], MCU_Cb[64], MCU_RGB[64*3];
8    int    R,G,B;
9    while(1)  {
10     input.read(MCU_Y,sizeof(MCU_Y));               /* <-| Lecture        */
11     input.read(MCU_Cb,sizeof(MCU_Cb));             /*   | Des            */
12     input.read(MCU_Cr,sizeof(MCU_Cr));             /* <-| Entrees        */
13
14     for(uint32_t i = 0 ; i < 8 ; i++) {            /* <-| Traitement     */
15       R = 1.402*(MCU_Cr[i]-128)+MCU_Y[i];          /*   | Conversion     */
16       G = MCU_Y[i]-0.34414*(MCU_Cb[i]-128)         /*   |  3 canaux      */
17           -0.71414*(MCU_Cr[i]-128);                /*   |    YUV         */
18       B = 1.7772*(MCU_Cb[i]-128)+MCU_Y[i];         /*   |                */
19       MCU_RGB[i*3] = R;                            /*   |    vers        */
20       MCU_RGB[i*3+1] = G;                          /*   |  1 canal       */
21       MCU_RGB[i*3+2] = B;                          /*   |    RGB         */
22     }                                              /* <-|                */
23     output.write(MCU_RGB, sizeof(MCU_RGB));        /* <-| Sorties        */
24   }
25 }
```

La figure 2.7 représente la vue structurelle de cette tâche logicielle dans SystemC. Le code source de la tâche est encapsulé dans un processus SystemC (SC_THREAD) et les communications se font au travers des interfaces input et output (sc_interface SystemC) directement par des appels de fonctions (lignes 10, 11,12 et 23 du code source 2.1), implémentées dans les canaux de communication TLM. Le comportement de la bibliothèque de communication qui serait utilisée dans une version finale du logiciel est donc modélisée par les canaux de communication eux mêmes.

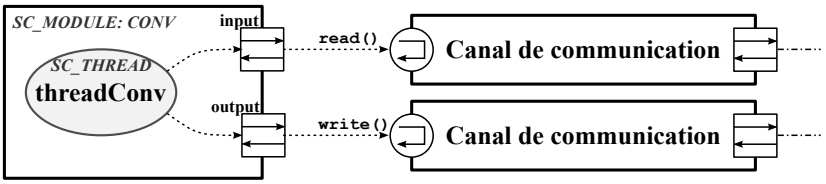


FIGURE 2.7 – Représentation SystemC du module logiciel de conversion

Les modules SystemC dans lesquels sont encapsulées les tâches logicielles sont gérés de la même manière que les modules matériels du système. SystemC ne fait aucune considération quant au type des modules (logiciel ou matériel) qui ne sont à ce niveau que

4. tâche permettant de convertir les composantes de couleur YUV au format RGB

des « *étiquettes* » données par les concepteurs. Ainsi toutes les tâches du système s'exécutent en un parallélisme simulé. Cela revient à modéliser le comportement d'un système d'exploitation directement avec le noyau de simulation, même si le degré de concurrence est complètement différent et que la manière d'exécuter une tâche logicielle et une tâche matérielle n'ont en réalité rien à voir.

Ce niveau de modélisation est utilisé pour la validation fonctionnelle du système et permet d'atteindre des vitesses de simulation très élevées. Le fait de ne pas disposer de tout le logiciel ni même d'un modèle précis de l'architecture matérielle ne permet bien évidement pas d'obtenir d'informations temporelles. La modélisation de certaines caractéristiques de l'architecture matérielle (types de ressources utilisées pour les communications, protocoles, etc.) dans les canaux TLM permet cependant d'extraire des informations statistiques sur les communications utiles dans le processus d'exploration.

Les approches de simulation natives sont donc moins précises que les approches TLM *classiques* utilisant des modèles PV. Même lorsqu'elles ne sont pas précises temporellement, les plateformes TLM *classiques* le sont au niveau des transactions de communication.

2.4 Le modèle de simulation Transaction Accurate

Le modèle de simulation *Transaction Accurate* (que nous appellerons TA dans la suite) proposé dans [BBY+05] est dérivé des approches TLM natives. Il est basé sur l'organisation en couche de la pile logicielle et permet une validation plus complète de celle-ci.

2.4.1 La couche d'abstraction du matériel

L'exécution native du logiciel n'est possible que s'il existe une interface (API [5]) clairement définie entre la partie logicielle à exécuter et le modèle de la plateforme matérielle sous-jacente. Plus simplement, le code logiciel ne doit contenir aucune dépendance au matériel (processeur + plateforme) sur lequel il est censé s'exécuter si l'on veut pouvoir le compiler pour le processeur hôte. C'est le cas dans l'exemple donné précédemment pour la simulation TLM native où cette interface est définie par l'API de communication.

Partant de cette considération, le niveau d'abstraction le plus bas qu'il est possible d'implémenter pour la simulation native doit fournir l'API pour la couche logicielle d'abstraction du matériel (HAL pour Hardware Abstraction Layer).

Définition 2.1 Hardware Abstraction Layer

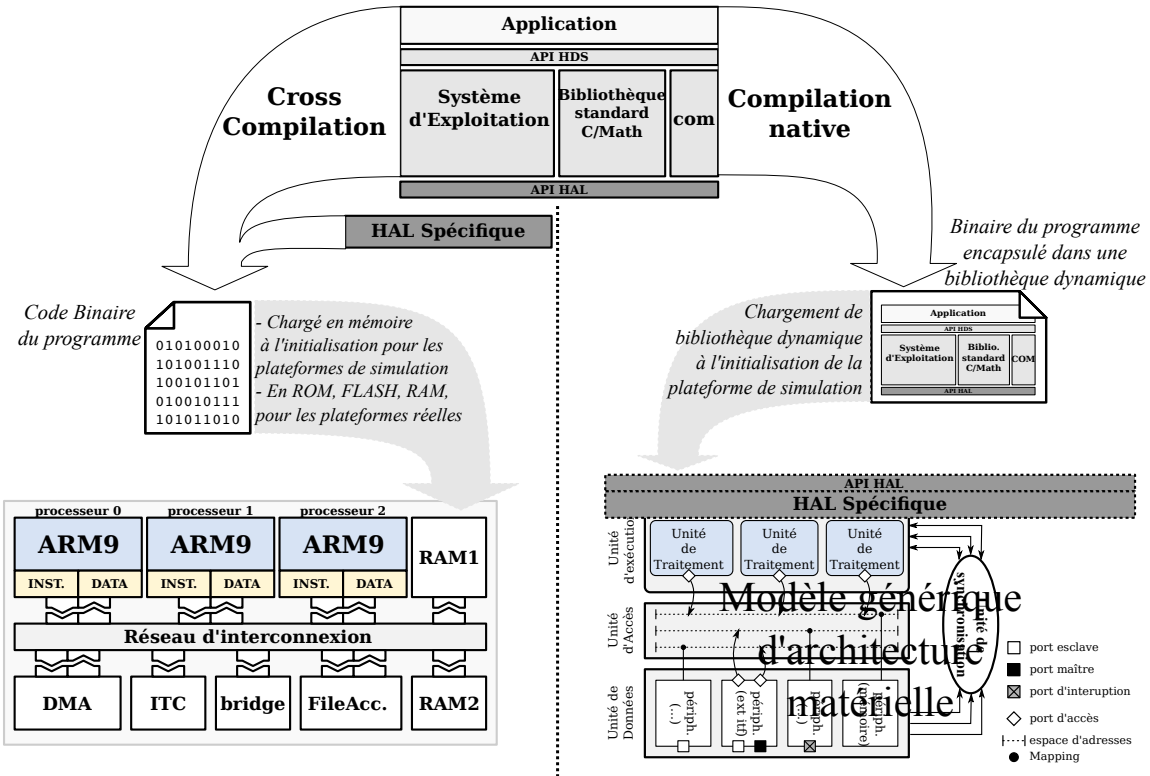
Comme son nom l'indique, le HAL est une couche d'abstraction, réalisée en logiciel, et permettant de masquer les spécificités d'une architecture matérielle aux couches logicielles de plus haut niveau s'exécutant sur celle-ci. Il existe cependant toutes sortes de HAL, chaque système d'exploitation donnant sa propre définition et version de l'API de plus ou moins haut niveau.

Le concept de HAL dans le domaine des SoC, introduit dans [YJ03], offre les fonctionnalités de démarrage, de commutation de contexte des processeurs, de configuration des mémoires caches et virtuelles et tous les mécanismes « fin » de gestion du matériel.

L'utilisation de ce HAL est particulièrement importante pour garantir la portabilité du logiciel (généralement des systèmes d'exploitation) entre différentes architectures matérielles. Cette portabilité reste également valable dans le cas de la simulation native.

5. Une interface de programmation (*Application Programming Interface*) est un ensemble de primitives (fonctions, classes, etc.) destinées à isoler différentes parties du logiciel en cachant leurs implémentations.

Idéalement, tout le code logiciel situé au dessus de la couche HAL peut être aussi bien *cross compilé* pour le processeur cible (figure 2.8(a)) que compilé nativement pour le processeur de la machine hôte (figure 2.8(b)).



a) Plateforme exécutant le programme cible (processeur réelle ou simulée avec des ISS au niveau CA ou TLM)

b) Plateforme TLM native, exécutant le programme compilé pour le processeur hôte

FIGURE 2.8 – Plateforme de simulation native d'un nœud logiciel

Dans le cas d'un portage pour l'architecture matérielle cible (partie gauche de la figure 2.8), la couche HAL spécifique au processeur embarqué est compilée avec l'application pour obtenir le code binaire final. Ce binaire peut ensuite être exécuté directement sur toute les plateformes capables d'interpréter le jeu d'instructions du processeur cible. Cela peut aller d'une plateforme matérielle réelle (prototype ou SoC final) à une plateforme de simulation du matériel (en VHDL, au niveau CA ou TLM à base d'ISS).

En ce qui concerne l'exécution native du logiciel (partie droite de la figure 2.8), le problème est légèrement différent : *Il s'agit là non pas d'implémenter directement un HAL pour le processeur hôte mais pour une plateforme de simulation s'exécutant sur celui-ci.*

Cela sous-entend que le modèle de simulation supportant l'exécution native doit être capable d'intercepter chacun des appels aux fonctions de l'API du HAL, afin de les rediriger vers les composants matériels concernés de la plateforme. Une solution simple consiste alors à définir chaque primitive de l'API comme une fonction *externe* qui sera implémentée indépendamment dans le modèle de la plateforme matérielle. Toutes les fonctions de l'API du HAL étant déclarées ainsi, l'ensemble du logiciel situé au dessus de cette interface est compilé en utilisant les outils spécifiques au processeur de la machine hôte. Le résultat de cette compilation est une bibliothèque dynamique, dans laquelle seuls les appels vers les fonctions HAL sont indéfinis.

Les listings 2.2 et 2.3 correspondent respectivement à l'implémentation de la fonction

`CPU_MP_ID()` permettant de connaître l'identifiant du processeur courant pour un ARM et pour une exécution native. L'implémentation pour le processeur ARM consiste à écrire le code assembleur permettant d'accéder à un registre de contrôle spécifique contenant cet identifiant (ligne 7 du listing 2.2).

Listing 2.2 Fonction HAL `CPU_MP_ID()` pour le processeur ARM966

```
1  #ifndef _ARM9ES_CPU_MP_H_
2  #define _ARM9ES_CPU_MP_H_
3
4  static inline unsigned long int CPU_MP_ID (void) {
5    register unsigned long int id;
6
7    __asm__ volatile ("mrc p15, 0, %0, c0, c0":"=r"(id));
8    return (id&0xF);
9  }
10
11 #endif
```

Listing 2.3 Fonction HAL `CPU_MP_ID()` pour la plateforme native

```
1  #ifndef _NATIVE_CPU_MP_H_
2  #define _NATIVE_CPU_MP_H_
3
4  extern unsigned long int CPU_MP_ID(void);
5
6  #endif
```

Dans le cas de la plateforme native, cette fonction n'est pas implémentée directement dans la couche HAL de la partie logicielle, mais elle est simplement déclarée comme étant une fonction `extern` (ligne 4 du listing 2.3), et sera réellement implémentée dans le modèle générique de l'architecture matérielle (figure 2.8(b)). La bibliothèque contenant le logiciel est chargée à l'initialisation de la simulation grâce au chargeur fourni par le système d'exploitation disponible sur la machine hôte. Il effectuera lui-même l'édition de liens entre les appels aux fonctions HAL et leurs implémentations dans la plateforme de simulation.

Au cours de l'exécution du logiciel, les appels aux fonctions du HAL seront donc redirigés vers leurs implémentations dans la plateforme de simulation, qui dispose alors de toutes les libertés possibles pour modéliser leurs comportements. Par exemple, la gestion des contextes d'exécution (indispensable pour le support d'application multi-tâches) pourra être réalisée sur un système Linux à l'aide des fonctions POSIX `makecontext()` et `swapcontext()`.

2.4.2 Exécution du logiciel natif

La contenu du modèle générique au niveau TA de la figure 2.8(b) est détaillé sur la figure 2.9, tiré de [BBY+05]. Il est important de noter que les différents éléments qui composent ce modèle ne correspondent pas à des composants matériels physiques, mais plutôt à des fonctionnalités logicielles telles que les voient les programmeurs du système. De plus, les flèches sur ce modèle ne représentent pas des signaux ou des bus, mais des relations logiques entre ces différents éléments fonctionnels.

L'exécution de la partie logicielle encapsulée dans la bibliothèque dynamique est prise en charge par une ou plusieurs unités de traitements, formant un composant (module SystemC) appelé *Unité d'Exécution* (EU pour Execution Unit). L'unité d'accès joue un rôle important dans le transfert des données entre les unités de traitements et les unités de

données, modélisant les périphériques du système. Les mécanismes d'interruptions du système sont modélisés par l'unité de synchronisation.

Le principe d'exécution du logiciel est représenté sur la figure 2.9. Le contexte d'exécution fourni par un processeur est modélisé par un processus SystemC (`SC_THREAD`) unique ① fourni par une unité de traitement et s'exécutant du point de vue de la simulation de manière concurrente avec les autres composants de la plateforme (y compris les autres unités de traitement).

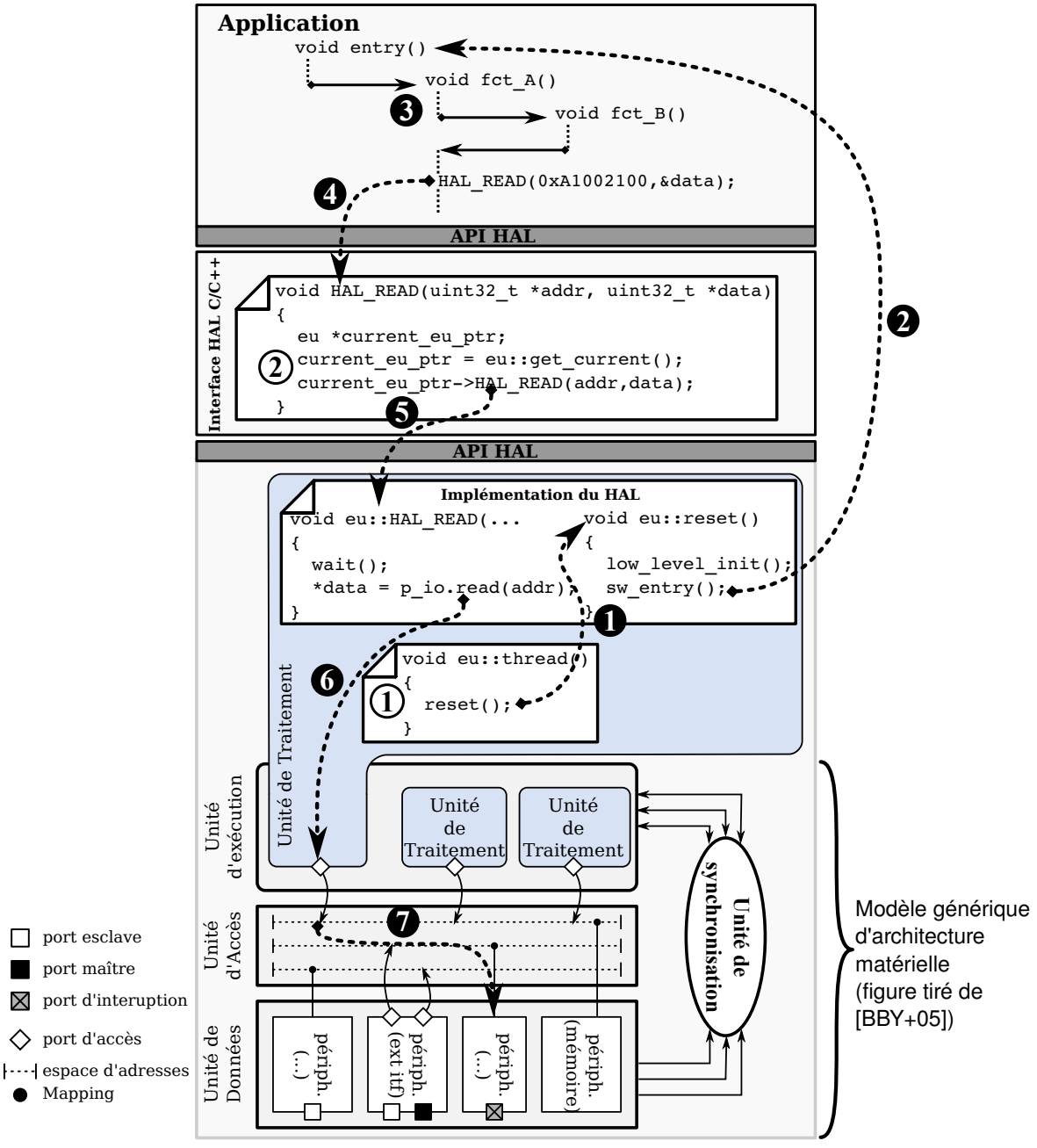


FIGURE 2.9 – Principe de l'exécution native sur une plateforme Transaction Accurate

L'exécution de logiciel sur une unité de traitement est très similaire à une exécution sur un processeur et se déroule de la manière suivante :

❶ Lorsqu'un processeur démarre, celui-ci fait automatiquement un appel à l'exception de `RESET` qui permettra de faire les initialisations « bas niveau » avant d'appeler le point d'entrée de l'application. Ce démarrage du processeur est modélisé par l'appel à la méthode C++ `eu::thread()` lorsque la simulation débute. Le branchement vers l'exception de `RESET` est implémenté par un simple appel vers la fonction du même nom implémentée dans la couche HAL de l'unité de traitement.

❷ Une fois les initialisations « bas niveau » réalisées (piles, interruptions, etc.), un appel à la fonction d'*entrée* de l'application logicielle permet d'amorcer son démarrage. Elle correspond typiquement au point d'entrée du système d'exploitation (contenu dans la bibliothèque dynamique).

❸ Le code de la partie logicielle étant indépendant de la plateforme de simulation, celui-ci s'exécute alors de manière séquentielle comme il le ferait sur un processeur réel. Cette exécution s'effectue cependant en temps simulé nul du point de vue de SystemC et de manière illimitée.

❹ Le noyau de simulation perd ainsi tout contrôle jusqu'à ce qu'un appel à une fonction HAL soit fait. Cet appel n'est pas directement intercepté par l'unité de traitement sur laquelle s'exécute le logiciel pour plusieurs raisons :
- Premièrement, le logiciel n'ayant pas « conscience » du support sur lequel il s'exécute (que ce soit un processeur ou une unité de traitement), il est nécessaire de mettre en place un mécanisme permettant de retrouver quelle est l'unité concernée, pour ensuite rediriger l'appel vers la bonne instance de module SystemC ②.
- Deuxièmement, une adaptation peut-être nécessaire lorsque la plateforme et le logiciel ne sont pas implémentés dans un même langage. Typiquement, si l'application est implémentée en langage C et la plateforme en langage C++.

❺ La couche d'adaptation HAL permet de rediriger l'appel ❹ vers l'unité de traitement où est implémentée la fonction HAL. Cet appel permet indirectement à SystemC de reprendre le contrôle de la simulation grâce à la primitive `wait()`. Cette fonction permet de suspendre l'exécution du processus qui l'appelle jusqu'à un prochain évènement ou pour une durée déterminée. Le noyau de simulation peut ainsi exécuter les autres processus concurrents du système et ainsi faire évoluer le temps simulé.

❻ Dans cet exemple, l'appel à la fonction HAL de lecture d'un registre de périphérique est redirigé vers l'interface de communication du processeur, comme le ferait un simulateur de processeur lors de l'exécution d'une instruction de type `LOAD`.

❼ Cette transaction, au sens TLM du terme, est alors aiguillée par l'unité d'accès vers le bon périphérique en fonction de l'adresse demandée. Une fois le transfert terminé, le retour dans l'application logicielle se fait naturellement. Celle-ci reprend alors son exécution séquentielle jusqu'à un nouvel appel à une fonction du HAL.

Cette solution présente l'intérêt de pouvoir exécuter et donc valider l'ensemble du logiciel reposant sur la couche HAL, qu'il s'agisse d'un système d'exploitation ou non. L'aptitude à pouvoir modéliser des architectures multiprocesseurs est indépendante de cette approche. Elle dépend uniquement de la capacité du modèle matériel à supporter les mécanismes nécessaires pour ce type de parallélisme. Typiquement, si la plateforme fournit les mécanismes de synchronisation entre processeurs et permet l'identification de ceux-ci, il sera alors possible de supporter l'exécution de systèmes d'exploitation de type SMP.

2.5 Problématiques liées à l'exécution native du logiciel

2.5.1 Temps d'exécution du logiciel

Un inconvénient majeur de la simulation native est lié au fait que l'exécution du logiciel dans un module SystemC se fait en temps simulé nul du point de vue des composants matériels du système. La figure 2.10 montre comment le temps de simulation évolue dans l'exemple de la tâche logiciel `CONV` de l'application MJPEG donné précédemment (listing 2.1). Une implémentation des canaux de communication avec des annotations temporelles permet de faire évoluer le temps de simulation lors des appels aux fonctions de lecture ou d'écriture des données (représenté par un trait noir épais sur le figure 2.10). Cependant, le temps de simulation consommé par l'exécution de l'algorithme de conversion de données (lignes 14 à 22 du code source 2.1) est nul.

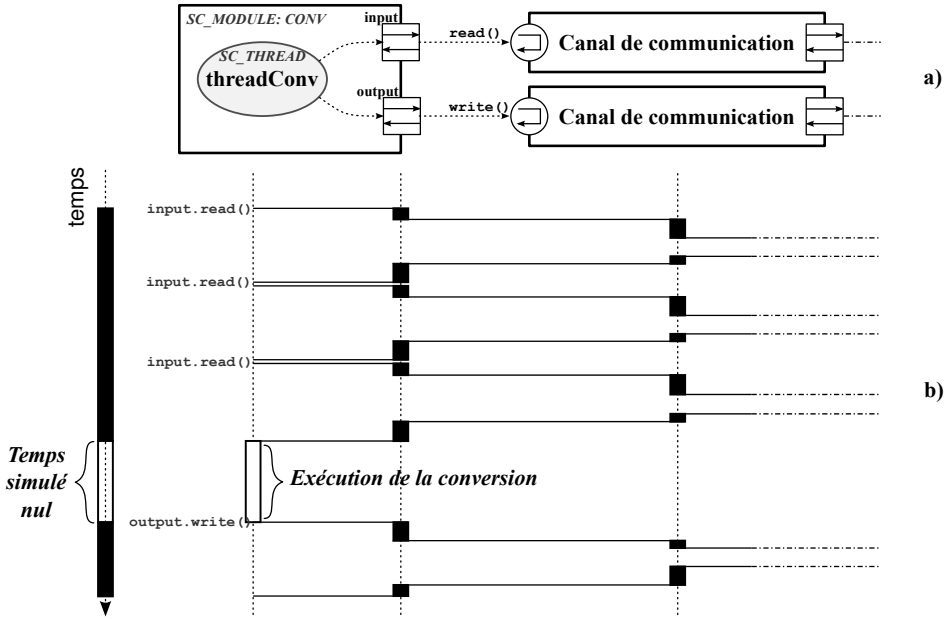


FIGURE 2.10 – Exécution temporelle de la tâche `CONV`

La seule solution consiste à insérer des annotations temporelles dans le code de l'application afin de faire appel à la fonction `wait()` de SystemC permettant de faire évoluer le temps de simulation. Le listing 2.4 correspond au code source annoté de la tâche de conversion encapsulée dans le module SystemC (lignes 3, 7 et 9).

Il est très important de bien être conscient que, d'une manière générale et par défaut, la plateforme de simulation (et donc SystemC) n'a aucun contrôle sur l'exécution native du logiciel. Ainsi en plus du problème évoqué ci dessus concernant le temps d'exécution nul du point de vue de la simulation, tous les accès à la mémoire effectués dans le code logiciel se font à l'insu de la plateforme de simulation. Cela concerne aussi bien les accès explicites, implémentés par le développeur logiciel (traitement d'un tableau de données par exemple), que les accès implicites, tels que ceux effectués pour charger les instructions du programme lui-même lors de son exécution ou encore ceux liés à l'utilisation de piles en mémoire pour la gestion des paramètres et des variables locales aux fonctions d'un programme.

Listing 2.4 Tâche `CONV` annotée temporellement dans SystemC

```
1  void CONV::threadConv(Channel *c[2]) {
2    ...
3    wait(2,SC_NS);                              /* <- Annotation 1 */
4    while(1)  {                                 /* SC_NS = nanosecondes */
5      input.read(MCU_Y,sizeof(MCU_Y));
6      ...
7      wait(2,SC_NS);                            /* <- Annotation 2 */
8      for(uint32_t i = 0 ; i < 8 ; i++) {
9        wait(64,SC_NS);                         /* <- Annotation 3 */
10       R = 1.402*(MCU_Cr[i]-128)+MCU_Y[i];
11       ...
12     }
13     output.write(MCU_RGB, sizeof(MCU_RGB));
14   }
15 }
```

2.5.2 Représentation de la mémoire

Un autre problème important dans cette approche est lié à la compilation native elle même. Il existe deux sources de dépendance du logiciel au matériel. La première est liée directement au matériel lui-même et correspond typiquement au jeu d'instructions du processeur ou encore aux spécificités de certains composants. Cette dépendance est résolue simplement par l'utilisation de la couche HAL comme interface entre les parties logicielles et matérielles telle qu'elle a été présentée dans le début de ce chapitre. La deuxième dépendance est liée à la représentation mémoire partagée entre le matériel et le logiciel.

Pour mieux comprendre en quoi cette dépendance pose problème dans les approches basées sur la simulation native, il est nécessaire de mettre en évidence les différentes représentations mémoire de la plateforme.

Les plans mémoire dans la plateforme native sont au nombre de deux et sont représentés sur la figure 2.11 :

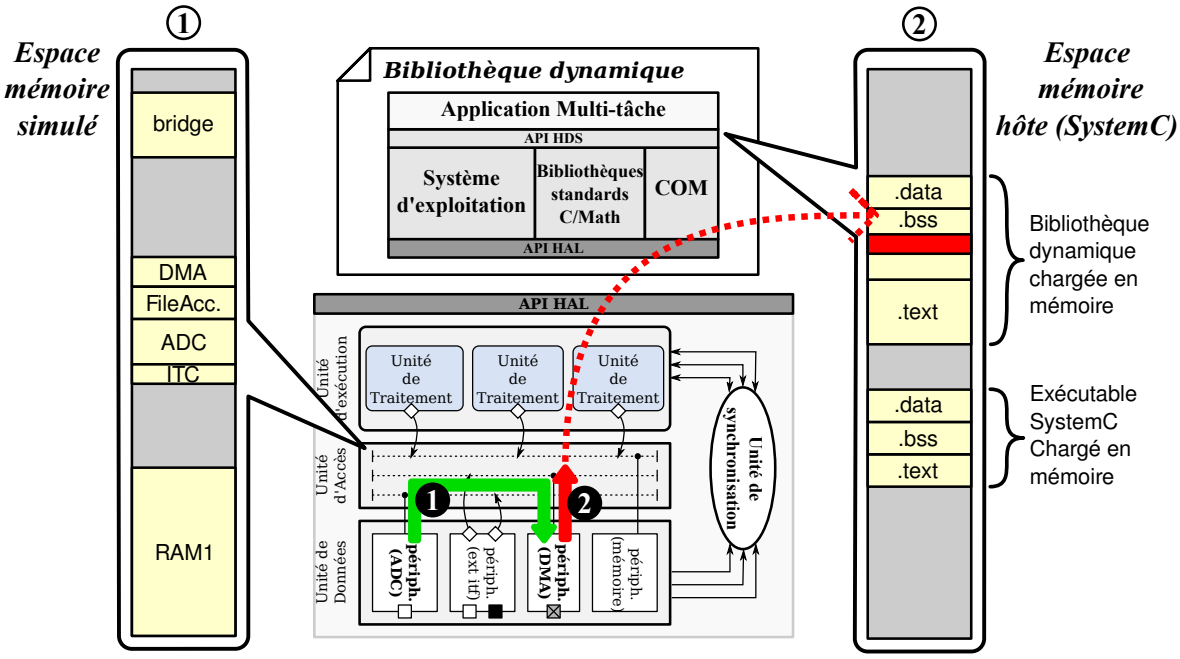


FIGURE 2.11 – Différentes vues mémoire sur une plateforme native

① Le plan mémoire vu par la partie matérielle de la plateforme. Celui-ci est défini par les concepteurs de l'architecture du MPSoC. Il est simulé par le décodeur d'adresses du mécanisme de communication (bus, réseau, etc.) de la plateforme.

② Le plan mémoire vu par le logiciel compilé nativement. Il correspond au plan mémoire du contexte dans lequel le logiciel natif s'exécute réellement, c'est-à-dire celui du programme exécutable du noyau de simulation SystemC. Ce plan mémoire est défini par le système de la machine sur laquelle est exécutée la simulation.

Cette hétérogénéité des espaces mémoire se traduit par une incapacité à modéliser certaines interactions pourtant essentielles entre le logiciel et le matériel, et peut être mise en évidence par un exemple de transfert DMA.

Soit une tâche logicielle (listing 2.5) dont le rôle est de traiter des données provenant d'un convertisseur analogique/numérique (ADC pour Analog to Digital Converter). Supposons que l'on veuille utiliser un transfert DMA pour récupérer les données se trouvant dans un registre de l'ADC et les copier dans un tableau alloué en mémoire.

L'adresse *source* du transfert se trouve dans le plan mémoire simulé ① et est définie « *en dur* » à la valeur `0xA0001004` (ligne 1). L'adresse de *destination* se trouve quant à elle dans le plan mémoire de l'exécutable SystemC (dans l'espace mémoire ②) puisqu'elle a été obtenue lors de la résolution dynamique de liens au moment du chargement de l'application. Cette adresse est placée plus précisément dans la zone de mémoire de données réservée pour les variables non initialisées du processus Unix (section *.bss*) tel que c'est le cas pour la déclaration du tableau `data_in` ligne 3, et ne peut être connue qu'au moment de l'exécution (nous prendrons arbitrairement la valeur `0xBF1ACE00`).

Listing 2.5 Tâche de traitement de données provenant d'un convertisseur Analogique/Numérique par transfert DMA

```
1  #define ADC_RCV_REG 0xA0001004;
2
3  static int data_in  [1024];
4  static int data_out [1024];
5
6  void thread_compute() {
7    /* variables locales ... */
8    while(1) {
9      DMA_TRANSFERT(ADC_RCV_REG, data_in,1,4, 1024);
10     for( .... )
11     {
12       data_out[i] = fct(data_in[i]);
13     }
14   }
15 }
```

Ces adresses sont utilisées lors de la configuration des registres du DMA effectuée dans la fonction `DMA_TRANSFER` (listing 2.6). Avant de démarrer le transfert par l'écriture du nombre de données à copier dans le registre de contrôle (ligne 14), les registres du DMA contiennent les valeurs suivantes :

```
DMA_SOURCE   =  0xA0001004  (ligne 10)
DMA_DESTIN   =  0xBF1ACE00  (ligne 11)
DMA_INC_SRC  =  0x00000000  (ligne 12)
DMA_INC_DST  =  0x00000004  (ligne 13)
```

Le programme peut ensuite continuer son exécution pendant que le DMA se charge de faire les transferts mémoires.

Listing 2.6 Fonction d'initialisation d'un transfert DMA

```
1  #define  DMA_BASE     0xB0000000
2  #define  DMA_CONTROL  DMA_BASE
3  #define  DMA_SOURCE   DMA_BASE + 0x04
4  #define  DMA_DESTIN   DMA_BASE + 0x08
5  #define  DMA_INC_SRC  DMA_BASE + 0x0C
6  #define  DMA_INC_DST  DMA_BASE + 0x10
7
8  void DMA_TRANSFER(void *src, void *dest, int inc_src, int inc_dest, int nb)
9  {
10   HAL_WRITE_32(DMA_SOURCE, src);       /* Adresse source */
11   HAL_WRITE_32(DMA_DESTIN, dst);       /* Adresse destination */
12   HAL_WRITE_32(DMA_INC_SRC, inc_src);  /* Increment d'adresse source */
13   HAL_WRITE_32(DMA_INC_DST, inc_dst);  /* Increment d'adresse destination */
14   HAL_WRITE_32(DMA_CONTROL, nb);       /* Debut transfert */
15 }
```

Si l'on considère maintenant les deux plans mémoire de la plateforme de simulation native, représentés sur la figure 2.11. Les opérations de transferts (représentées sur la figure 2.11) se déroulent de la manière suivante :

❶ Les données sont lues à l'adresse configurée dans le registre `DMA_SOURCE`. Dans cet exemple, l'adresse utilisée est valide dans l'espace mémoire ①. Le réseau d'interconnexion le redirige donc vers le bon périphérique (le composant ADC) et l'accès à la donnée peut se faire normalement.

❷ Une fois récupérée, cette donnée doit être écrite à l'adresse de destination configurée dans le registre `DMA_DESTIN`. Cette adresse n'est pas valide pour le décodeur d'adresse du réseau d'interconnexion puisqu'elle est définie dans l'espace mémoire ②.

Sans modification du code logiciel ou des mécanismes de communication de la plateforme TA, ce type d'architecture n'est pas supportée directement. Rappelons que le plupart des architectures réelles disposent d'au moins un DMA, ou d'un composant maître connecté sur le réseau de communication, autre qu'un processeur. Cet exemple de transfert DMA illustre bien les limitations de la simulation native sur une plateforme TA dès lors qu'un composant matériel souhaite accéder à l'espace mémoire vu par le logiciel s'exécutant nativement.

Il existe également le problème réciproque dans lequel c'est le logiciel qui souhaite accéder directement à des zones mémoires modélisées dans la partie matérielle décrite au niveau TA. Prenons l'exemple d'un RAMDAC[6] dont la mémoire serait directement accessible à l'adresse `0x000A0000`.

Dans le listing 2.7, un pointeur C est initialisé directement à cette valeur (ligne 6) et est directement utilisé pour accéder à la mémoire (ligne 9). Cet accès ne pose pas de problème sur une plateforme réelle ou CA (sous réserve que le RAMDAC se trouve effectivement à cette adresse) mais il engendrera une erreur de segmentation en simulation native puisque cette zone de mémoire n'est pas accessible par l'application. Ce type d'accès direct à des zones mémoires n'est donc pas possible tel quel. Les accès par pointeur doivent soit être interdits, soit être faits par l'intermédiaire d'une API dédiée à la simulation native.

6. Un RAMDAC est un convertisseur numérique/analogique, c'est-à-dire un composant électronique, chargé de convertir l'image numérique stockée dans la mémoire vidéo en signal analogique destiné au moniteur.

Listing 2.7 Accès directe en mémoire par pointeur C

```
1  #define RAMDAC_BASE   0x000A0000
2
3  void init_color()
4  {
5    volatile unsigned int *ramdac_ptr;
6    ramdac_ptr = (volatile unsigned int *)RAMDAC_BASE;
7
8    for( ...) {
9      *ramdac_ptr = 0x00FFFFFF;
10       ramdac_ptr++;
11   }
12 }
```

2.6 Conclusion : besoins et objectifs

Il sera nécessaire de disposer dans un futur proche de modèles de simulation très performants tout en gardant un niveau de précision suffisant pour permettre la validation et l'exploration d'architectures. Les solutions actuelles basées sur l'exécution native du logiciel et la modélisation TA du matériel donnent de très bonnes performances de simulation et semblent très prometteuses quant à la quantité de code pouvant être validé, malgré les problèmes exposés précédemment.

Nous souhaitons donc approfondir ce type d'approche en proposant une plateforme de simulation native ayant pour objectifs d'offrir de bonnes performances de simulation, tout en conservant un niveau de précision acceptable.

Sur le tableau 2.1, la première ligne donne un récapitulatif du classement des différents niveaux d'abstraction du matériel en fonction de leurs compromis entre vitesse de simulation et précision de modélisation. La seconde ligne présente quant à elle la quantité de code logiciel pouvant être validé sur chacun des modèles de simulation.

	Précision → *Performances*			
Niveau de modélisation du matérielle	CA	***TLM-classique***	TLM-natif	SL
Validation de partie la logicielle	Pile logicielle complète	Pile logicielle complète	***Pile logicielle indépendante du matériel***	Tâches applicatives

Table 2.1 – Compromis précision/performances de simulation

Du point de vue matériel, le niveau TLM *classique* semble être un bon candidat en raison de son bon compromis entre performances de simulation et précision du modèle matériel simulé, bien qu'il pâtisse de l'utilisation d'ISS pour la validation du code logiciel.

La précision de la simulation est également fortement liée à la quantité de code logiciel validé. En ce qui concerne le logiciel, la simulation native sur un modèle Transaction Accurate telle que nous l'avons présentée précédemment semble donc être la solution idéale, puisqu'elle allie vitesse de simulation et quantité de code validé.

Une des premières question qui peut alors être posée et qui dresse l'objectif global de cette thèse est la suivante :

Est-il possible d'insérer dans le tableau 2.1 une colonne intermédiaire, où le matériel serait modélisé au niveau TLM classique*, et où le logiciel ne serait pas interprété par des ISS mais exécuté nativement sur le processeur de la machine hôte ?*

Étant donnés les problèmes liés à l'exécution native évoqués dans ce chapitre, il est pertinent de se poser les questions suivantes :

- Comment exécuter une pile logicielle complète de manière native sur une plateforme de simulation TLM ?
- Comment faire en sorte de maximiser la quantité de code source final réutilisé tout en minimisant les contraintes de codage ?
- Dans quelle mesure est-il possible de prendre en compte les détails architecturaux au niveau du logiciel natif, tels que l'exécution parallèle du logiciel, les représentations mémoire ainsi que les évènements matériels ?

L'intérêt d'une telle plateforme de simulation reste cependant limité à la simulation fonctionnelle en raison de l'absence totale de considération sur le code logiciel exécuté. L'exploration d'architecture requiert des plateformes de simulation qu'elles apportent un minimum d'informations, indispensables dans les phases de conception et notamment lors des choix architecturaux. Ces informations, qu'elles soient temporelles, énergétiques ou toute autre grandeur physique, nécessitent l'instrumentation du code logiciel. Étant donné la quantité de code que peut représenter une pile logiciel, seule une instrumentation entièrement automatique est envisageable.

La deuxième partie de cette thèse sera donc consacrée à apporter des réponses aux questions soulevées par l'instrumentation du code logiciel :

- A quel niveau instrumenter le code logiciel (source, binaire, ...) ?
- Comment refléter l'exécution du logiciel sur le processeur cible ?
- Comment prendre en compte ces informations dans une plateforme de simulation native ?

Chapitre 3

Exécution native du logiciel et estimation de performances

Sommaire

L'exécution native est une méthode très largement employée pour la validation fonctionnelle dans la plupart des domaines d'application du logiciel embarqué. Dans les flots de conception de MPSoC, cette solution a été utilisée principalement pour accélérer les vitesses de simulation et anticiper la validation du logiciel. L'inconvénient majeur des approches basées sur cette méthode de simulation est lié à une modélisation très haut niveau du matériel de la plateforme de simulation, voire même l'absence totale de celui-ci, rendant ce type de solutions éloignées en terme de précision structurelle et temporelle des architectures cibles.

Les approches récentes basées sur l'exécution native du logiciel cherchent maintenant à réintroduire un maximum de précision dans les plateformes de simulation. Pour cela, les solutions actuellement proposées ont pour objectif de fournir un support d'exécution plus réaliste, en permettant au logiciel de prendre en compte les spécificités de l'architecture matérielle sous-jacente et la modélisation du temps d'exécution du logiciel, par l'analyse et l'instrumentation de celui-ci.

Ce chapitre présente l'état de l'art de ces deux idées directrices qui sont aussi celles de cette thèse, (1) l'exécution native du logiciel sur des plateformes de simulation de systèmes multiprocesseurs, (2) l'instrumentation du code logiciel permettant l'estimation des performances d'exécution.

3.1 Plateformes de simulation native pour les systèmes multiprocesseurs

Il existe principalement deux types d'approches pour la conception de plateformes de simulation native dédiées aux systèmes multiprocesseurs. La première est basée sur l'encapsulation de certaines parties du logiciel (généralement les tâches de l'application) et la seconde repose sur l'organisation en couches du logiciel en utilisant les différentes interfaces (API) existantes.

3.1.1 L'encapsulation du logiciel

Les approches basées sur l'encapsulation du logiciel sont les plus répandues en simulation native et correspondent au niveau TLM *natif* déjà présenté dans la problématique. Elles consistent à intégrer certaines parties du code logiciel dans des modules matériels représentants les processeurs [GYNJ01, BYJ02, WSH08, CHB09b]. Ce sont généralement les tâches logicielles qui sont implémentées dans les mêmes modules qui servent à modéliser les composants matériels (figure 3.1) et seront donc simulés en tant que tel (c'est-à-dire de manière concurrente). Cette solution impose donc certaines contraintes d'implémentation des tâches logicielles qui doivent reposer notamment directement sur l'interface matérielle des modules pour communiquer (généralement celles des ports TLM en SystemC).

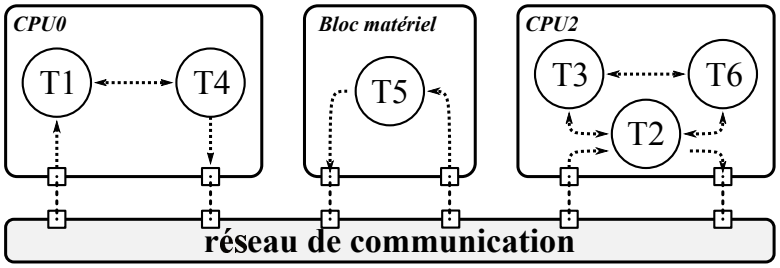


FIGURE 3.1 – Encapsulation du code logiciel en simulation native

Plusieurs travaux proposent de modéliser le système d'exploitation qui prendra en charge l'exécution de ces tâches à partir du langage de description utilisé pour le matériel. On peut citer [GYG03, SD08] pour SpecC ou encore [LMPC04, HWTT04, PAS+06, NST06] pour SystemC. Ces solutions (figure 3.2(a)) permettent aux tâches logicielles de s'exécuter nativement et de manière séquentielle dans le contexte du module représentant le processeur, en respectant la politique d'ordonnancement implémentée dans le modèle du système d'exploitation, améliorant ainsi le « réalisme » de la simulation.

Une solution différente est proposée dans [CBR+04], où les tâches logicielles sont toujours encapsulées dans des modules SystemC mais plutôt que d'intégrer un modèle de système d'exploitation dans SystemC, c'est un système d'exploitation à part entière qui ordonnance les modules représentant des tâches logicielles (figure 3.2(b)). Cela confère à ce type de solutions plusieurs avantages, notamment une certaine simplicité dans le mécanisme de raffinement permettant de passer entre les différents niveaux d'abstraction existants. Elle permet également de faciliter le processus d'exploration d'architecture en modifiant directement le type (logiciel/matériel) des modules SystemC et offre la possibilité d'exécuter et donc de valider une partie importante du logiciel final.

Un problème commun à toutes ces solutions est que les tâches restent fortement liées aux modules matériels leur servant de support, ce qui rend impossible la modélisation

d'architectures de type SMP. Ces approches montrent également leurs limites dès lors que certains aspects du matériel doivent être considérés plus en détails. Un exemple typique concerne les accès à la mémoire effectués depuis le logiciel exécuté nativement et pour lesquels le simulateur s'exécutant sur la machine hôte n'a aucun contrôle.

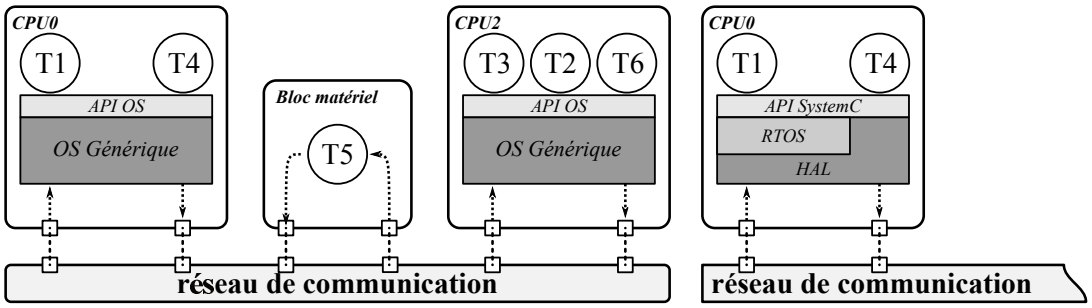


FIGURE 3.2 – Simulation native des tâches logicielles (a) au dessus d'un modèle de système d'exploitation et (b) au dessus d'un système d'exploitation réel

Ce problème est adressé dans [KKW+06, WSH08, CHB09b]. Les solutions proposées consistent à instrumenter automatiquement le code des tâches logicielles pour insérer des appels à des fonctions spécifiques lors des accès aux différentes mémoires. Ces appels seront interceptés par le modèle de simulation du matériel pour ensuite être redirigés vers les composants concernés après une conversion d'adresse entre les espaces mémoire de la machine hôte et de la plateforme de simulation. Ces solutions entrent typiquement dans le cas d'hétérogénéité des espaces mémoire présentés précédemment dans le chapitre de problématique, rendant impossible la modélisation de certaines interactions entre le logiciel et le matériel (exemple d'utilisation d'un DMA).

3.1.2 Exécution native basée sur une interface logicielle

Par opposition aux approches où le logiciel est encapsulé dans les modules de la plateforme de simulation matérielle, les solutions basées sur l'utilisation d'API logicielles reposent sur le fait que les modules matériels peuvent fournir une implémentation des primitives de l'API logicielle réelle, étant implémentés eux-mêmes dans un langage de programmation initialement dédié au logiciel. Les couches logicielles supérieures à cette API peuvent donc être compilées de manière complètement indépendantes (par rapport au modèle matériel) et leur exécution sur la plateforme de simulation repose uniquement sur l'implémentation des couches logicielles inférieures dans les modules matériels (figure 3.3(a)).

Dans [YBB+03, BYJ04] l'API logicielle utilisée est celle de la couche HAL sur laquelle repose le système d'exploitation, permettant ainsi de compiler et d'exécuter nativement la quasi totalité de la pile logicielle (figure 3.3(b)) indépendamment de la plateforme matérielle. Celle-ci peut donc être utilisée sans qu'il soit nécessaire d'apporter de modifications pour simuler toutes les piles logicielles reposant sur la même API.

Partant de ce principe, et afin de fournir des niveaux d'abstraction plus élevés, d'autres approches [BBY+05, TRKA07, PJ07] se basent sur l'interface fournie par le système

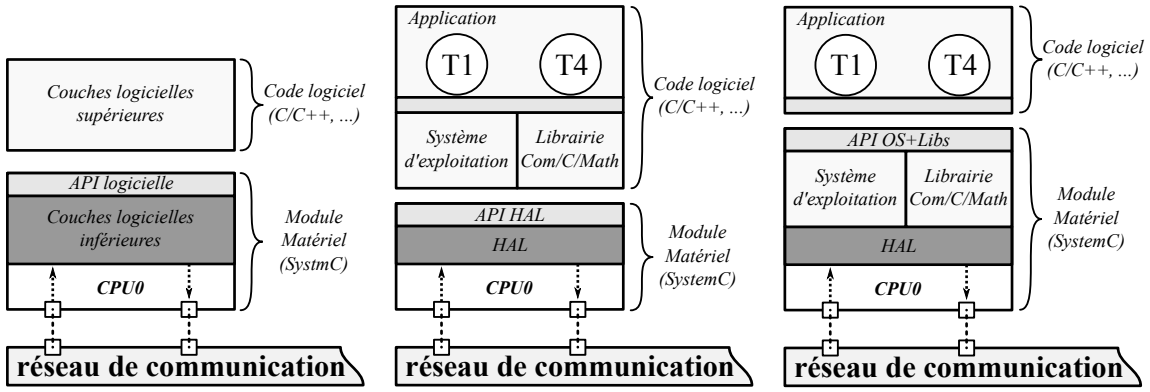


FIGURE 3.3 – Simulation native du logiciel au dessus d'une API (a) principe, (b) HAL et (c) Système d'Exploitation

d'exploitation à l'application (figure 3.3(c)). La mise en œuvre de cette dernière solution reste cependant difficile sur des applications réalistes en raison du nombre très important de fonctions que doit fournir l'API à ce niveau. En théorie, toutes les fonctions utilsables par l'application doivent être fournies par le module SystemC supportant son exécution, incluant aussi bien celles du système d'exploitation que celles des bibliothèques utilisées (bibliothèques mathématiques, C standard, de communications, etc.). En pratique, seules certaines primitives sont implémentées, ce qui impose des contraintes sur l'écriture de l'application logicielle. Dans certain cas, l'API C standard disponible sur la machine hôte est directement utilisée, introduisant par la même occasion une perte de précision car celle-ci s'exécute « en dehors » du contexte de la simulation.

Les principales limitations de ces approches que nous pouvons qualifier de « purement » natives ont été présentées dans la problématique. Comme pour les approches par encapsulation et malgré l'utilisation d'une API très bas niveau, les interactions fines entre le logiciel et le matériel sont confrontées à des problèmes d'hétérogénéités des espaces mémoires utilisés par les modules matériels de la plateforme de simulation et par le logiciel s'exécutant nativement. Un point également très limitant est l'exécution en temps nul du logiciel natif, rendant ce type de solution utilisable uniquement pour de la validation fonctionnelle et sur une catégorie d'application réduite desquelles sont exclues les applications fortement dépendantes du temps.

3.1.3 Les solutions hybrides

Certaines solutions se basent partiellement sur la simulation native. Les approches *hybrides* [BPN+04, MRRJ05, GKK+08, KEBR08] tirent à la fois profit des vitesses de simulation élevées qu'offrent les approches natives et de la précision des modèles à base d'ISS.

Dans [KGW+07] et [GKL+07], le mécanisme de simulation s'organise autour de deux outils de simulation, l'un utilisant un ISS et l'autre un modèle de processeur virtuel supportant l'exécution native. Une logique de contrôle permet de gérer le passage du mode d'interprétation du programme par l'ISS au mode d'exécution native du logiciel, qui ne peut se faire que lors d'appels de fonctions. Le code source est compilé d'une part pour le

processeur cible, et d'autre part pour le processeur hôte, après une phase d'instrumentation, permettant de définir les points de synchronisation possible qui seront utilisés pour commuter entre les deux simulateurs. L'architecture de l'environnement *HySim* utilisé dans [KGW+07, GKL+07] et [GKK+08] est représenté sur la figure 3.4.

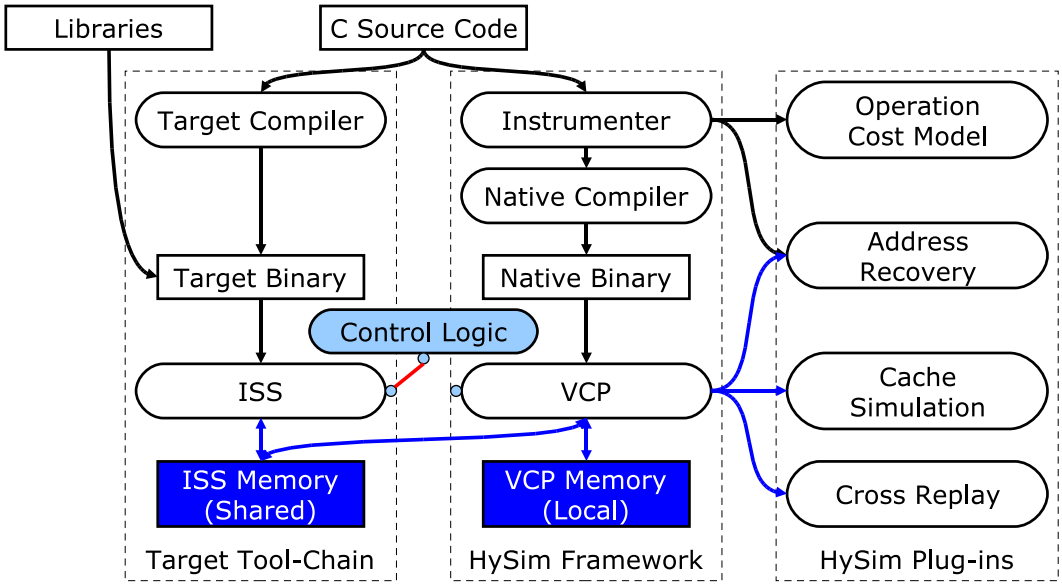


FIGURE 3.4 – Architecture de l'environnement HySim (figure tirée de [GKK+08])

La principale difficulté dans ce type d'approche reste le mécanisme de contrôle permettant de sélectionner le mode de simulation qui sera utilisé. Dans [GKK+08], cette sélection est faite dynamiquement lors de la simulation. L'exécution native du code logiciel instrumenté permet de générer la trace du flot de contrôle d'une fonction (un scénario). Si ce scénario n'a pas encore été enregistré, la fonction exécutée nativement est alors *rejouée* sur l'ISS pour obtenir et enregistrer les informations permettant de prendre en compte la précision de la plateforme matérielle. Trois types d'enregistrement, qui sont les valeurs d'entrée de fonction, les accès mémoire et la valeur de retour, permettent de déterminer si une fonction doit être *rejouée*.

Dans [MRRJ05] et [KGW+07], cette sélection est faite statiquement à la compilation. Les appels des fonctions à exécuter nativement sont remplacés dans le code interprété par des mécanismes similaires aux RPC (Remote Procedure Call). La sélection des fonctions candidates est basée dans [MRRJ05] sur une contrainte du taux d'erreur maximum probable de l'estimation de l'énergie [MRRJ04], alors que [KGW+07] utilise une méthode manuelle, permettant soit de choisir des fonctions précises, soit de déterminer un point du code source à atteindre rapidement. Dans [KEBR08], cette sélection est faite au niveau de l'interface logicielle du système d'exploitation. Ce dernier est modélisé en SystemC et le code binaire de l'application est interprété par un ISS.

Les principales limitations dans ces approches sont liées à la complexité des mécanismes d'interaction et de maintien de la cohérence entre les ISS et les modèles de simulation rapide. La précision des simulations est généralement bonne grâce à la prise en compte des détails architecturaux de la plateforme matérielle (accès mémoire, congestion de communication, etc.) mais les performances de simulation restent très limitées (inférieures à dix fois la vitesse de simulation des ISS).

3.2 L'estimation des performances dans les systèmes multiprocesseurs

L'estimation des performances dans les systèmes multiprocesseurs à des niveaux d'abstraction élevés et plus particulièrement celle du temps d'exécution du logiciel, représente aujourd'hui pour l'exploration d'architecture plus qu'un besoin, une nécessité. Même si elle n'est pas indispensable à la validation du logiciel, elle offre également un support supplémentaire très appréciable pour le débogage des applications fortement dépendantes du temps.

La plupart des méthodes utilisées dans les approches récentes sont basées sur des techniques d'estimation de performances ayant fait leurs preuves sur des architectures monoprocesseur. Un des critères pouvant être utilisé pour les classifier est le niveau d'abstraction auquel ces méthodes d'estimation opèrent, correspondant chacun à une « *vue* » du logiciel (il ne s'agit pas ici des niveaux d'abstraction des modèles de simulation présentés dans le chapitre précédent). Dans l'ordre croissant d'abstraction, nous pouvons trouver dans la littérature :

- Le niveau code binaire ou assembleur (BLS pour Binary Level Simulation)
- Le niveau représentation intermédiaire (IRLS pour Intermediate Representation Level Simulation)
- Le niveau source (SLS pour System Level Simulation)

Un autre critère incontournable dès lors que l'on s'intéresse au problème de l'estimation des performances est celui du type d'analyse effectuée : statique vs dynamique. Nous n'avons cependant pas choisi d'utiliser ces critères afin de différencier les approches utilisées dans le domaine de la simulation des architectures MPSoC pour la simple raison que la plupart d'entre elles sont basées sur une utilisation conjointe de ces deux méthodes d'estimation.

Les méthodes statiques, qu'elles soient basées sur des approches WCET [MML97, WEE+08], linéaires [GMH01, HR00] ou non linéaires [BK02, OZW04] présentent toutes plus ou moins de difficultés à prendre en compte le comportement du logiciel au moment de son exécution. Cette limitation est d'autant plus accentuée lorsque l'exécution de ce logiciel peut migrer entre des processeurs comme c'est le cas sur des architectures SMP.

Une des solutions mise en œuvre dans la plupart des approches efficaces [WSH08, CHB09a, ILK+, WH09] est d'utiliser l'analyse statique à une granularité plus fine dans le logiciel, la partie dynamique de l'estimation étant obtenue par la simulation du programme instrumenté.

Les différentes méthodes d'analyses statiques existantes ne font pas l'objet de l'étude qui suit, dans laquelle nous tenterons plutôt de montrer les principales solutions proposées dans l'état de l'art, et plus précisément celles sur les techniques d'instrumentation utilisées dans le logiciel exécuté sur des plateformes de simulation d'architectures MPSoC natives.

3.2.1 Estimation au niveau du code objet

Les approches au niveau objet sont basées sur l'analyse et la conversion du code assembleur du programme directement obtenu à partir des chaines de compilation dédiées au processeur cible, bénéficiant ainsi de toutes les optimisations réalisées dans le compilateur, en une représentation utilisant un langage de plus haut niveau.

La mise en œuvre la plus courante consiste à transformer les instructions assembleur en un code source C équivalent, dans lequel les annotations seront insérées [ZM96, LBH+00]. Les listings 3.1 et 3.2 tirés de [LBH+00] montrent respectivement un fichier assembleur obtenu à partir de la chaine de compilation d'un processeur MIPS et la vue « *assembleur-C* » générée par les outils de traduction et d'instrumentation. Dans ce dernier, chacune des instructions équivalentes en langage C utilise les registres émulés du processeur cible (typiquement sous forme de variables globales dans le programme) et sont précédées par un appel à la fonction `DELAY()` permettant de prendre en compte le nombre de cycles relatifs au type d'instruction donné en arguments.

Listing 3.1 Code assembleur

```
1  sb   $2, v__st_tmp.2
2  jal  startup
3  lw   $2, proc
4
5  sll  $2, $2, 2
6  lw   $2, frozen_inp_events($2)
7  lbu  $4, 0($2)
```

Listing 3.2 Code « assembleur-C »

```
1  DELAY(sb);          v__st_tmp = _R2;
2  DELAY(jal);         // startup(proc); deferred
3  DELAY(lw+nop);      _R2 = proc;
4                      startup(proc);
5  DELAY(sll);         _R2 = _R2 << 2;
6  DELAY(lw+nop);      _R2 = *(&frozen_inp_events + _R2);
7  DELAY(lbu+nop);     _R4 = *(0 + _R2);
```

Le code « *assembleur-C* » est ensuite recompilé pour le processeur de la machine hôte, pour être finalement exécuté nativement. Ces solutions permettent de prendre en compte les optimisations des compilateurs spécifiques aux processeurs cible et l'instrumentation du code source est relativement simple en raison du lien direct qui existe entre le code assembleur et la vue « *assembleur-C* ». Le code instrumenté peut cependant être soumis à des transformations importantes lors de la compilation pour le processeur hôte, pouvant nécessiter l'adaptation de la technique d'annotation en fonction du compilateur et des optimisations réalisées [BKL+00]. Certains aspects dynamiques de l'exécution sont également complexes à prendre en compte lors de l'analyse statique des instructions du code assembleur, tels que les branchements indirects.

3.2.2 Estimation au niveau source

L'estimation des performances au niveau du code source consiste à utiliser directement le code source d'origine pour y insérer des annotations. Ces annotations fournissent un ensemble d'informations relatives à une portion de code. Ces informations permettent l'estimation de différentes grandeurs physiques, liées à l'exécution de la portion de code associée, telle que le temps d'exécution, la consommation ou encore le nombre d'accès à la mémoire.

Dans [WSH08], le code source est tout d'abord compilé pour le processeur cible. Les informations de débogage sont ensuite utilisées pour faire le lien entre les instructions en assembleur du code binaire cible et les lignes du code source d'origine. Cette solution reste complexe à implémenter car l'insertion des annotations est tributaire de la syntaxe du

langage source utilisé et l'insertion de ces annotations aux bons endroits peut nécessiter des modifications d'écriture dans le code source. Les optimisations des compilateurs peuvent également accentuer ce problème.

[BKL+00] propose également d'instrumenter directement le code source du programme. Pour cela, le programme est partiellement compilé en un jeu d'instructions virtuelles indépendantes du processeur cible. L'instrumentation du code source d'origine consiste à insérer les annotations qui permettront de prendre en compte les instructions virtuelles générées à partir de chaque ligne du programme. Le nombre de cycles d'exécution liés à chaque instruction n'est pas directement donné dans l'annotation mais il sera obtenu à partir d'un fichier de description du processeur. Ainsi, changer de processeur revient uniquement à utiliser un autre fichier de description. Une solution identique est utilisée dans [GMH01] et [MSVSL08]. Afin de minimiser les problèmes liés à la syntaxe du langage utilisé, [MSVSL08] préconise l'emploi d'outils d'amélioration de la syntaxe avant d'effectuer l'instrumentation.

Il est très difficile dans ce type d'approche de prendre en compte les optimisations des compilateurs. Par exemple, dans [BKL+00] aucune considération n'est faite sur le nombre de registres du processeur cible.

Dans [ILK+], l'instrumentation du code source est utilisée non pas pour obtenir des informations permettant d'estimer les performances lors de la simulation, mais pour générer la trace d'exécution du CFG du programme sous forme d'un *bitstream*. L'instrumentation consiste à utiliser des *macros* sur chacune des opérations de branchements. Là encore cette solution est tributaire de la richesse de la syntaxe du langage utilisé. Les *switch* du langage C ou les appels de fonctions par pointeurs par exemple ne permettent pas à l'instrumentation de générer le *bitstream*. En plus des problèmes classiques des approches basées sur les traces de simulation (dépendantes des données utilisées en entrées lors de la génération de ces traces), cette solution ne permet pas de prendre en compte les transformations importantes ayant pu être effectuées sur le CFG du programme lors de la compilation pour le processeur cible.

L'approche proposée dans [PHS+04] ne consiste pas directement à instrumenter le code source du programme, mais elle repose sur le polymorphisme du C++ pour surcharger les opérateurs de base afin d'injecter automatiquement des annotations de performances au moment de l'exécution native. Bien que l'auteur reporte des précisions raisonnables en terme d'estimation de performances, il n'est pas donné de détails supplémentaires permettant de comprendre comment ces résultats prennent en compte les optimisations spécifiques au processeur cible à un tel niveau d'instrumentation.

D'une manière générale, les solutions basées directement sur l'annotation du code source présentent un intérêt pour permettre la validation des applications sur les plateformes de simulation qui requièrent la modélisation du temps, sans pour autant nécessiter un niveau de précision important.

3.2.3 Estimation au niveau IR

L'utilisation de la représentation intermédiaire sur laquelle dans les compilateurs se fondent permet la prise en compte des principaux facteurs déterminant dans le processus d'estimation des performances.

La génération d'une représentation intermédiaire trois adresses (*3-AC IR*) est utilisée dans [KAFK+05] et [KKW+06]. Cette représentation intermédiaire utilise la syntaxe du langage C et peut donc être directement instrumentée puis compilée pour le processeur hôte. Cette solution permet de prendre en compte les optimisations du compilateur ainsi

que l'aspect dynamique de l'exécution du programme mais les informations de débogage sur le programme d'origine sont par défaut perdues et la reconstruction de celles-ci à partir des sections disponibles dans le code final ne semble pas immédiate.

Dans [HAG08], la représentation intermédiaire issue du *front-end* de l'environnement de développement *ESE* [ESE] est instrumentée pour prendre en compte des annotations temporelles. À partir de cette représentation intermédiaire annotée temporellement, un code C/C++ est généré afin d'être utilisé dans un environnement de cosimulation. Cependant, cette instrumentation ne prend en compte que l'aspect dynamique du graphe de contrôle du programme (CFG) issue du *front-end*, sans tenir compte des optimisations qui auraient certainement affectées celui-ci lors du *back-end* du compilateur.

Une approche similaire est utilisée dans [CHB09a] pour générer des modules SystemC contenant le code des tâches logicielles instrumentées. Ces modules SystemC peuvent être utilisés sur des plateformes de simulation au dessus de modèles de systèmes d'exploitation tels que [LMPC04, NST06]. Contrairement à l'approche utilisée dans [HAG08], la solution proposée permet de prendre en compte les optimisations effectuées par le compilateur ainsi que les aspects liés à l'architecture interne des processeurs, tels que les pénalités de branchement ou les inter-blocages entre les instructions. Bien que les accès en mémoire de données soient pris en compte lors de l'instrumentation, celle ci ne considère pas les accès aux mémoires contenant les instructions, générés par l'exécution du programme lui même. Enfin, dans les deux approches présentées ici, aucune ne permet le débogage du programme en utilisant son code source d'origine (lors du débogage, le code source visible sera celui du code instrumenté).

L'approche utilisée dans *iSciSim* et proposée dans [WH09] est une évolution de la solution [WSH08] basée initialement sur l'instrumentation au niveau source. Elle repose sur l'utilisation de la représentation intermédiaire *GIMPLE* utilisée dans GCC. L'architecture de *iSciSim* est représentée sur la figure 3.5.

Dans la première phase, le code source est tout d'abord *cross compilé* pour le processeur cible. A l'issue de cette compilation, la représentation intermédiaire est directement fournie dans un fichier par GCC au format persistant *GIMPLE*, proche de la syntaxe du langage C. Ce « *pseudo* » code C est ensuite converti en véritable code C nommé ISC (pour Intermediate Source Code).

La seconde phase consiste à *cross compiler* à nouveau le fichier ISC pour le processeur cible puis à insérer dans celui-ci les annotations temporelles pour obtenir un fichier ISC instrumenté. L'analyse statique du fichier binaire permet d'obtenir les informations temporelles et l'analyse du fichier de débogage permet d'extraire les informations de localisation des données en mémoire qui seront nécessaires lors de la simulation afin de modéliser l'aspect dynamique de l'exécution du logiciel.

Finalement, ce code *ISC* instrumenté est finalement compilé pour la machine hôte. L'exécution native du programme permet alors de prendre en compte les aspects dynamiques tels que ceux liées aux mémoires caches. Cette méthode présente un certain nombre d'avantages tel que la capacité à prendre en compte de manière précise à la fois les aspects statiques (latence d'exécution des instructions) ou les effets liés au pipeline (toutes les instructions sont connues lors de l'analyse) et les aspects dynamiques comme la prédiction de branchement.

Un premier inconvénient à cette solution est une vitesse de simulation relativement réduite lorsque que l'instrumentation prend en compte les effets de cache, en raison d'un nombre important de synchronisations nécessaires avec le modèle de simulation du ma-

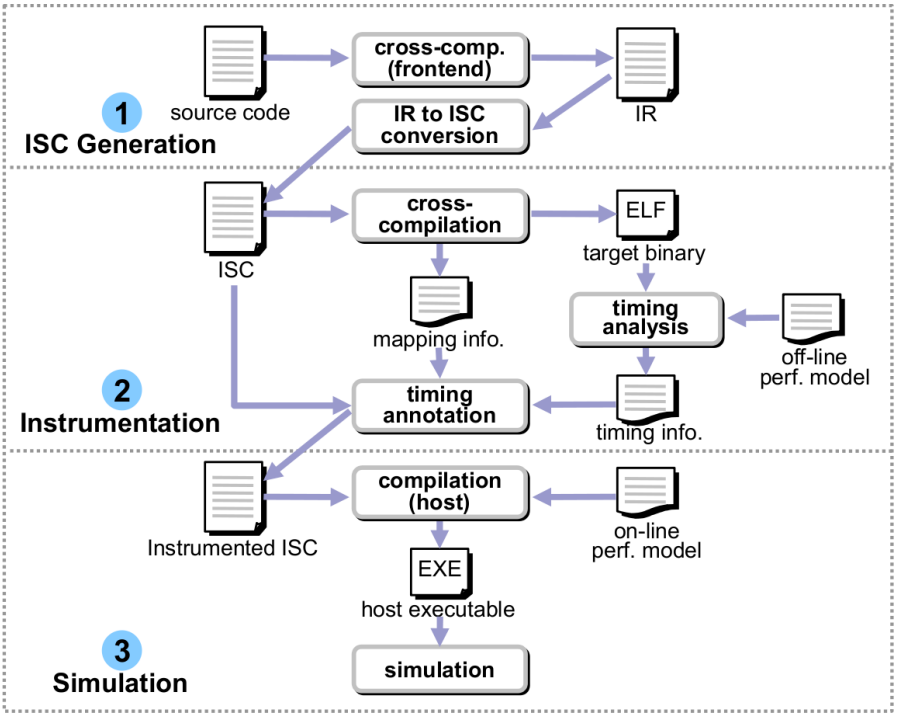


FIGURE 3.5 – Architecture de l'approche iSciSim (figure extraite de [WH09])

tériel. De plus, le débogage de l'application se fait directement au niveau du code *ISC* instrumenté. Il est alors quasiment impossible de « remonter » jusqu'au code source d'origine, ce qui rend cette solution difficilement utilisable pour la validation du logiciel.

3.3 Conclusion

Nous avons présenté, dans ce chapitre, les principales solutions proposées dans les deux axes de recherches de cette thèse qui sont l'exécution native du logiciel sur des plateformes de simulation de systèmes multiprocesseurs et l'instrumentation du code logiciel.

L'utilisation des approches basées sur les interfaces de simulation semble palier en partie au manque de « réalisme » des plateformes de simulation native. Toutefois, plusieurs aspects importants manquent encore à ces solutions pour les rendre applicables à des systèmes MPSoC beaucoup plus réalistes, aussi bien au niveau de l'architecture matérielle modélisée qu'au niveau du code embarqué.

Une plateforme de simulation reposant sur le même principe est proposée dans la suite. Elle permet au logiciel exécuté nativement de prendre en compte de manière beaucoup plus précise les aspects liés à l'architecture matérielle.

La technique d'instrumentation du logiciel proposée est basée sur la représentation intermédiaire des compilateurs. Par opposition aux techniques existantes, elle est complètement dissociée du mécanisme d'analyse d'estimation des performances.

Chapitre 4

Plateforme TLM pour l'exécution native de logiciel

Sommaire

La première contribution de cette thèse est la proposition d'une méthodologie de conception des plateformes de simulations basée sur l'exécution native du logiciel. Les plateformes de simulation ainsi développées permettent, malgré le niveau d'abstraction utilisé, l'exécution de la quasi totalité du logiciel final (système d'exploitation, pilotes de périphériques, application) sur des modèles réalistes de l'architecture matérielle du système.

Les différentes solutions mises en œuvres seront présentées en détails dans ce chapitre. Elles reposent premièrement sur l'utilisation des Unités d'Exécution comme uniques interfaces entre le logiciel et le matériel et deuxièmement sur l'uniformisation des espaces mémoires entre la plateforme de simulation et le logiciel exécuté nativement.

4.1 Rappels

La problématique cernée dans le chapitre 2 est brièvement rappelée ici. La figure 4.1 représente la plateforme de simulation native au niveau Transaction Accurate telle qu'elle est définie dans [BBY+05] et sur laquelle est basée notre solution.

La partie matérielle de cette plateforme est réalisée par un modèle générique (en SystemC), fournissant également une implémentation de la couche logicielle HAL spécifique à la machine hôte. La partie logicielle reposant sur l'API HAL est compilée pour le processeur hôte et encapsulée dans une bibliothèque dynamique. Rappelons également que lorsque le logiciel natif s'exécute, tous les accès en mémoire (de données et d'instructions) se font à l'insu de la simulation SystemC. Les interactions entre le logiciel et le matériel peuvent se faire uniquement lors des appels aux fonctions de l'API HAL.

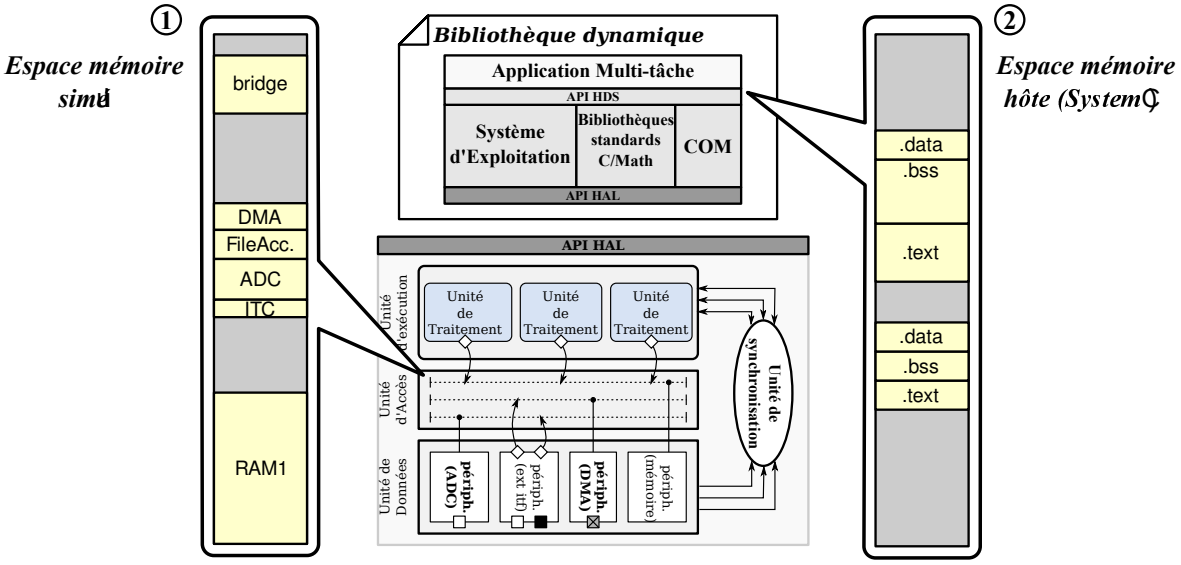


FIGURE 4.1 – Plateforme de simulation TA

Un des problèmes soulevé dans le chapitre 2 concernait les limitations de cette plateforme à supporter l'exécution native de logiciel sans apporter de modifications à son code source. Pour cela, nous avons mis en évidence deux sources de dépendances entre le logiciel et le matériel.

La première, liée directement au processeur (par son jeu d'instructions) et aux composants de la plateforme, est résolue par l'utilisation d'une séparation stricte entre le matériel et le logiciel. Cette séparation est fournie par la couche HAL et est communément acceptée par les développeurs de systèmes d'exploitation.

La deuxième source de dépendance non résolue est liée à l'utilisation de deux représentations de la mémoire, celle simulée par la plateforme matérielle ① et la représentation mémoire réellement considérée par le logiciel natif lors de son exécution, correspondant à celle utilisée par l'exécutable SystemC ②.

Dans la suite de ce chapitre, nous proposons une solution permettant de prendre en compte cette seconde source de dépendance afin de permettre l'exécution d'une pile logicielle complète dans un environnement réaliste. Avant cela, nous présenterons notre solution mise en œuvre pour utiliser des modèles de simulation TLM *classiques* en lieu et place des modèles génériques proposés dans les travaux antérieurs. Cette solution consiste à utiliser les Unités d'Exécution comme uniques interfaces entre le logiciel et le matériel.

4.2 Les Unités d'Exécution : Uniques interfaces logicielle/matérielle

Le concept d'Unité d'Exécution (EU) initialement proposé dans [BBY+05] et présenté dans le chapitre 2.4.2 permet de fournir une partie du support d'exécution au logiciel natif, le reste étant fourni par des composants spécifiques du modèle générique de la plateforme de simulation matérielle. Cette généricité va à l'encontre d'une validation précise du logiciel, qui nécessite une modélisation plus fine du matériel. Dans l'approche proposée, les EU sont utilisées comme unique support à l'exécution du logiciel natif. Ce choix présente l'intérêt de pouvoir utiliser des modèles TLM de plateformes matérielles classiques. Idéalement, les EU peuvent directement remplacer les ISS utilisés en simulation TLM *classiques*. Elles deviennent ainsi l'unique interface entre le logiciel et le matériel de l'application, au même titre que l'est un processeur tel qu'il est défini dans [PH98].

Une EU est un composant hybride [GSC+07] permettant d'interfacer une plateforme matérielle, décrite à un niveau d'abstraction donné, à un environnement logiciel (un programme) au travers d'une API quelconque. Dans notre contexte, l'interface matérielle d'une EU doit être connectée avec le réseau de communication du sous-système processeur comme le seraient les ISS. L'interface logicielle doit quant à elle fournir toute l'API HAL sur laquelle repose l'exécution des couches logicielles supérieures.

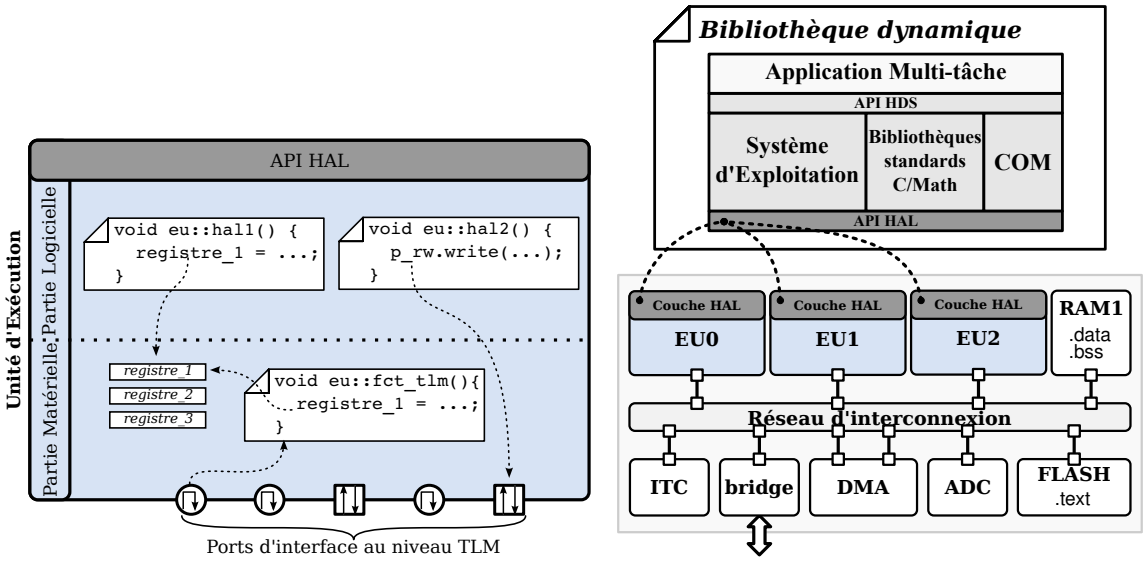
L'implémentation d'une EU est schématisée sur la figure 4.2(a). La partie matérielle est composée des registres internes au processeur et de l'interface TLM permettant de communiquer avec les autres composants de l'architecture matérielle. La partie logicielle contient quant à elle l'implémentation des fonctions de l'API HAL. Cette implémentation mixte du matériel et du logiciel au sein d'une même entité permet de modéliser simplement et efficacement les interactions entre le logiciel et le matériel. Par exemple, les registres internes pouvant être modélisés dans une EU sont accessibles aussi bien à partir des fonctions qui implémentent le comportement du matériel (`fct_tlm()` dans la figure 4.2(a)) qu'à partir des fonctions de la couche HAL (`hal1()`). Ces dernières ont également accès aux ports TLM de l'interface matérielle, permettant par exemple d'effectuer des opérations d'entrées/sorties (`hal2()`).

La plateforme de simulation native générique présentée jusqu'alors devient donc une plateforme de simulation TLM *classique* comme elle est représentée sur la figure 4.2(b), où chaque processeur est modélisé par une EU. Le principe d'exécution du logiciel sur ces EU est identique à celui présenté dans la section 2.4.2.

4.2.1 Partie logicielle d'une EU

Le fait de se baser sur une API HAL ne constitue en rien une contrainte liée à l'exécution native du logiciel. La seule contrainte étant qu'il doit exister une API strictement hermétique entre le logiciel à exécuter nativement et la plateforme de simulation sous-jacente. Par exemple, une EU dont l'interface logicielle serait constituée de toutes les fonctions de l'API POSIX et de la bibliothèque C standard (LibC) pourrait directement servir de support d'exécution à la couche applicative de la pile logicielle, malgré le fait que cette dernière solution ne soit pas facilement réalisable en pratique (comme nous l'avions évoqué dans la section 3.1.2).

L'utilisation de l'API HAL présente cependant plusieurs intérêts. Tout d'abord, plus l'interface avec le logiciel sera de bas niveau, plus la quantité de code validé sera importante (système d'exploitation, pilotes de périphériques, etc.). Ensuite, le nombre de fonctions



(a) Principe de réalisation d'une EU

(b) EU sur une plateforme de simulation TLM *classique*

FIGURE 4.2 – Principe de réalisation d'une EU

constituant ce type d'API est généralement très limité (de quelques dizaines à une centaine de fonctions en général) et leur implémentation très basique par rapport à des API de plus haut niveau telles que l'API POSIX.

4.2.1.1 Implémentation de la couche HAL

Au même titre qu'il n'existe pas un HAL unique (chaque système d'exploitation définissant son propre HAL), il n'existe pas une implémentation unique d'EU. Ainsi, le portage d'un nouveau système d'exploitation sur la plateforme de simulation native nécessite le développement d'une nouvelle EU, de la même manière que le portage vers une nouveau processeur.

Il n'existe donc pas de règles précises pour l'implémentation de la couche HAL dans une EU. Cependant, à la différence d'une implémentation *classique* pour une plateforme réelle où seules les ressources du processeur cible peuvent être utilisées, une implémentation de couche HAL pour une EU consiste à utiliser toutes les ressources disponibles de la machine hôte, permettant de modéliser le comportement de chacune des fonctions de l'API .

Ces ressources peuvent aussi bien être celles de la machine hôte, comme des fonctions système ou des registres du processeur hôte, que celles modélisées dans l'EU, telles que des registres simulés ou encore l'interface avec le réseau de communication. L'important étant que lors de l'exécution de l'application, tout se passe « comme si » les services attendus par celle-ci avaient été fournis.

Un exemple typique de fonctions HAL nécessitant des ressources du système d'exploitation de la machine hôte concerne la gestion des contextes d'exécution du processeur. Ces fonctions permettent de sauvegarder l'état du processeur en mémoire (registres généraux, registres d'états, etc.) et de charger un état précédemment sauvegardé pour reprendre

l'exécution d'un ancien contexte. Elles sont utilisées par les systèmes d'exploitation pour la gestion des processus dits « *légers* » (en anglais, *thread*). Étant donné qu'il s'agit d'une exécution native, le processeur concerné est celui de la machine hôte, et non le processeur cible. Dans ce cas, des fonctions de gestion de contexte au niveau utilisateur peuvent être utilisées si le système d'exploitation de la machine hôte le permet. Dans le cas contraire, il sera nécessaire d'implémenter ses propres fonctions de manipulation de contexte pour le processeur hôte.

La norme POSIX offre à cet effet un certain nombre de fonctions (cf. pages de manuel de `makecontext()`). Le listing 4.1 est un extrait d'implémentation de trois fonctions *HAL* permettant à un système d'exploitation (celui simulé) de manipuler des contextes d'exécution.

Listing 4.1 Exemple de fonctions HAL fournies par une EU pour la gestion de contexte de processus « léger »

```
1  #include <ucontext.h>
2
3  void init_context(ucontext_t **ctx,void *stack,int32_t ssize,void *entry,void *↩
       arg){
4    *ctx = (ucontext_t *) malloc(sizeof(ucontext_t));
5    getcontext(new_ctx);
6    new_ctx->uc_stack.ss_sp = stack;
7    new_ctx->uc_stack.ss_size = ssize;
8    makecontext(new_ctx, entry, 1, arg);
9  }
10
11 void load_context(ucontext_t **to) {
12   setcontext(*to);
13 }
14
15 void switch_context(ucontext_t **from, ucontext_t **to) {
16   swapcontext(*from, *to);
17 }
```

La fonction `init_context()` (ligne 3) permet d'initialiser un nouveau contexte en utilisant les fonctions du système d'exploitation hôte `getcontext()` et `makecontext()` (lignes 5 et 8). Le contexte ainsi créé exécutera la fonction `entry()` dont l'adresse est donnée en argument (ligne 3) lorsqu'il sera chargé en mémoire par l'appel aux fonctions `load_context()` ou `switch_context()` (lignes 11 et 15). Ces dernières utilisent directement les fonctions POSIX `setcontext()` et `swapcontext()` (lignes 12 et 16).

Un exemple d'utilisation de ressources modélisées par une EU a déjà été présenté lors de la description du principe de l'exécution native (section 2.4.2). Elle est brièvement rappelée ici.

L'interface matérielle connectée au réseau de communication d'une EU est l'une de ces ressources. Elle peut typiquement être utilisée par des fonctions HAL permettant d'accéder aux registres des périphériques connectés sur ce réseau. Le listing 4.2 donne l'implémentation d'une fonction HAL permettant de lire un mot de 32*bits* à l'adresse `address`. Dans cet exemple, on suppose que l'EU est implémentée en SystemC et dispose d'un port `p_rw` permettant de faire un accès en lecture vers l'extérieur. La méthode C++ de lecture fournie par ce port peut être directement utilisée dans la fonction `__hal_read()` pour réaliser l'accès mémoire (ligne 3).

Les ressources de la machine hôte ainsi que celles modélisées dans les EU peuvent ne pas être suffisantes pour permettre l'implémentation de la couche logicielle HAL. Par exemple, l'utilisation d'unités de gestion de mémoires virtuelles (MMU) dans les architec-

Listing 4.2 Exemple de fonction HAL d'accès en lecture à une adresse donnée

```
1  eu::__hal_read(uint32_t address, uint32_t &data)
2  {
3    p_rw->read(address,data);
4  }
```

tures matérielles peuvent poser un certain nombre de problèmes techniques en simulation native et ne sont pas supportées pour l'instant dans notre plateforme. Ceci nous permet de mettre en évidence une première limitation à cette approche.

Limitation 4.1 Implémentation de la couche HAL

La capacité à implémenter la couche logicielle HAL dans une EU est fortement liée à la capacité à modéliser le comportement attendu uniquement à l'aide des ressources de la machine hôte et de celles modélisées dans les EU, sans apporter de modification au code source de la pile logicielle.

4.2.1.2 L'API HAL comme point de synchronisation

Dans le cadre de la simulation native, le rôle des fonctions de l'API HAL ne se limite pas à l'abstraction du matériel. Elles servent également de points de synchronisation sans lesquels le moteur de simulation de SystemC ne pourrait pas faire évoluer le temps (cf. point ❺ sur l'exécution du logiciel natif dans la section 2.4.2). Cela signifie que toutes les fonctions HAL doivent *impérativement* consommer du temps de simulation SystemC par des appels à la fonction `wait()`, sans quoi la simulation pourrait se « figer »[1]. Nous parlerons de temps de synchronisation que nous nommerons $Tps_{synchro}$. Les fonctions HAL données dans le listing 4.1 en exemple doivent donc toutes commencer par un appel à la fonction `wait()` ayant pour argument le temps $Tps_{synchro}$, comme le montre le listing 4.3 (lignes 2, 11 et 16).

Listing 4.3 Points de synchronisation dans les fonctions HAL de gestion de contexte

```
1  void init_context(ucontext_t **ctx,void *stack,int32_t ssize,void *entry,void *↩
       arg) {
2    wait(Tps_synchro);
3    *ctx = (ucontext_t *) malloc(sizeof(ucontext_t));
4    getcontext(new_ctx);
5    new_ctx->uc_stack.ss_sp = stack;
6    new_ctx->uc_stack.ss_size = ssize;
7    makecontext(new_ctx, entry, 1, arg);
8  }
9
10 void load_context(ucontext_t **to) {
11   wait(Tps_synchro);
12   setcontext(*to);
13 }
14
15 void switch_context(ucontext_t **from, ucontext_t **to) {
16   wait(Tps_synchro);
17   swapcontext(*from, *to);
18 }
```

1. Une boucle d'attente sur une valeur de registre de périphérique en est un exemple. Elle ne permettrait jamais à SystemC de reprendre le contrôle de la simulation et donc de modifier la valeur de ce registre.

En l'absence totale d'informations sur le code exécuté nativement entre deux appels à des fonctions HAL, se pose le problème de savoir quelles valeurs de temps affecter à $Tps_{synchro}$ lors des appels à la fonction `wait()`. Étant donné la forte imprécision temporelle due à l'exécution en temps simulé nul du logiciel, une première approche consisterait à dire que cette valeur n'a que peu d'importance. Cela est vrai pour une simulation purement fonctionnelle où le matériel ne modélise pas de temps. Malheureusement, le choix de ce temps consommé « par défaut » à chaque appel aux fonctions du HAL a nécessairement une influence non négligeable sur le comportement de l'application. En effet, ce temps traduit indirectement les performances d'un processeur (modélisé par une EU) et donc sa capacité à exécuter correctement l'application logicielle. Ainsi, le problème se pose dès lors que l'application (l'ensemble de la pile logicielle) utilise la notion de temps. C'est le cas par exemple dans les systèmes d'exploitation utilisant des horloges pour l'ordonnancement des tâches de l'application logicielle, ou encore toutes les applications « temps réel ».

Pour garantir un fonctionnement correct de l'application par rapport aux contraintes temporelles, il faut :

- Qu'une tâche exécutée en réponse à un évènement externe s'exécute en un temps borné suffisamment court par rapport aux occurrences de cet évènement.
- Que l'ensemble des traitements de tous les évènements possibles ne dépasse pas les limites temporelles.

On parle alors de *condition de charge* dont l'objectif est de respecter l'équation suivante :

$$\sum_{i \in I} \frac{C_i}{T_i} \leq 1 \tag{4.1}$$

Où I est l'ensemble des sources d'évènements du système, i est une source d'évènement, C_i est le temps nécessaire à l'exécution de la tâche associée à l'évènement i et T_i le temps entre deux occurrences de cet évènement. L'objectif est de conserver une valeur inférieure à 1 pour garantir le fonctionnement correct du système. Cela passe entre autre par le choix adapté d'un processeur et de sa fréquence d'horloge.

Sur un processeur réel, la charge de calcul C_i est égale au temps que met le processeur à exécuter toutes les instructions nécessaires au traitement de la tâche i. Ce temps peut être calculé ou estimé à partir d'approches WCET (Worst Case Execution Time) ou tout simplement mesuré sur une plateforme précise (en simulation CABA/RTL ou réellement). Dans le cas de la simulation native, la granularité d'exécution du logiciel n'est plus définie par les instructions, mais par les appels aux fonctions HAL. La charge C_i devient alors égale au nombre d'appels $nb\_appels\_hal_i$ à l'API HAL effectués durant l'exécution de la tâche i multipliée par le temps de synchronisation $Tps_{synchro}$, consommé entre chaque appel. L'équation de condition de charge 4.1 devient alors :

$$\sum_{i \in I} \frac{nb\_appels\_hal_i \times Tps_{synchro_i}}{T_i} \leq 1 \tag{4.2}$$

En dehors de la solution idéale qui consisterait à consommer le temps qu'*aurait* mis réellement le logiciel à s'exécuter sur le processeur cible, nous n'avons aucune modélisation mathématique permettant d'obtenir une approximation du temps de simulation à consommer entre chaque appel aux fonctions du HAL. Ceci vient du fait qu'il n'existe aucune corrélation entre les appels aux HAL et le nombre d'instructions exécutées dans un programme. De plus, l'aspect dynamique de l'exécution du programme qu'il serait nécessaire de prendre en compte dans ce calcul est d'autant plus difficile à appréhender lorsque

l'application est partagée entre plusieurs processeurs. Néanmoins, ce calcul peut être utilisé afin de déterminer un temps de synchronisation, permettant de prendre en compte des évènements matériels tels que des interruptions, dans une application dont le logiciel ne contient pourtant aucune information temporelle. Le choix d'une valeur pour $Tps_{synchro}$ se résume donc à sélectionner une constante permettant de respecter la condition de charge du système :

$$0 < Tps_{synchro} \leq \frac{1}{\sum_{i \in I} \frac{nb\_appels\_hal_i}{T_i}} \tag{4.3}$$

De la même manière que le nombre d'instructions ou le temps nécessaire pour les exécuter est variable sur une architecture réelle (mémoire cache, congestion de communication, ...), le nombre d'appels à des fonctions HAL au cours d'un traitement peut varier. La valeur $nb\_appels\_hal_i$ doit donc représenter le nombre *maximum* d'appels possibles si l'on veut respecter la condition de charge, (équivalent à du temps réel « dur »). Ce nombre pourra être déterminé manuellement si la complexité de la tâche le permet (par exemple dans les routines de gestion d'interruption, généralement de tailles limitées), ou mesuré à l'aide d'outils de profilage logiciel. Il en est de même pour T_i qui doit alors être le minimum de temps séparant deux occurrences d'un évènement i.

En pratique, le choix d'une valeur trop petite pour $Tps_{synchro}$ par rapport à la borne supérieure n'est pas une bonne solution car elle risque de diminuer très fortement les performances de simulation, sans pour autant apporter plus de précision.

Pour conclure ce paragraphe, nous mettrons en évidence le fait que le temps consommé ici ne représente en rien une estimation (même grossière) du temps d'exécution réel, mais il n'est que le moyen de permettre à l'application logicielle native de s'exécuter de manière fonctionellement correcte. Ceci nous mène à une seconde limitation de cette approche.

Limitation 4.2 Validation fonctionnelle du logiciel

Même si la quantité de code pouvant être exécutée nativement est importante (du système d'exploitation à l'application), sans aucune estimation du temps d'exécution du logiciel, cette solution n'autorise qu'une validation fonctionnelle. Les traitements fortement dépendants du temps tels que l'estimation du taux de charge des processeurs souvent réalisés dans les systèmes d'exploitation donneront nécessairement des résultats faux.

4.2.2 Partie matérielle d'une EU

Une EU est un modèle abstrait de processeur aussi bien du point de vue de son interface avec le matériel que de son architecture interne. La première question que l'on peut alors se poser concernant leur modélisation est :

Quelles parties de l'architecture interne et de l'interface du processeur cible faut-il et peut-on modéliser dans les EU ?

4.2.2.1 Modélisation de l'interface matérielle du processeur cible

Dans le cas de l'interface du processeur, la réponse dans un premier temps est très simple puisque sans annotations spécifiques du logiciel embarqué, il nous est impossible de prendre en compte les accès mémoire implicites (instructions et données) lors de son exécution native.

Ainsi, seuls les accès mémoire explicitement réalisés par l'intermédiaire de l'API HAL peuvent être pris en compte, c'est le cas de la fonction `__hal_read()` donnée dans le listing 4.2. Pour autant, nous ne recommandons pas aux développeurs logiciel d'utiliser ce type de fonction pour accéder aux données d'un tableau en mémoire par exemple, afin que ceux-ci soient visibles sur la plateforme de simulation. Cela ne représenterait qu'une partie des accès réellement effectués, les autres étant les accès aux variables scalaires générées par le compilateur, et n'apporterait donc aucune information exploitable lors de la simulation. Par ailleurs, un des principaux objectifs est de conserver le code source du logiciel d'origine pour la simulation native et d'imposer un minimum de contraintes aux développeurs logiciel. Donc, nous recommandons de n'utiliser l'API HAL seulement pour ce qu'elle a été définie, à savoir abstraire le matériel.

De la même manière qu'une EU implémente une API HAL spécifique en ce qui concerne la partie logicielle, l'interface matérielle d'une EU devrait de manière rigoureuse ne modéliser qu'un seul type de processeur (Harvard vs Von Neumann par exemple).

D'un point de vue matériel cependant, une seule implémentation d'EU permet de couvrir plusieurs types de processeur. Ceci est possible en raison de la modélisation au niveau TLM de l'interface du processeur dont l'abstraction permet de simplifier et donc de regrouper plusieurs modèles de processeur sous une même implémentation matérielle d'EU.

Dans une première approche, une Unité d'Exécution composée d'une seule interface d'accès en lecture ou écriture à la mémoire modélisée dans la plateforme matérielle permet d'offrir un modèle très simple de processeur, suffisant pour permettre l'exécution native du logiciel. Si l'on ajoute à cela des entrées d'interruptions, on obtient une EU permettant de supporter des applications beaucoup plus réalistes.

Un exemple d'une telle interface, composée d'un port TLM maître `I/O` et des deux ports TLM esclaves d'interruption `Reset` et `IT` est représentée sur la figure 4.3.

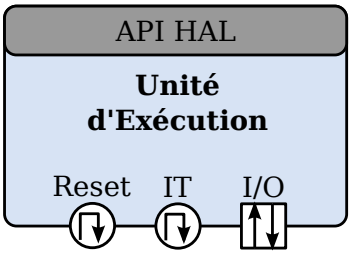


FIGURE 4.3 – Interface matérielle basique d'une EU

4.2.2.2 Modélisation de l'architecture interne du processeur cible

Étant donné le fait que le logiciel s'exécute uniquement sur le processeur hôte, celui-ci utilise inévitablement et de manière implicite toutes les ressources matérielles de ce dernier (*pipeline*, registres, unités arithmétiques, mémoires caches, etc.). Parmi toutes ces ressources, seules celles explicitement utilisées par le développeur du logiciel au travers de l'API HAL doivent être modélisées dans les EU afin de simuler leur comportement du point de vue des couches logicielles supérieures. Les autres ressources sur lesquelles le développeur ne peut exercer aucun contrôle, telles que les unités arithmétiques, le *pipeline* et bien d'autres encore, ne seront donc pas représentées dans les EU.

Comme pour l'implémentation de la couche HAL présentée précédemment, il n'existe

là non plus aucune règle précise pour l'implémentation de la partie matérielle dans une EU. Celle-ci doit être réalisée en fonction des besoins de l'API HAL utilisée et de la capacité à modéliser le matériel concerné.

Modélisation des mémoires caches et gestionnaires de mémoire virtuelle :

Parmi les ressources de l'architecture interne des processeurs utilisées dans les systèmes embarqués et devant êtres modélisées, on trouvera très fréquemment des mémoires caches. Certains systèmes d'exploitation évolués (maintenant utilisés dans les systèmes embarqués) peuvent exploiter certaines fonctionnalités de ces mémoires dans un souci de performances (chargements anticipés, remise à zéro, ...). Les mémoires caches correspondent au type de ressource qu'il n'est pas toujours possible de modéliser dans une EU. Théoriquement, la prise en compte de ces mémoires dans les EU devrait affecter le temps consommé par le logiciel exécuté nativement (en modifiant la valeur de $Tps_{synchro}$ étudiée dans la section précédente 4.2.1.2) de manière à modéliser l'effet de celles-ci sur les temps d'accès aux instructions et aux données.

Pour les mêmes raisons qui nous ont poussés à choisir une valeur constante pour le temps de synchronisation $Tps_{synchro}$, la modélisation du comportement des mémoires caches n'est pas possible dans les EU sans instrumentation du code logiciel. Nous avons considéré arbitrairement dans cette description que les mémoires caches font partie de l'architecture interne du processeur. La situation reste identique lorsque celles-ci sont considérées externes au processeur puisque la source du problème vient de l'absence d'informations sur les accès à la mémoire réalisés lors de l'exécution du logiciel.

Dans tous les cas, les différentes fonctions de l'API HAL dédiées à la gestion des mémoires caches devront tout de même être fournies par l'EU et leurs implémentations faites de telle sorte que l'exécution du logiciel n'en soit pas affecté.

D'une manière générale, toutes les ressources internes manipulant les mémoires utilisées par le logiciel ne peuvent pas être modélisées sans des annotations spécifiques du code logiciel. C'est le cas par exemple des unités de gestion des mémoires virtuelles (MMU) utilisées par les systèmes d'exploitation multi-tâches.

Modélisation des mécanismes de gestion d'interruptions :

L'utilisation d'interruptions dans les systèmes à base de processeurs est indispensable et les mécanismes de gestion d'interruptions doivent évidement êtres modélisés dans les EU. Il ne s'agit pas ici des contrôleurs d'interruptions externes au processeur (qui seront modélisés comme tous les périphériques dans le modèle TLM de la plateforme matérielle), mais bien du mécanisme de gestion des interruptions local au processeur.

Sur un processeur cible, les interruptions sont prises en compte entre chaque exécution d'instruction. En simulation native, la prise en compte des interruptions ne peut se faire que lors des appels aux fonctions de l'API HAL. La totalité du code logiciel situé entre deux appels à une fonction du HAL étant exécutée de manière atomique, cette prise en compte ne doit se faire qu'après l'appel à la fonction `wait()` par lequel débute chaque fonction du HAL. En effet, le temps mis par le logiciel à s'exécuter doit impérativement être entièrement consommé du point de vue de la simulation avant de traiter les interruptions afin de conserver une cohérence temporelle.

En dehors du fait que la granularité de prise en compte des interruptions sera beaucoup moins fine que sur un processeur réel, son principe de fonctionnement reste très similaire.

La gestion des interruptions étant essentielle, un exemple d'implémentation d'un tel mécanisme est donné dans la suite. Il servira également de modèle à l'implémentation des mécanismes matériels dans les EU.

Considérons que l'on veuille modéliser un processeur disposant des deux entrées `Reset` et `IT` pouvant interrompre l'exécution du logiciel. Ces entrées sont modélisées par des ports esclaves TLM (les ports `Reset` et `IT` sur la figure 4.4) permettant de mettre à un ou à zéro les registres internes `Reset_flag` et `IT_flag`. Un registre `IT_Enable` permet d'autoriser les interruptions sur l'entrée `IT` (l'entrée `Reset` n'étant pas masquable). L'API HAL offre de son côté deux fonctions `hal_enable()` et `hal_disable()` permettant au système d'exploitation exécuté au dessus de celle-ci d'activer ou non les interruptions en modifiant la valeur du registre `IT_Enable` comme cela serait fait sur un processeur réel.

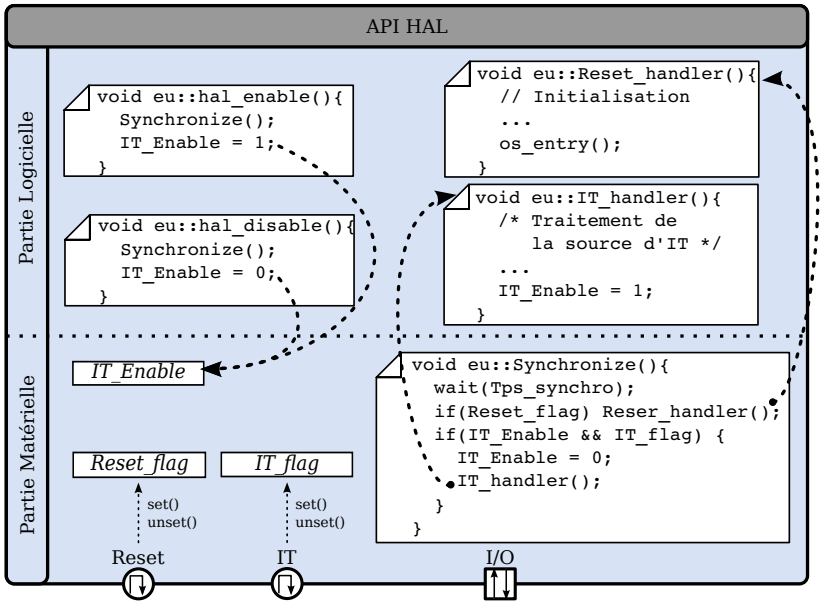


FIGURE 4.4 – Implémentation d'un mécanisme de gestion d'interruptions dans une EU

L'appel à la fonction `wait()` obligatoire en début de chaque fonction du HAL est remplacé ici par un appel à une fonction de synchronisation (`Synchronize()`) qui se chargera elle même de consommer le temps de simulation et qui implémentera en plus le mécanisme de gestion des interruptions. Une implémentation simplifiée de la fonction `Synchronize()` est donnée dans la figure 4.4. La prise en compte des interruptions consiste alors à appeler les routines d'interruptions `Reset_handler()` et `IT_handler()` en fonction des valeurs des différents registres concernés.

Sur un processeur réel, les adresses de ces routines sont contenues dans la table des vecteurs d'interruptions. Généralement, l'implémentation consiste à déterminer la source d'interruption (en interrogeant un contrôleur externe par exemple) puis d'appeler la sous-routine de traitement d'interruption correspondante. L'adresse de cette sous-routine aura bien entendu été préalablement configurée par le logiciel, à l'aide de fonctions HAL permettant d'affecter une adresse de fonction à chaque source d'interruptions.

Sur une EU les adresses des routines d'interruptions n'ont pas à être placées dans une table de vecteurs puisqu'elles sont accessibles directement dans l'implémentation abstraite du matériel. Cependant, le principe d'appel de celles-ci lorsqu'une interruption est dé-

clenchée reste similaire, de même que le traitement permettant de déterminer la source d'interruption jusqu'à l'appel à la sous-routine associée à cette source.

4.3 Représentation uniforme de la mémoire

Le deuxième concept de la solution proposée repose sur une représentation mémoire uniforme, partagée entre les points de vue logiciel et matériel, construite dynamiquement à l'initialisation de la simulation. Ce plan mémoire n'étant pas connu par avance, il s'en suit une phase d'édition dynamique de liens permettant de résoudre les valeurs des adresses mémoires utilisées par le logiciel.

L'idée consiste à utiliser le plan mémoire de l'exécutable SystemC (le plan mémoire hôte, ② sur la figure 4.1) comme unique représentation entre le logiciel et le matériel. Ce choix a été motivé par des raisons techniques :

- Une uniformisation « artificielle » en utilisant des techniques de conversion entre les espaces mémoire n'est pas applicable directement et impose des règles d'écriture de code contraignantes (exemple du DMA et des pointeurs C dans la problématique).
- Une uniformisation du plan mémoire utilisé par le logiciel vers le plan mémoire simulé n'est pas possible. Les adresses utilisées seraient invalides pour le système hôte, qui prendrait ces accès pour des erreurs de segmentation [2].

Mais aussi par des raisons pratiques :

- Le plan mémoire de la machine hôte est celui déjà utilisé par le logiciel exécuté nativement. Cela permet donc d'utiliser les outils de compilation et de débogage standard pour le logiciel, sans aucune contrainte particulière.
- Il est beaucoup plus aisé d'adapter le plan mémoire du modèle de simulation à celui de la machine hôte (qui est d'ores et déjà un plan mémoire construit), plutôt que d'adapter le logiciel au plan mémoire simulé.

Dans une plateforme réelle, le plan mémoire utilisé par le logiciel est uniquement lié aux adresses des composants de la plateforme (registres des périphériques, mémoires, etc.). Sur la plateforme native, le plan mémoire uniforme est toujours dépendant explicitement des zones occupées par les périphériques, mais également des zones mémoire occupées par la bibliothèque dynamique contenant le logiciel. Le plan mémoire final doit donc être construit à partir des adresses des périphériques et des zones mémoires de l'application.

4.3.1 Adresses des périphériques matériels

Partant du principe que chacun des composants matériels SystemC de la plateforme modélisent tous leurs registres ou zones mémoires internes, il existe alors pour chacune des adresses dans l'espace mémoire simulé ① de la figure 4.5, une adresse équivalente dans le plan mémoire ② correspondant à la zone réellement allouée sur la machine hôte.

Dans la figure 4.1, l'adresse de base (simulée) du segment mémoire occupé par l'ADC se trouve à l'adresse `0xA0001000` et occupe 4*ko* de mémoire. Si l'on considère maintenant l'implémentation SystemC de ce composant donnée dans le listing 4.4, la zone mémoire occupée par l'ADC est modélisée par un tableau de 1024 mots de 32*bits* à la ligne 14.

L'adresse réelle de ce tableau en mémoire, connue uniquement lors de l'exécution de la simulation SystemC, sera considérée comme étant l'adresse de base du périphérique ADC.

2. Une erreur de segmentation (*segmentation fault*) est une exception levée lorsqu'une application tente d'accéder à un emplacement mémoire qui ne lui était pas alloué

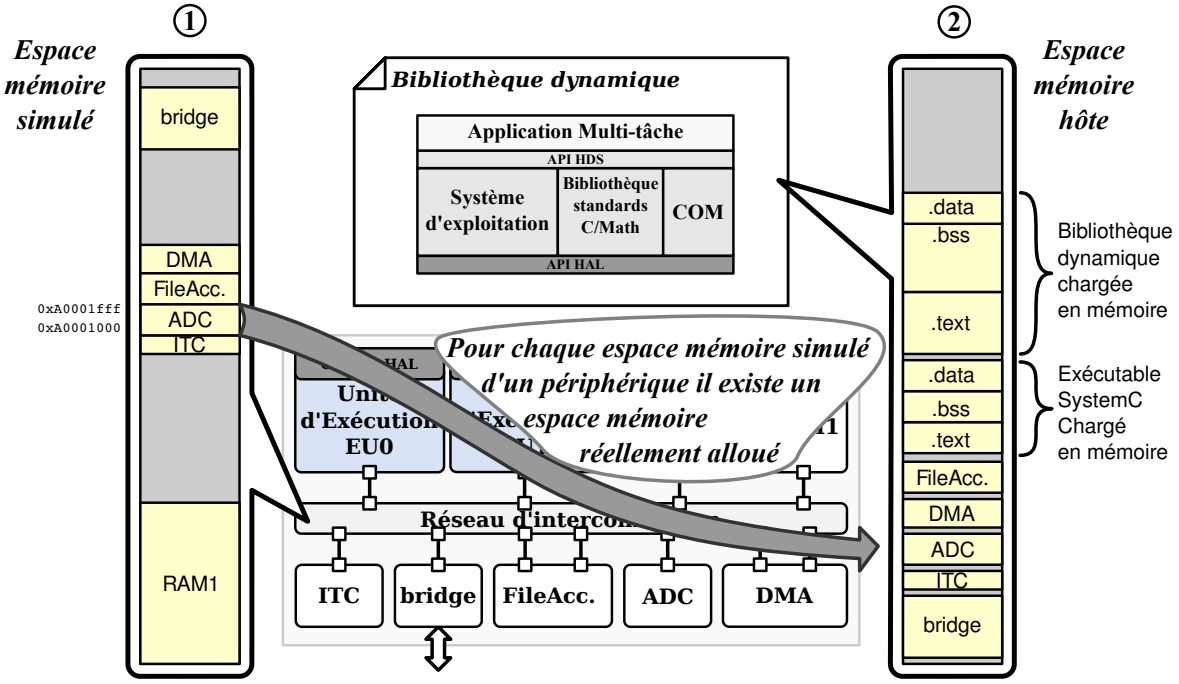


FIGURE 4.5 – Espaces mémoire simulés et réellement alloués des périphériques

Listing 4.4 Définition du composant ADC en SystemC

```
1  SC_MODULE(ADC):
2    public IO
3  {
4    public:
5      sc_export < IO >   exp_io;   /* Interface esclave */
6
7      void slv_read (uint32_t *addr, uint32_t *data);
8      void slv_write (uint32_t *addr, uint32_t data);
9
10     ADC(sc_module_name name);
11     ~ADC();
12
13   private:
14     uint32_t       _registers[1024]; /* Allocation des registres */
15 };
```

Les composants matériels de la plateforme sont les mieux à même de fournir les adresses qu'ils occupent en mémoire. Ainsi, chaque composant devra être capable de fournir une liste de description des espaces mémoires qu'il occupe. Cette description permettra la construction dynamique du plan mémoire final utilisé par le réseau de communication de la plateforme et contiendra typiquement les informations suivantes :

- Un nom permettant l'identification de la zone mémoire.
- L'adresse de base dans l'espace mémoire de la machine hôte.
- La taille mémoire occupée.

D'autres informations pourront éventuellement s'ajouter à cette description, comme les droits d'accès en lecture, écriture ou exécution.

Nous pouvons d'ores et déjà définir deux règles d'implémentation des composants de la plateforme matériel afin de permettre l'exécution native du logiciel.

Règle 4.1 Implémentation des composants matériels

Chaque composant de la plateforme (y compris le réseau de communication s'il occupe une zone mémoire) doit :
- Allouer l'espace mémoire susceptible d'être accédé à partir du logiciel et ce en respectant les caractéristiques du plan mémoire réel du composant.
- Permettre l'accès à la description des zones mémoire allouées.

Règle 4.2 Implémentation des réseaux de communication

Le réseau de communication doit construire sa table de décodage après l'initialisation de la plateforme, à partir de la description des zones mémoire allouées par les périphériques esclaves accessibles par ses ports d'entrées sorties.

4.3.2 Zones mémoire de l'application

Le plan mémoire construit à partir des zones occupées par les composants de la plateforme n'est cependant pas complet. Les différentes zones mémoire vues par l'application logicielle doivent également être accessibles par les périphériques matériels (tel que le DMA) au travers des composants modélisant la mémoire.

Ces zones de mémoire, appelées segment, sont celles obtenues lors de la compilation native du logiciel telles que la zone `.text` contenant le code exécutable de l'application, la zone `.bss` contenant les données initialisées à zero (explicitement ou implicitement) ou encore la zone `.data` pour les données initialisées à des valeurs différentes de zéro.

Afin de faciliter la répartition des segments sur les mémoires de la plateforme, nous nous reposons sur l'utilisation d'un composant spécifique (non SystemC) appelé *Linker* (figure 4.6).

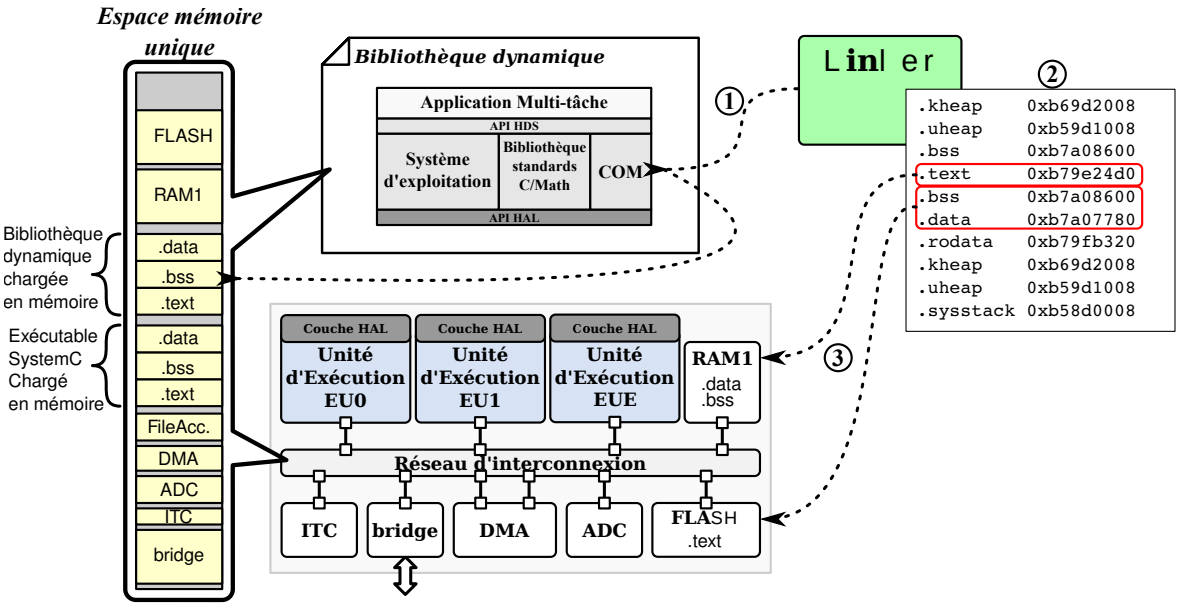


FIGURE 4.6 – Plateforme native avec représentation de la mémoire unifiée

A l'initialisation de la simulation, ce *Linker* est responsable du chargement dynamique de l'application contenue dans la bibliothèque ①. Il s'agit là, d'un chargement classique de bibliothèque dynamique, consistant à charger le programme en mémoire, et à résoudre les liens des symboles non définis [Lev99]. C'est d'ailleurs durant cette phase que les adresses des fonctions du HAL seront résolues. Un des rôles du *Linker* est de mettre à disposition de la plateforme la liste des segments de l'application ②, définis de la même manière que les zones de mémoire des périphériques.

Dans la figure 4.6 ③, la zone `.text` est placée dans la mémoire `FLASH` de la plateforme. A l'initialisation, celle-ci interrogera donc le *Linker* afin d'obtenir la description de la zone `.text` (adresse de base, taille, ...). La mémoire `RAM1` contient quant à elle les segments `.bss` et `.data` obtenus également du *Linker*. Les mémoires devront également allouer l'espace nécessaire afin de modéliser correctement la taille qui leur aura été affectée. C'est à dire, la taille de la mémoire moins la taille des segments qu'elles contiennent.

Règle 4.3 Implémentation des mémoires

Les mémoires de la plateforme contenant des segments logiciels (`.text`, `.bss`, etc.) doivent les rendre accessibles au travers du réseau de communication. Les autres mémoires modélisent l'espace alloué comme des composants classiques (méthodologie 4.1). Toutes les mémoires doivent fournir une liste de description des zones allouées qui seront nécessaires pour le plan mémoire du réseau de communication.

4.3.3 Construction dynamique du plan mémoire

Le plan mémoire finalement utilisé est construit par le réseau d'interconnexion lui même, avant que la simulation ne débute, et après l'initialisation de tous les composants de la plateforme.

Ce plan mémoire est défini par l'ensemble de toutes les zones accessibles par le réseau d'interconnexion et donc par tous les composants connectés sur les ports esclaves de celui-ci. Il suffit donc d'interroger chacun des composants accessibles au travers des ports esclaves afin d'obtenir la liste des descriptions des zones mémoire. Le réseau peut ensuite construire la table de décodage d'adresses utilisée pour identifier le port esclave concerné par un accès. La table de décodage obtenue pour l'exemple de plateforme utilisée jusqu'à présent est représentée sur la figure 4.7.

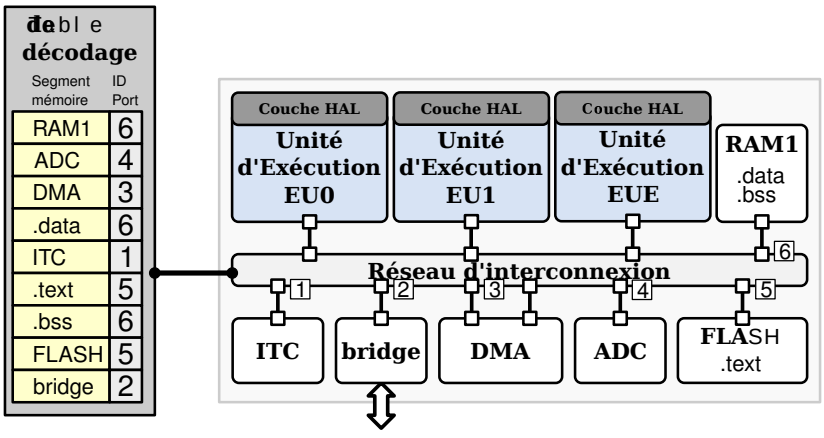


FIGURE 4.7 – Construction du plan mémoire utilisé par le réseau de communication

Ce plan mémoire est équivalent au plan mémoire réel en ce qui concerne le nombre et la taille des différentes zones occupées par les périphériques. Seules les adresses de base sont différentes. La cohérence mémoire est quant à elle assurée par la machine hôte.

Une exception concerne cependant les mémoires dans lesquelles sont placés des segments logiciels. En effet, une mémoire contigüe dans la plateforme réelle ne le sera plus nécessairement dans le plan mémoire construit dynamiquement. C'est le cas par exemple de la mémoire `RAM1`, qui se trouve segmentée en trois parties dans la table de décodage de la figure 4.7. Ces trois segments sont les zones `.bss`, `.data` et la zone `RAM1` correspondant au reste de l'espace disponible dans cette mémoire.

Ce plan mémoire segmenté sera cependant bien accessible par un seul port ce qui permettra de modéliser correctement les accès concurrents à la mémoire concernée et donc les problèmes de congestion dans le réseau de communication. Il est également important de noter que la taille de certains segments logiciels (tel que le segment `.text`) sera différente entre la plateforme native et la plateforme réelle si le processeur cible est différent du processeur hôte (ce qui est fort probable).

L'uniformisation de la mémoire de la plateforme vers celle de la machine hôte permet l'exécution directe du logiciel compilé nativement *à priori* sans contrainte particulière. L'accès aux zones mémoire peut maintenant être fait indifféremment par le logiciel, comme par le matériel.

4.4 Édition dynamique de liens

Il est nécessaire dans le logiciel (généralement « *bas niveau* ») de connaître certaines adresses physiques de la plateforme matérielle. C'est le cas par exemple dans les pilotes de périphériques pour lesquels il faut fournir les adresses permettant d'accéder aux registres dans le plan mémoire, typiquement, l'adresse de base de ces registres. Il n'est pas rare dans le monde du logiciel embarqué de voir ces adresses directement codées « *en dur* », comme dans le listing 4.5 où l'adresse d'un convertisseur analogique/numérique est directement affectée à un pointeur (lignes 8).

Listing 4.5 Exemple de plan mémoire défini « *en dur* »

```
1  typedef struct {
2    uint32_t  ADCCTL;     /* ADC control register */
3    uint32_t  ADCDATA;    /* ADC data register */
4    uint32_t  ADCCLKDIV; /* ADC clock divider register */
5    uint32_t  ADCCLKCTL; /* ADC clock control register */
6  } adc_registers;
7
8  volatile adc_registers  *adc_base = (adc_registers*) 0xA0001000;
```

Dans certains cas, les valeurs des adresses en mémoire peuvent être utilisées de manière encore plus directe :

```
#define adc_base   ((volatile adc_registers*)0xA0001000)
adc_base->ADCCTL = 0x00F1E;
```

Contraintes 4.1 Codage des adresses « *en dur* »

L'utilisation d'adresses « *en dur* » n'est pas supportée par la simulation native car les adresses des périphériques dans le processus de simulation ne sont pas connues au moment de la compilation du code.

La résolution de ce type d'adresses doit donc être reportée à l'initialisation de la plateforme de simulation. Cette phase de résolution d'adresses peut être vue comme une édition dynamique de liens entre le logiciel et le matériel qui sera effectuée par le *Linker* après la construction du plan mémoire de la plateforme.

Dans la solution proposée, toutes les adresses absolues liées directement à la plateforme matérielle (et plus précisément à son plan mémoire) doivent être définies par des pointeurs comme dans le listing 4.5 (qui peuvent quand même être initialisées en fonction des adresses de la plateforme finale). Les valeurs des adresses seront mises à jour lors de l'initialisation de la plateforme native. Un support de la part des modèles de périphériques est requis pour assister le *Linker* dans cette opération. Chaque périphérique a la charge de définir les symboles (couple *{nom,valeur}*) attendus par le pilote logiciel associé. Ainsi, le modèle du périphérique ADC dans l'exemple de plateforme native donné figure 4.8 devra créer le symbole attendu par le pilote logiciel correspondant (listing 4.5) nommé `adc_base` et lui affecter la valeur de l'adresse de base de ces registres ①.

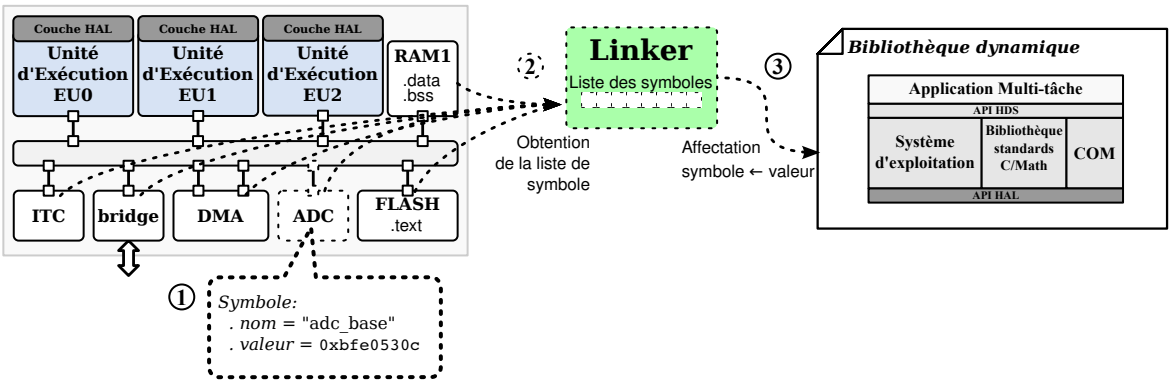


FIGURE 4.8 – Mécanisme d'édition dynamique de liens

Le *Linker* peut ensuite récupérer les listes de tous les symboles définis par les périphériques de la plateforme. Pour cela, il interroge chacun des composants qui lui seront connectés de la même manière que des ports TLM ②. Ces connections ne seront utilisées que lors de l'élaboration de la plateforme de simulation. Il pourra ensuite retrouver les symboles réels dans le code binaire du logiciel pour finalement leur affecter directement l'adresse résultant de l'instanciation des périphériques ③. Dans la figure 4.8, le pointeur `adc_base` aura donc la valeur `0xbfe0530c` après l'initialisation, et pourra être directement utilisé dans le logiciel puisque cette adresse est valide dans l'espace mémoire unifié.

Règle 4.4 Implémentation des composants matériels

Chaque composant de la plateforme doit :
– Décrire la liste des symboles attendus par le logiciel associé sous la forme {*nom*,*valeur*}
– Permettre l'accès à sa liste de symboles.

Règle 4.5 Implémentation des réseaux de communication

Les réseaux de communication doivent permettre de récupérer la liste de tous les symboles définis dans les composants connectés (typiquement pour la fournir ensuite au *Linker*).

Il existe une autre manière d'utiliser des adresses physiques directement dans le code du logiciel. Toute édition de liens d'un logiciel est contrôlée par un fichier de script (*linker*

script). Il ne s'agit pas ici de l'édition dynamique de liens présentée précédemment en vue de lier le code logiciel natif avec la plateforme matérielle, mais bien de l'édition de lien utilisée pour la génération du code binaire final sur la cible. Le principal objectif de ce script est de décrire comment les différentes sections des fichiers objets du logiciel doivent être placées en mémoire, et de contrôler l'organisation des segments dans cette mémoire.

L'édition de liens définit également un certain nombre de symboles supplémentaires permettant d'identifier les adresses de début et de fin des segments en mémoire. On trouvera typiquement dans un environnement Unix les symboles `_etext` et `_edata` délimitant la zone `.data` ou encore `__bss_start` et `_end` pour la zone `.bss`. Contrairement à la technique présentée précédemment, la valeur de ces symboles ne pointe pas l'adresse physique en mémoire, mais l'adresse physique considérée est celle du symbole lui même (`&_edata` donne l'adresse de début du segment `.data`). La figure 4.9(a) représente un espace mémoire où les adresses de symboles donnent directement l'adresse physique en mémoire. On peut voir la différence avec l'utilisation de pointeur, le symbole `adc_base` utilisé précédemment est un symbole dont l'adresse correspond au mot mémoire contenant la valeur de l'adresse physique des registres en mémoire. La valeur de ce dernier est donc modifiable à l'exécution alors qu'elle ne l'est pas pour les symboles définis pendant l'édition de liens. Les adresses de ces symboles sont généralement utilisées par les systèmes d'exploitation, par exemple pour initialiser la zone `.bss` à zéro.

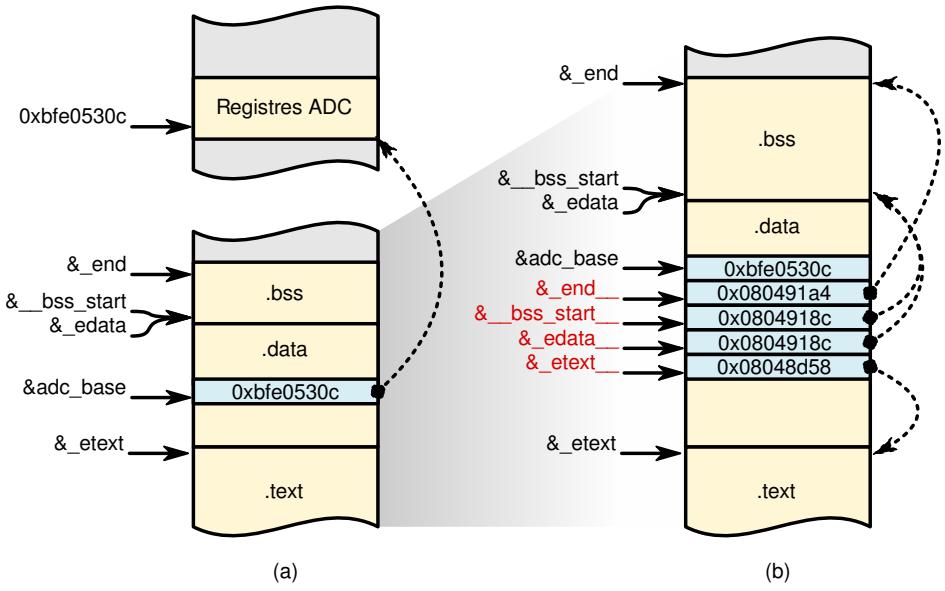


FIGURE 4.9 – (a) Symboles créés lors de l'édition de liens classique et (b) utilisation de pointeurs à la place des symboles par définition de nouveaux symboles

Contraintes 4.2 Symboles définis à l'édition de liens

Il n'est pas possible d'utiliser directement des symboles définis pendant la phase d'édition de liens dans le code logiciel car il n'est pas possible de modifier leurs adresses en mémoire lors de l'exécution.

Une solution pour palier ce problème est d'utiliser des pointeurs comme pour les pilotes de périphériques afin que le *Linker* puisse initialiser leurs valeurs. La figure 4.9(b) illustre l'utilisation de pointeurs à la place de symboles pour l'identification des zones mémoire. Par exemple, pour obtenir l'adresse de débute de la zone `.bss`, on ne pourra plus utilisée

`&__bss_start`, mais dans ce cas il faudra utiliser `__bss_start`, ce dernier étant définit comme un pointeur. On peut alors se poser la question suivante : si le *Linker* est responsable de l'initialisation dynamique de ces symboles à l'exécution, qui est responsable de leurs déclarations ?

Dans le cas des symboles attendus par les pilotes logiciels, nous avons fait le choix de faire définir ces symboles par les périphériques associés, étant donné le lien étroit qui existe entre eux.

Pour les autres symboles attendus par le logiciel (et en particulier par le système d'exploitation) on peut faire l'analogie avec un environnement « classique » où c'est l'éditeur de liens et plus particulièrement le script de commande qui définit ces symboles. Nous garderons cette analogie dans l'approche proposée. Ainsi le *Linker* est un composant (au sens logiciel du terme) spécifique au système d'exploitation devant s'exécuter nativement sur la plateforme. Il découle finalement de ces dernières contraintes une règle de codage concernant cette fois-ci la partie logicielle.

Règle 4.6 Implémentation de la partie logicielle

L'accès à des adresses physiques en mémoire à partir du logiciel ne devra se faire que par l'utilisation de pointeurs, aussi bien dans les pilotes de périphériques que dans le système d'exploitation.

4.5 Conclusion

Nous avons présenté dans ce chapitre une méthode de conception de plateforme de simulation d'architecture MPSoC basée sur les deux concepts fondamentaux que sont l'utilisation des EU comme uniques interfaces entre le logiciel et le matériel et l'utilisation d'une représentation mémoire uniforme entre le modèle matériel de l'architecture et le logiciel exécuté nativement.

Dans cette approche, mis à part les EU implémentant la couche HAL, SystemC est utilisé uniquement pour la modélisation du matériel. Le logiciel est un programme exécutable contenant toutes les couches de la pile logicielle (application, bibliothèques, *middleware* et système d'exploitation) liées ensembles. Les EU jouant le rôle de processeurs démarrent le logiciel comme le feraient des processeurs classiques, permettant aux tâches de l'application de s'exécuter sur un environnement logiciel et matériel réaliste puisque l'ensemble du système d'exploitation et des bibliothèques est exécuté avec une plateforme fidèle à l'architecture matérielle finale. La conjonction de ces deux concepts permet notamment d'utiliser plusieurs EU afin de modéliser des architectures de type SMP, sur lesquelles la migration des tâches entre des processeurs identiques est possible sous contrôle du système d'exploitation.

C'est une différence fondamentale avec les approches de co-simulation récentes basées sur l'exécution native du logiciel [CHB09a] dans lesquelles les tâches de l'application et les services du système d'exploitation sont directement encapsulés dans des modules SystemC, qui souvent n'autorisent qu'une seule tâche par module jouant le rôle de processeur. Notre approche est la seule à l'heure actuelle permettant de modéliser nativement tout type de plateforme.

Nous avons également mis en évidence un certain nombre de limitations à cette approche, la première étant liée à l'utilisation d'une couche HAL hermétique. Les applications ou bibliothèques contenant du code assembleur (optimisation d'algorithme de calcul)

ou des accès à la mémoire sans passer par le HAL ne sont pas supportés par cette solution. L'ensemble du code source doit également être disponible afin d'être compilé pour le processeur hôte.

Certaines limitations, principalement liées à la manipulation d'adresses dans la plateforme matérielle (accès aux registres des périphériques, déclaration des segments de mémoire), ont été résolues par la proposition de méthodes de développement du code logiciel. Les contraintes de développement restent cependant acceptables car elles ne concernent qu'une faible quantité du logiciel, généralement situé dans les couches bas niveau (les pilotes de périphériques par exemple).

Concernant l'implémentation de la couche HAL dans les EU, nous avons également montré qu'il n'y a pas de garantie de faisabilité, chaque implémentation étant réalisée en fonction des capacités et des possibilités offertes par la machine hôte. La plupart des mécanismes complexes permettant la validation d'applications multi-tâches sur des architectures multiprocesseurs ont cependant pu être modélisées au cours de cette thèse. Elle seront présentées et utilisées pour les expérimentations.

Nous terminerons sur la principale limitation de cette plateforme liée à l'absence totale d'annotations dans le code logiciel exécuté nativement. Ainsi, et malgré le niveau de détails offerts par la plateforme matérielle, cette solution reste confinée à la simulation fonctionnelle de la pile logicielle.

Chapitre 5

Instrumentation automatique du logiciel embarqué

Sommaire

L'estimation des performances du logiciel s'exécutant sur une plateforme de simulation de MPSoC abstraite est un élément déterminant pour l'exploration d'architectures. Elle l'est également pour la validation du logiciel où la notion de temps est très présente, c'est-à-dire dans la plupart des domaines d'application des MPSoC. Cette estimation sur la plateforme de simulation passe nécessairement par l'instrumentation du code logiciel.

Le processus d'instrumentation présenté dans ce chapitre est clairement séparé du problème d'estimation des performances. L'instrumentation consiste à déterminer où et comment placer les annotations dans le code logiciel afin qu'il reflète au mieux l'exécution qu'il aurait eu sur un processeur cible au moment de sa simulation sur le processeur hôte. Une première solution d'estimation des performances consistera dans le chapitre suivant à déterminer quelles informations associer à ces annotations.

5.1 Concepts de base et principales difficultés

Comme toutes les approches de simulation native, l'objectif initial de la solution présentée dans le chapitre précédent était la simulation fonctionnelle du logiciel embarqué sur une plateforme MPSoC. Nous avons mis en évidence dans la section 4.2.1.2 la nécessité de consommer un temps d'exécution « *artificiel* » du logiciel (Appelé $Tps_{synchro}$) dont le but était uniquement de permettre à la simulation de fonctionner. La notion de temps dans le modèle de simulation proposé est donc déjà présente dans les Unités d'Exécution (EU). Le problème consiste alors à proposer une solution permettant de déterminer dynamiquement les valeurs de $Tps_{synchro}$ afin que ce temps ne soit plus une constante fonctionnelle mais une estimation du temps d'exécution réel du logiciel. Les EU n'ayant aucun contrôle sur le logiciel lorsqu'il s'exécute nativement, la seule solution est que ce soit le logiciel lui-même qui fournisse les informations sur son exécution. Une solution consiste à instrumenter le programme par des appels de fonctions qui permettront de fournir à la plateforme de simulation les informations sur le cours de son exécution.

Si l'on considère le code source en langage C donné dans le listing 5.1, l'instrumentation du logiciel consiste alors à insérer les annotations dans la version native de ce code, mais de manière à refléter son exécution sur le processeur cible.

Listing 5.1 Code source à instrumenter

```
1  x = (y != 0) ? 23 : 1234567;
```

La première difficulté inhérente à l'instrumentation de code est due à la variabilité d'exécution du programme lui-même en fonction des données. Une solution à la dépendance d'exécution aux données du programme est de suivre son graphe de flot de contrôle (CFG pour Control Flow Graph). En pratique, cela signifie que l'instrumentation doit être effectuée au niveau des *blocs de bases*.

Définition 5.1 Définition : CFG et *bloc de base*

Un CFG est une représentation sous forme de graphe de tous les chemins traversés par un programme au cours de son exécution.

Chaque nœud dans le graphe représente un bloc de base, c'est à dire une séquence de code ne contenant qu'un seul point d'entrée et un seul point de sortie et qui ne peut contenir aucune instruction de saut à l'exception de la dernière [ASU86].

Les arcs orientés représentent les sauts éventuels entre les blocs de base dans le flot de contrôle.

Dans une instrumentation au niveau du code objet du programme, les annotations sont directement insérées en début de chaque bloc de base. La figure 5.1(b) représente le CFG de la ligne de programme donnée dans le listing 5.1, compilée pour le processeur hôte *x86* et instrumentée. L'argument passé à la fonction `annotate()` représente l'identificateur du bloc de base correspondant dans le code objet du même programme compilé pour le processeur cible, un processeur *ARM* dans cet exemple. Le CFG de cette portion de programme compilée pour le processeur *ARM* est donné sur la figure 5.1(a).

Cette technique suppose que le CFG du code objet natif et celui du code objet cible sont isomorphes, ce qui n'est généralement pas le cas. Par exemple, la compilation du même programme avec un niveau d'optimisation plus élevé pour le processeur *ARM* produit un CFG complètement différent (figure 5.1(c)) de celui du processeur hôte (figure 5.1(b)), en tirant profit de l'instruction conditionnelle `movne`.

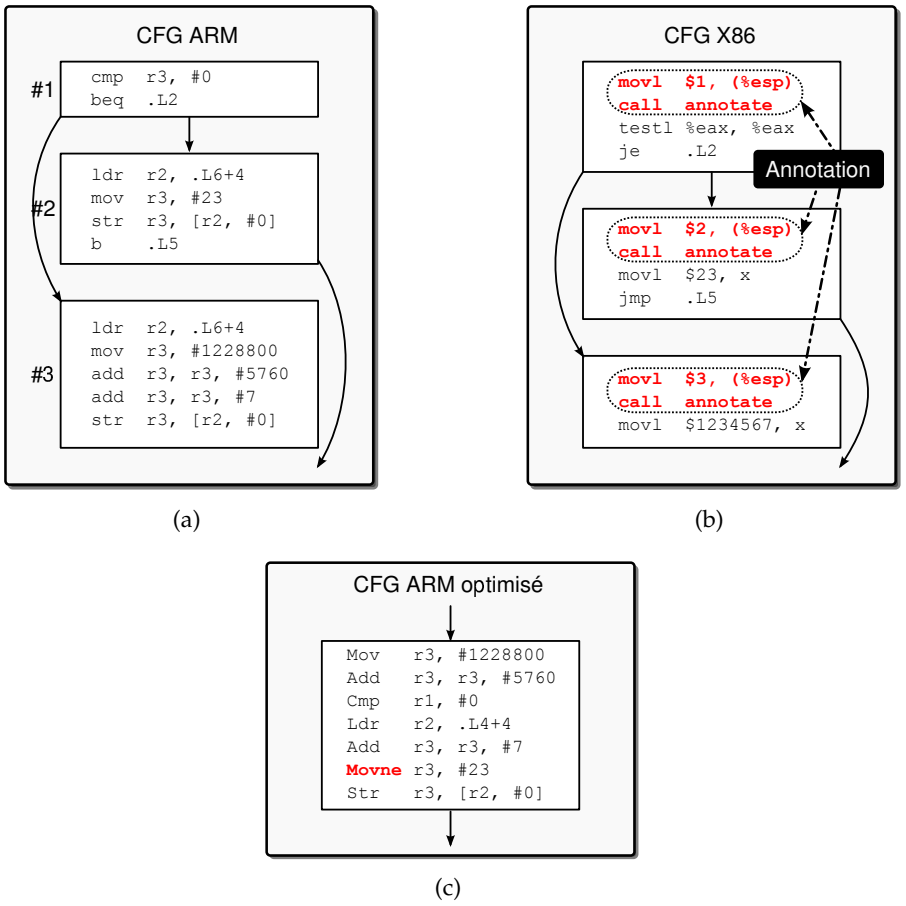


FIGURE 5.1 – Code objet pour les processeurs (a) *ARM*, (b) *x86* et (c) *ARM* avec optimisations

L'instrumentation du logiciel au niveau source permet de se défaire des dépendances au processeur hôte, en insérant les annotations directement dans le code source d'origine à partir des informations provenant de l'analyse du code objet cible. Le code source ainsi instrumenté est finalement compilé pour le processeur hôte. En pratique, cette méthode est complexe à implémenter. En effet, la recherche des frontières entre les blocs de base dans le code source est une tâche très difficile en raison des syntaxes riches et complexes des langages de programmation, combinés avec les effets d'optimisation des compilateurs, ce qui mène à une structuration du code objet qui n'a plus grand chose à voir avec le code source initial. L'exemple donné dans le listing 5.1 devrait être écrit de manière mieux adaptée à l'instrumentation du code source, comme dans le listing 5.2. De plus, la modification du code source du logiciel, surtout si elle est automatisée, peut présenter certains désavantages de lisibilité lors des phases de débogage.

Listing 5.2 Instrumentation au niveau du code source

```
1  annotate(1);
2  if(y != 0) {
3    annotate(2);
4    x = 23;
5  }
6  else {
7    x = 1234567;
8  }
```

5.2 L'instrumentation à la compilation

5.2.1 Principe de la solution proposée

La solution proposée pour l'instrumentation présente à la fois les avantages des méthodes d'instrumentation au niveau source et au niveau code objet du programme. L'idée principale pour automatiser l'instrumentation du code logiciel est d'utiliser la représentation intermédiaire (nommée IR dans la suite pour Intermediate Representation) du compilateur comme cible pour l'insertion des annotations dans les blocs de base du programme. Plusieurs avantages ont motivé ce choix :

- De la même manière que dans les approches d'instrumentation au niveau source, cela permet de rester indépendant de la machine hôte, à partir du moment où le processus d'instrumentation se trouve avant le *back-end* spécifique au processeur hôte.
- L'annotation reste également indépendante du langage de programmation utilisé.
- Les IR des compilateurs contiennent toutes les informations sur le CFG du programme qui permettront de refléter son exécution sur un processeur cible.

La figure 5.2(a) représente l'architecture simplifiée d'un compilateur. Après les différentes transformations et optimisations réalisées dans les phases de *front-end* et éventuellement *middle-end*, la IR est fournie en entrée du *back-end*. Dans un *back-end* conventionnel, la IR est généralement transformée en une autre représentation plus spécifique qui subira une série de transformations jusqu'à la génération du code objet final. A ce stade, le CFG de la IR spécifique utilisée par le *back-end* et celui de la IR initiale sont structurellement différents pour des raisons d'optimisation telles que celles évoquées dans la section précédente.

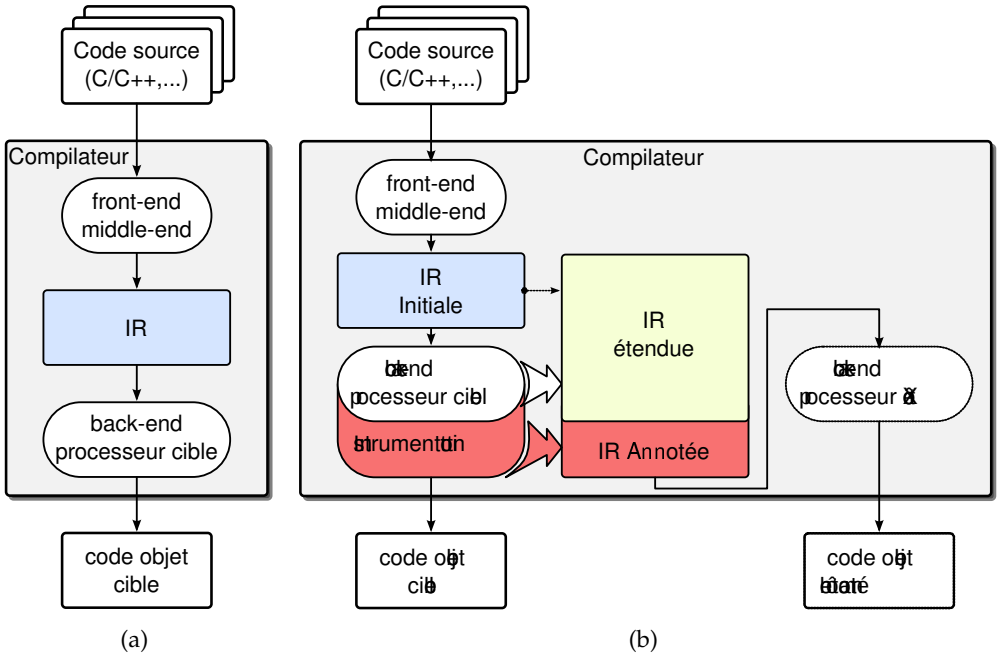


FIGURE 5.2 – Architecture (a) du compilateur d'origine et (b) du compilateur modifié

Afin de prendre en compte toutes les modifications effectuées sur le graphe de contrôle du programme, la IR est étendue tout au long de la phase de *back-end* spécifique au processeur cible, comme le montre la figure 5.2(b). Lorsqu'une passe d'optimisation du *back-end*

opère des transformations sur la structure du CFG, celle-ci devra répercuter ces modifications sur la IR étendue. Ainsi, une fois toutes les passes d'optimisation du *back-end* terminées, le CFG du programme pour le processeur cible et celui de la IR étendue sont isomorphes. L'instrumentation des blocs de base présentée précédemment pourra ainsi directement être appliquée.

Cette instrumentation est réalisée par une ultime passe de compilation ajoutée à la fin du *back-end* (figure 5.2(b)). Cette passe permet d'insérer les annotations relatives au programme *cross-compilé* pour le processeur cible, directement dans la IR étendue.

5.2.2 Construction de la IR étendue

La IR étendue est construite à partir de la IR initiale passée en entrée du *back-end* du processeur cible. Les transformations opérées sur la IR tout au long de ce *back-end* peuvent être représentées par des variations de sémantique entre des jeux d'instructions différents (nommés ISA pour Instruction Set Architecture).

Nous devons considérer ici trois jeux d'instructions différents. Celui du processeur cible ISA^{Cible}, celui du processeur hôte ISA^{Hote} et enfin celui de la représentation intermédiaire du compilateur ISA^{IR}. Ces jeux d'instructions sont représentés sous forme d'ensembles sur la figure 5.3 dont les intersections modélisent les instructions de même type.

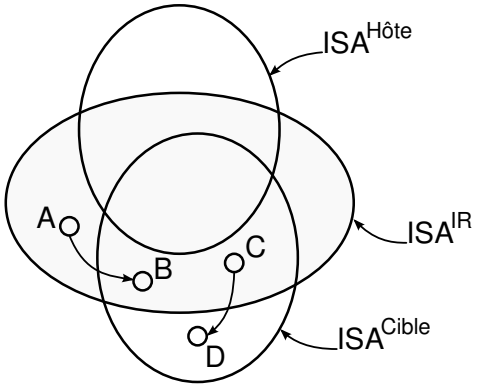


FIGURE 5.3 – Intersections des différentes sémantiques de jeux d'instructions

Plusieurs transformations seront effectuées sur les instructions du programme lors du *back-end* spécifique au processeur cible. La première de ces transformations consiste à convertir les instructions indépendantes du processeur appartenant à ISA^{IR} en instruction appartenant à ISA^{Cible}. Lors des phases d'optimisation, d'autres transformations seront ensuite effectuées, impliquant des conversions de ISA^{IR} vers ISA^{Cible}. On peut alors mettre en évidence trois situations de conversion possibles :

Situation 1 : Des instructions de ISA^{IR} appartenant à $ISA^{IR} \cap ISA^{Cible}$ sont converties en instructions de ISA^{Cible} appartenant également à $ISA^{IR} \cap ISA^{Cible}$.

Situation 2 : Des instructions de ISA^{IR} appartenant à $ISA^{IR} - ISA^{Cible}$ sont converties en instructions de ISA^{Cible} appartenant à $ISA^{IR} \cap ISA^{Cible}$. Correspondant au chemin $A \rightarrow B$ sur la figure 5.3.

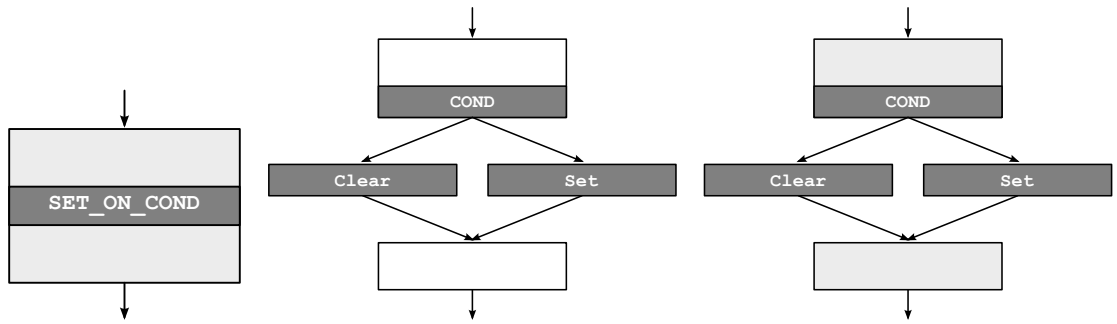
Situation 3 : Des instructions de ISA^{Cible} appartenant à $ISA^{IR} \cap ISA^{Cible}$ sont converties en instructions de ISA^{Cible} appartenant à $ISA^{Cible} - ISA^{IR}$. Correspondant au chemin $C \rightarrow D$ sur la figure 5.3.

Le premier cas correspond à une conversion d'instruction dont le type existe dans les deux représentations intermédiaires (celle utilisée par le *back-end* et celle indépendante du processeur cible). Ce cas ne pose pas de problème particulier car aucune transformation ne sera effectuée sur la structure du CFG cible. Les deux autres cas peuvent cependant amener à des modifications dans la structure du CFG cible (suppression, séparation, unification de blocs de base) qui doivent donc êtres prises en compte pour être répercutées sur le CFG de la IR étendue.

5.2.2.1 Situation n°2 : Conversion d'une instruction de ISA^{IR} n'ayant pas d'équivalent dans ISA^{Cible}

Cette situation ne peut intervenir que lors de la première passe du *back-end*, consistant à transformer les instructions utilisées dans la IR initiale en instructions spécifiques aux processeurs cible afin d'obtenir une IR dépendante de celui-ci, sur laquelle seront effectuées les optimisations. Même si cette phase n'effectue aucune optimisation sur le code du programme, des transformations de CFG peuvent tout de même être nécessaires lorsqu'il n'existe pas d'équivalence entre une instruction de ISA^{IR} dans le jeu d'instructions du processeur cible ISA^{Cible}.

C'est le cas par exemple si la ISA^{IR} possède l'instruction complexe « *Set-On-Condition* »[1]. Si cette instruction n'appartient pas au jeu ISA^{Cible}, celle-ci devra être transformée en un ensemble d'instructions plus simples, présentes dans l'ensemble ISA^{Cible}. Ce passage d'une instruction de ISA^{IR} vers ISA^{Cible}, modélisé par le chemin $A \rightarrow B$ sur la figure 5.3 peut nécessiter la transformation du CFG cible comme le montre la figure 5.4.



(a) CFG IR initiale (b) CFG IR cible (c) CFG IR étendue

FIGURE 5.4 – Transformation d'une instruction de ISA^{IR} en plusieurs instructions de ISA^{Cible}

Le bloc de base de la figure 5.4(a) est découpé en plusieurs blocs dans le CFG cible (figure 5.4(b)) réalisant la même opération que l'instruction « *Set-On-Condition* ». Ces transformations devront également être opérées sur le CFG de la IR étendue (figure 5.4(c)) afin de garantir l'isomorphisme des CFG. Cela sous-entend que les instructions utilisées dans le CFG de la IR cible pour traduire l'instruction complexe doivent également appartenir à l'ensemble ISA^{IR} (point B sur la figure 5.3).

1. Ce type d'instruction affecte la valeur un ou zéro à son opérande en fonction de la condition de test.

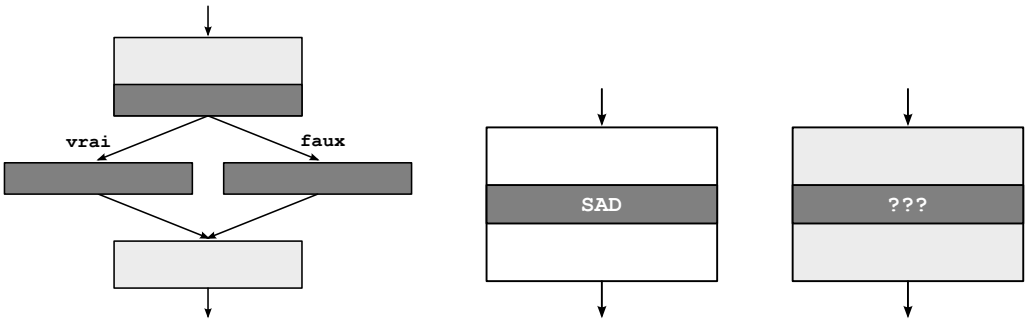
5.2.2.2 Situation n°3 : Transformation d'une instruction de ISA^{Cible} n'ayant pas d'équivalent dans ISA^{IR}

Des transformations de CFG sont également faites au cours des optimisations dépendantes du processeur cible. Deux cas de modifications peuvent alors être distingués :

- Les transformations d'optimisation n'introduisent pas d'instructions de l'ensemble ISA^{Cible} n'ayant pas d'équivalent dans l'ensemble ISA^{IR}.
- Les transformations d'optimisation tirent profit d'instructions spécifiques du processeur cible, n'ayant pas d'équivalent dans le jeu d'instructions de la représentation intermédiaire. Il correspond à la situation n°3 donnée précédemment.

La première situation peut être gérée en répercutant, lorsque cela est possible, les transformations du CFG cible sur le CFG de la IR étendu (découpage, suppression de blocs de base, etc.) comme nous l'avons présenté dans l'exemple précédent.

Le second cas, modélisé par le chemin $C \rightarrow D$ sur la figure 5.3, peut également aboutir à des transformations du CFG cible représentées sur la figure 5.5. Dans cet exemple, l'optimisation utilise l'instruction « *Sum-of-Absolute-Differences* » (SAD) disponible sur certains processeurs tels que les DSP. Plusieurs blocs de base du CFG initial (figure 5.5(a)) peuvent ainsi être regroupés en un seul bloc de base dans le CFG cible (figure 5.5(b)). Afin de respecter la contrainte d'isomorphisme sur les CFG cible et étendu, il faudrait en théorie répercuter les mêmes transformations sur le CFG de la IR, ce qui est à priori impossible puisque l'instruction équivalente n'existe pas dans l'ensemble ISA^{IR}.

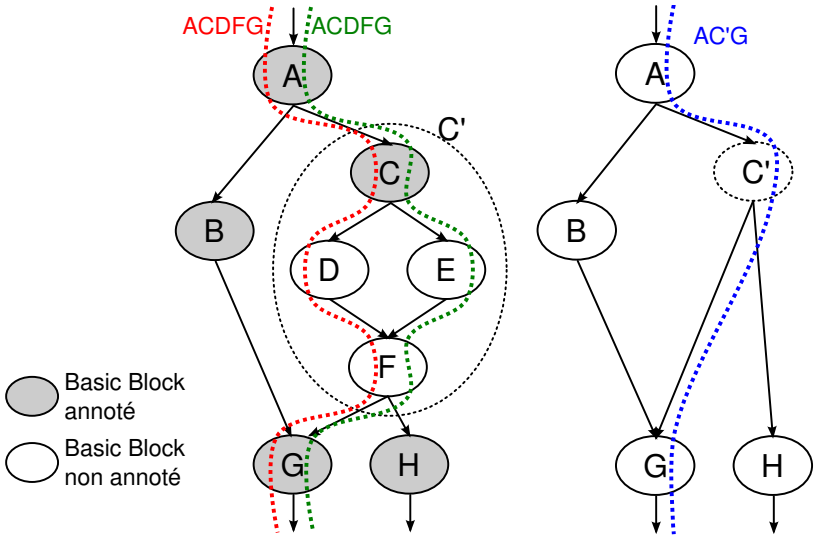


(a) CFG IR initiale (b) CFG IR cible (c) CFG IR étendue

FIGURE 5.5 – Transformation du CFG de la IR initiale en une instruction complexe du processeur cible n'ayant pas d'équivalence dans ISA^{IR}.

Si l'on regarde cependant le problème du point de vue de l'annotation (et non pas du point de vue de l'isomorphisme strict entre les CFG), on peut s'apercevoir très vite qu'un isomorphisme *strict* n'est pas nécessaire. En effet, cette condition ne porte finalement pas sur le CFG du programme effectivement exécuté en natif mais sur le CFG *apparent* décrit par les annotations. Cela signifie que sous certaines conditions, il est possible de ne pas annoter tous les blocs de base du CFG étendu afin de respecter cet isomorphisme apparent. Nous dirons alors que les deux CFG sont *équivalents*.

Dans le CFG de la figure 5.6(a), les blocs de base $CDEF$ ont été regroupés en un seul bloc C' dans le CFG cible (figure 5.6(b)). L'instrumentation devra dans ce cas insérer l'annotation relative au bloc C' uniquement dans le bloc C. Ainsi, au cours de l'exécution du programme natif, le chemin menant de A à G en passant par C sera toujours équivalent au chemin $AC'G$ dans le CFG cible, quelque soit le chemin d'exécution pris dans le sous CFG $CDEF$.



(a) CFG IR étendue (b) CFG IR cible

FIGURE 5.6 – Isomorphisme apparent du CFG de la IR étendue.

La construction de la représentation intermédiaire étendue consiste donc à répercuter les modifications des CFG uniquement lorsque cela est nécessaire. Cela signifie en pratique que chaque cas de modification effectuée au cours d'un *back-end* de processeur cible devra être analysé manuellement afin de déterminer s'il est nécessaire de le répercuter dans le CFG étendu.

5.2.3 Compilation de la IR étendue annotée pour le processeur hôte

La dernière étape consiste à faire compiler la IR étendue instrumentée par le *back-end* spécifique au processeur de la machine hôte. Cela revient donc à faire une nouvelle conversion de jeu d'instructions (dans la figure 5.3) de l'ensemble ISA^{IR} vers l'ensemble ISA^{Hote}.

Les appels de fonctions insérés lors de l'instrumentation au début de chaque bloc de base permettent cependant de nous prémunir contre les modifications qui auraient pour effet la modification du CFG finale. Le compilateur ne dispose pas du code des fonctions d'annotation et ne peut donc pas faire d'optimisations très poussées.

5.3 La passe d'instrumentation de code

Nous supposons ici que les deux CFG sont équivalents et que pour tout bloc de base d'un CFG, il est possible de trouver le bloc lui correspondant dans l'autre CFG. La passe d'instrumentation de code consiste à insérer les annotations dans chaque bloc de base du CFG de la IR étendu. Ceci permet de prendre en compte uniquement les blocs réellement exécutés au moment de la simulation de l'application.

La seule instrumentation des blocs de base du programme peut ne pas être suffisante pour garantir une bonne estimation des grandeurs physiques. En effet, le chemin emprunté dans le CFG lors de l'exécution du programme peut avoir un impact non négligeable en fonction du type d'estimation. Dans le cadre de l'estimation du temps d'exécution du logiciel, une étude réalisée à ce sujet dans [CHB09a] montre que les pénalités de branchement, qui pourront être annotées sur les arcs du CFG, peuvent représenter jusqu'à 30% du temps d'exécution du logiciel. L'annotation des arcs du CFG est donc nécessaire et sera également réalisée lors de la passe d'instrumentation.

5.3.1 Instrumentation des blocs de base

Pour chacun des blocs de base de la IR étendue, l'instrumentation se déroule en trois étapes :

❶ **L'analyse** statique permettant d'extraire les informations sur le bloc de base du CFG cible correspondant.

❷ **L'identification et le stockage** des annotations obtenues lors de la phase d'analyse permettront lors de l'exécution native du programme de retrouver toutes les informations nécessaires, relatives au bloc en cours d'exécution.

❸ **L'instrumentation** du bloc de base dans le CFG de la IR étendue telle qu'elle a été présentée précédemment peut être appliquée directement. L'appel à la fonction d'annotation, prenant comme argument l'identifiant obtenu précédemment est inséré au tout début du bloc.

Ces trois étapes sont représentées sur la figure 5.7. Le CFG de la IR cible est celui d'un programme compilé pour un processeur *ARM* et le CFG de la IR étendue correspond à celui d'un processeur *x86*. En pratique, les codes assembleurs contenus dans les blocs de base ne seraient pas ceux des processeurs *ARM* et *x86* mais les pseudos assembleurs utilisés dans les représentations intermédiaires du compilateur.

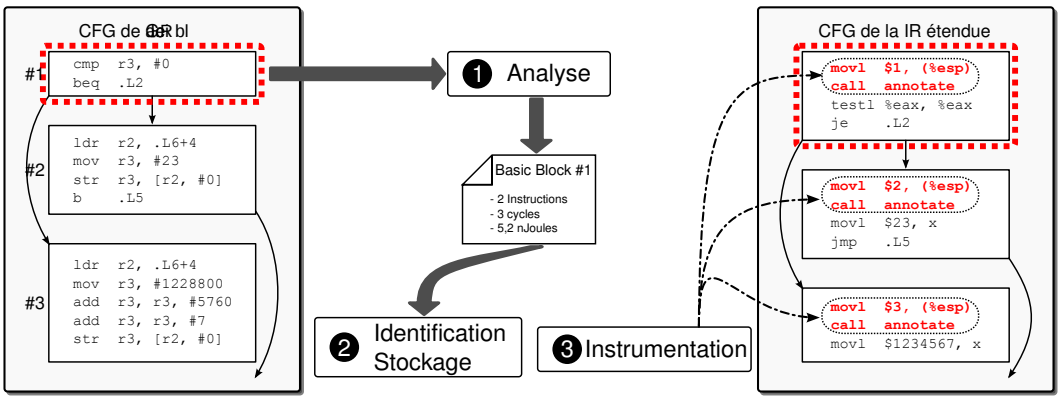


FIGURE 5.7 – Instrumentation d'un bloc de base

L'étape d'analyse ❶ des blocs de base et les informations qui en découlent sont complètement indépendantes du processus d'instrumentation. Son implémentation sera donc réalisée en dehors du contexte de l'instrumentation de code, en fonction du processeur cible et du type d'estimation que l'on souhaite faire (temps, consommation).

5.3.2 Instrumentation des arcs du CFG

Ce type d'instrumentation consiste à insérer des blocs de base supplémentaires sur les arcs du CFG étendu lorsque cela est nécessaire. Ces blocs de base ne contiendront aucune instruction liée au programme à instrumenter, mais uniquement des instructions permettant de faire l'appel à la fonction d'annotation. L'identifiant passé en argument permettra dans ce cas de retrouver les informations liées au coût de passage par un arc donné du programme lors de son exécution. Cette solution n'affecte pas le comportement du logiciel et bien que le CFG étendu soit modifié, l'équivalence entre les CFG cible et étendu reste correcte du point de vue de l'instrumentation.

Le principe d'instrumentation des arcs du CFG est donc très similaire à celui de l'annotation des blocs de base et suit également trois étapes. Ces étapes ne sont plus seulement effectuées pour chaque bloc de base mais pour chaque successeur de chaque bloc de base :

❶ **L'analyse** doit permettre de déterminer si l'arc entre un bloc de base et son successeur doit être instrumenté et fournir l'annotation le cas échéant. Si l'arc ne doit pas être annoté, les deux étapes suivantes sont ignorées.

❷ **L'identification et le stockage** des annotations relatives à l'arc instrumenté est faite de la même manière que pour l'instrumentation des blocs de base.

❸ **L'insertion** d'un bloc sur l'arc concerné du CFG de la IR étendue permettra d'ajouter un appel à la fonction d'annotation, appelée uniquement lorsque le programme prendra ce chemin de CFG.

Ces trois étapes sont représentées sur la figure 5.8. Sur cette figure, seul l'arc entre le bloc de base n°1 et le n°3 est annoté.

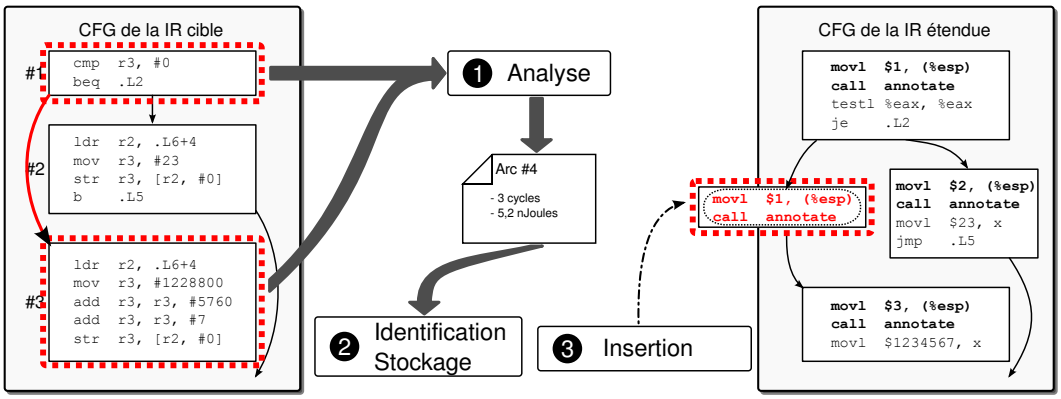


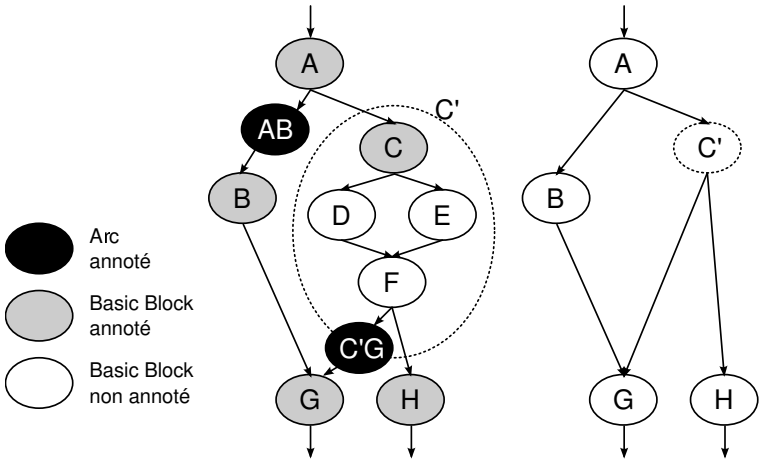
FIGURE 5.8 – Instrumentation d'un arc du CFG de la IR étendue

Le non-isomorphisme des CFG est également à prendre en considération lors de la passe d'instrumentation des arcs. En effet, dans ce cas, il ne faudra pas prendre en compte les successeurs d'un bloc de base dans le CFG de la IR étendue mais les successeurs du sous CFG équivalent au bloc de base en cours d'analyse dans le CFG de la IR cible. Ce cas est représenté sur la figure 5.9(a) où le sous CFG $CDEF$ est équivalent au bloc de base C' de la figure 5.9(b). L'annotation de l'arc $C'G$ du CFG de la IR cible devra être insérée sur l'arc FG dans le CFG de la IR étendue.

5.4 Limitations

La première contrainte de l'approche proposée est de disposer d'un compilateur offrant à la fois un *back-end* pour le processeur cible et pour le processeur hôte. Cette condition remplie, la principale limitation est liée à l'implémentation de la technique proposée.

La passe d'instrumentation décrite jusqu'alors considère que le CFG de la IR étendue est équivalent au CFG de la IR cible. Cela sous-entend que toutes les transformations effectuées sur ce dernier ont pu être répercutées sur le CFG de la IR étendue ou ont pu trouver



(a) CFG IR étendue (b) CFG IR cible

FIGURE 5.9 – Instrumentation des arcs dans un CFG étendu équivalent non isomorphe

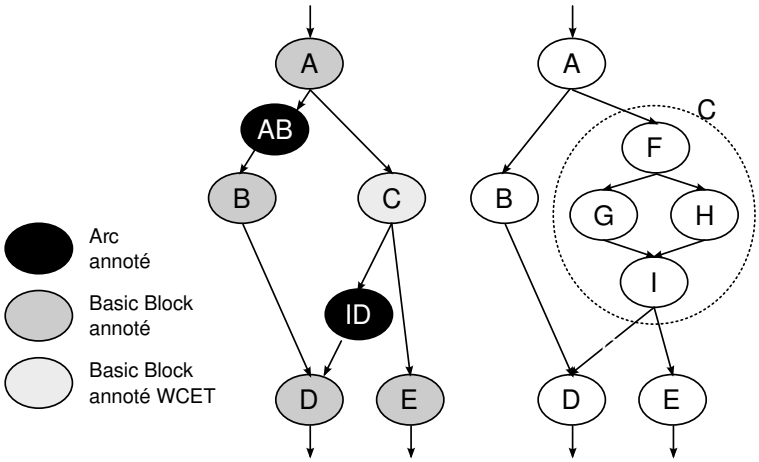
une équivalence. Dans certains cas, il se peut que des transformations ne soient pas applicables sur le CFG étendu pour des problèmes techniques dus par exemple à une suite d'optimisations complexes ou très spécifiques de certaines parties du CFG cible.

Cette approche nécessite idéalement la maîtrise de toutes les passes d'optimisations effectuées dans le *back-end* afin de maintenir l'équivalence des CFG. A moins d'être le développeur d'une de ces passes, une phase de rétroconception (reverse engineering) plus ou moins fastidieuse peut alors être nécessaire. En fonction de la complexité algorithmique, de la qualité d'écriture ou encore de la documentation du code source de ces passes, cette rétroconception peut également être une des raisons pour laquelle toutes les modifications ne pourront pas nécessairement être prises en compte. Un des avantages de l'approche mise en œuvre face à ce genre de situation est sa flexibilité face à ce genre de situation. En effet, si l'équivalence de CFG n'a pas pu être garantie, des approximations pourront éventuellement être faites en utilisant des approches d'estimation de pire temps d'exécution (WCET) localement.

La figure 5.10(b) montre un exemple de transformation sur le CFG de la IR cible n'ayant pas pu être répercutée sur le CFG de la IR étendue. Le bloc de base C a été ici transformé en un sous-CFG $FGHI$ dans la IR cible. Dans ce cas, seul le bloc C sera instrumenté, il sera de la responsabilité de l'analyse de fournir une estimation sur le sous CFG $FGHI$. L'annotation des arcs pourra également être effectuée de manière réciproque à celle effectuée dans les CFG équivalents, en considérant cette fois-ci le sous CFG dans la IR cible. Dans cet exemple, une pénalité de branchement correspondant à l'arc ID du CFG cible est insérée sur l'arc CD du CFG de la IR étendue.

Une des contraintes de l'approche proposée, commune à toutes les solutions basées sur l'utilisation de la représentation intermédiaire des compilateurs, est la nécessité de disposer du code source du logiciel à instrumenter. De plus, ce code source ne doit faire l'usage d'*aucune* instruction en assembleur.

Cette contrainte, qui s'applique d'une manière plus générale à toutes les approches basées sur l'exécution native, peut s'avérer pénalisante à différents points de vues. Tout d'abord, les développeurs logiciels sont amenés parfois à utiliser le langage assembleur du processeur cible afin d'implémenter des algorithmes critiques en temps d'exécution ou pour forcer l'utilisation d'instructions très spécifiques qui ne seront pas générées automatiquement par le compilateur. Dans un milieu industriel, l'utilisation de programmes



(a) CFG IR étendue (b) CFG IR cible

FIGURE 5.10 – Instrumentation CFG étendu non équivalent

fournis par des tiers et pour lesquels les sources ne sont évidemment pas disponibles est également très fréquente (bibliothèques propriétaires, systèmes d'exploitation, etc.). Enfin, le compilateur lui-même est également à l'origine de l'insertion dans le code binaire de portions de code qui ne pourront êtres instrumentées. C'est le cas des primitives dites « *built-in* » utilisées afin d'optimiser les performances en fonction des instructions disponibles sur le processeur cible. Ainsi une instruction de division pourra par exemple être remplacée par un appel de fonction *built-in* si le processeur n'en dispose pas, ou encore la fonction `memcpy()` de la bibliothèque standard C peut être remplacée dans GCC par `builtin_memcpy()` suivant la valeur des arguments au moment de la compilation.

Toutefois, une méthode basée sur la décompilation du code binaire pour reconstruire une représentation intermédiaire [RBL06] pourrait être une solution envisageable afin de prendre en considération les cas présentés ci-dessus .

Une autre limitation concerne le code « auto-modifiant »,pratique jugée en général inacceptable, mais qui est nécessaire dans le cas précis de chargement des bibliothèques dynamiques, ou de l'interprétation d'un programme compilé à la volée (JiT pour Just in Time). Les solutions efficaces restent à trouver.

Pour conclure sur les limitations, les optimisations pouvant êtres effectuées au moment de l'édition de lien (LTO pour Link Time Optimization) [DBDSVP+04] ne sont également pas prises en compte.

5.5 Implémentation dans LLVM

5.5.1 L'environnement LLVM

LLVM (Low Level Virtual Machine) est une infrastructure de compilateur *Open Source* [LA04], c'est à dire un ensemble de modules et de composants réutilisables permettant de créer des machines virtuelles[2] ou des compilateurs. La figure 5.11 représente une architecture type de compilateur utilisant les composants LLVM. Le cœur de cette architecture est la représentation intermédiaire, également nommée *LLVM*, conçue pour

2. Une machine virtuelle est un logiciel permettant de simuler le comportement d'une machine (processeur, ordinateur, etc.) sur une machine réelle utilisée comme support d'exécution dont le principe date de 1969 [Ric69], la plus connue est la JVM pour le langage Java.

permettre l'implémentation de transformations et d'optimisations « bas niveau » (mais indépendantes du processeur cible) durant l'étape de *middle-end*. La représentation *LLVM* peut être manipulée sous trois formats différents :

- Un format mémoire utilisé par les différents outils basés sur LLVM (compilation, exécution à la volée).
- Un format de fichier binaire (*bytecode*), pour le stockage sur disque.
- Un format « texte », équivalent à un assembleur lisible.

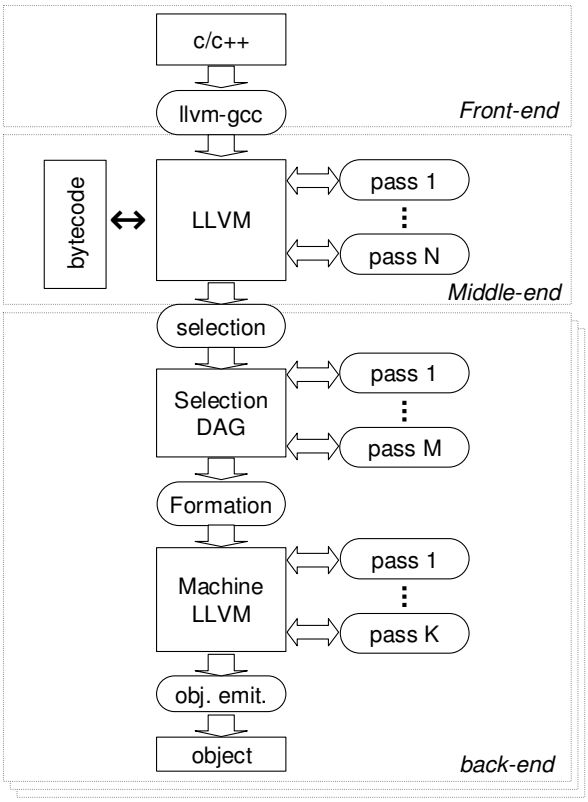


FIGURE 5.11 – Architecture d'un compilateur basé sur LLVM

L'analyse de programme en langage C peut être faite à l'aide du compilateur GNU GCC, utilisé ici en tant que *front-end*. LLVM fournit également l'infrastructure permettant de construire des *back-end* basés sur une seconde représentation intermédiaire appelée *Machine-LLVM*.

Un programme est représenté dans LLVM sous forme de *Modules*. Chaque module est lui-même composé de fonctions, de variables globales et de tables de symboles, décrites à partir de la représentation *LLVM* ou *Machine-LLVM*. Si le programme est construit à partir d'un langage logiciel (ce qui n'est pas nécessairement le cas dans LLVM), chaque *Module* correspondra à la traduction d'un fichier du programme dans la représentation *LLVM*. Tous les traitements sur les représentations intermédiaires *LLVM* et *Machine-LLVM* sont organisés sous forme d'un ensemble de « passes » de compilation contrôlées par un gestionnaire (*pass manager*).

L'intégration dans LLVM de la technique d'instrumentation proposée est représentée sur la figure 5.12. La représentation *LLVM* est étendue tout au long du *back-end* et doit être maintenue à jour lors des modifications effectuées dans les passes de compilation, jusqu'à la passe d'instrumentation.

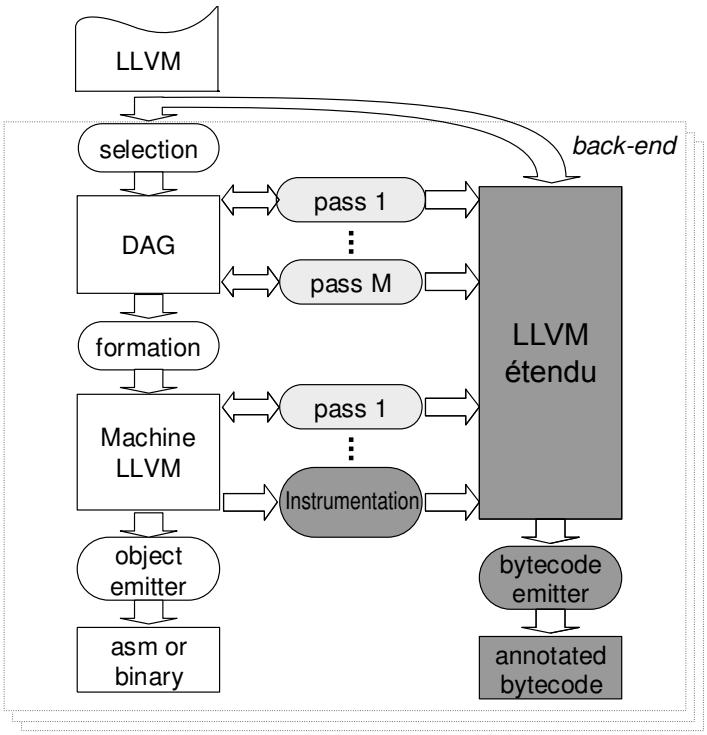


FIGURE 5.12 – Architecture de l'environnement LLVM étendue pour l'instrumentation de code

Un des avantages de LLVM est le lien qui existe entre la représentation *Machine-LLVM* dépendante du processeur cible et la représentation *LLVM* issue du *middle-end* (indépendante du processeur). Ainsi, tout au long du *back-end* spécifique, il est possible par exemple de manipuler à la fois la vue *Machine-LLVM* et *LLVM* d'une fonction ou d'un bloc de base. Idéalement, si, lors de l'implémentation des passes spécifiques à un processeur, le lien entre les deux niveaux de représentation était maintenu, l'isomorphisme de CFG requis pour l'annotation de code serait directement disponible à la fin du *back-end*. Malheureusement, en pratique, toutes les passes de compilation ne maintiennent pas la cohérence entre les deux représentations et la phase de rétroconception évoquée précédemment est nécessaire.

Le *back-end* que nous avons retenu pour implémenter cette méthode d'instrumentation est celui du processeur ARM. L'intérêt d'utiliser ce *back-end* est en partie dû au fait que le jeu d'instruction de la représentation intermédiaire *LLVM* couvre relativement bien le jeu d'instructions du processeur ARM, limitant ainsi les transformations effectuées lors de la phase de sélection des instructions. L'objectif ici n'étant pas de réécrire les nombreuses passes de compilations, mais de vérifier que la manière dont le CFG est instrumenté permet bien de refléter l'exécution du programme sur le processeur cible. Certaines passes de simplification de branches, telles que *AnalyzeBranch*, *IfConverter* ou encore *BranchFolder*, ont été modifiées afin de maintenir la représentation *LLVM* à jour. Cela n'a pas été réalisé pour toutes les passes de compilation, et nous verrons dans la suite que certaines parties des CFG peuvent ne pas être annotées. Celles-ci restent cependant peu nombreuses dans le cas du processeur ARM.

5.5.2 Implémentation de la passe d'instrumentation du code

Bien que la passe d'instrumentation soit exécutée à la fin du *back-end*, celle-ci est implémentée indépendamment du processeur cible. Elle sera ainsi directement utilisable pour tous les types de processeurs disponibles dans LLVM (x86, SPARC, MIPS, Itanium, SPU CELL, ...). L'ajout du support d'instrumentation pour un processeur donné consistera alors uniquement à répercuter les transformations effectuées dans le *back-end* sur le CFG *LLVM* et à implémenter les deux méthodes `MachineBasicBlockAnnotation()` et `MBBBranchAnnotation()` dans le *back-end* spécifique au processeur cible, et qui seront appelées par la passe d'instrumentation. Ces deux méthodes retourneront les informations d'annotations dans un tableau d'octets, qui sera stocké en mémoire sous forme de données initialisées.

D'un point de vue plus technique, la passe d'instrumentation est une classe C++ qui dérive de la classe de base *MachineFunctionPass* et qui redéfini la méthode *runOnMachineFunction* qui sera appelée pour chaque fonction du programme. Le cœur de l'instrumentation se trouve donc implémenté dans cette méthode et est décomposé en deux phases. La première consiste à parcourir les représentations *LLVM* et *Machine-LLVM* afin de construire des structures de données qui nous permettront de déterminer si les *CFG* sont équivalents. Ces structures de données permettent principalement de mettre en correspondance les blocs de base de chacune des représentations et de faciliter la détection de problèmes d'équivalence dans le CFG.

La deuxième phase de l'instrumentation, représentée dans l'algorithme simplifié 1, consiste à parcourir tous les blocs de la représentation *LLVM* (ceux devant être annotés). Pour chacun d'entre eux, l'appel à la méthode `check()` permet de déterminer s'il y a équivalence ou non avec la représentation *Machine-LLVM*.

Algorithm 1 Algorithme simplifié de l'instrumentation des *blocs de base* et de arcs du CFG

```
1:  for all BasicBlock in LLVM Function do
2:    if check(BasicBlock) = true then
3:      annotationDB = MachineBasicBlockAnnotation(BasicBlock);
4:      if annotationDB ≠ NULL then
5:        annotateLLVMBasicBlock(annotationDB, BasicBlock);
6:      end if
7:      for all SuccBasicBlock in BasicBlock successors do
8:        annotationDB = MBBBranchAnnotation(BasicBlock, SuccBasicBlock);
9:        if annotationDB ≠ NULL then
10:         annotateLLVMArc(annotationDB, BasicBlock, SuccBasicBlock);
11:       end if
12:     end for
13:   end if
14: end for
```

Si tel est le cas, la méthode `MachineBasicBlockAnnotation()` (ligne 3) permet d'obtenir la base de données d'annotation relative au bloc en cours de traitement. La méthode `annotateLLVMBasicBlock()` (ligne 5) stocke cette base de données en mémoire et insère un appel à la fonction `annotate()` (déclarée externe) en début du bloc de base, lui passant comme argument l'adresse de la base de données en mémoire (plus exactement son *label*, dont l'adresse sera déterminée à l'édition de liens finale). Le listing 5.3 est la représentation désassemblée d'un bloc de base *LLVM* instrumenté (sous la forme assembleur de LLVM).

L'appel à la fonction `annotate()` est la première instruction à la ligne 2. Elle prend ici en argument l'adresse définie par le label `@bb92_db` et qui pointe un tableau de 12 octets (`[12 x i8]`) en mémoire.

Listing 5.3 Exemple de bloc de base *LLVM* instrumenté

```
1  bb92:   ; preds = %bb91_split, %bb78
2    call void @annotate( i8* getelementptr ([12 x i8]* @bb92_db, i32 0, i32 0) )
3    %448 = load i32* %flit_size, align 4    ; <i32> [#uses=1]
4    %449 = load i32* %step, align 4   ; <i32> [#uses=1]
5    %450 = icmp ult i32 %449, %448    ; <i1> [#uses=1]
6    %451 = zext i1 %450 to i8   ; <i8> [#uses=1]
7    %toBool93 = icmp ne i8 %451, 0    ; <i1> [#uses=1]
8    br i1 %toBool93, label %bb79.branch_penalty, label %bb94
```

Le listing 5.4 correspond quant à lui à la version assembleur du bloc de base précédent, compilé pour le processeur cible. On retrouve l'adresse de la base de donnée `@bb92_db` qui est passée en argument à la ligne 2, avant l'appel à la fonction `annotate()` à la ligne 3. Les données associées à ce bloc sont directement stockées avec le code à la ligne 14, elles sont composées ici des valeurs 5, 7 et 2 représentées sous forme de mots de quatre octets.

Listing 5.4 Code assembleur généré pour un processeur hôte *x86*

```
1  .LBB1_51: # bb92
2    movl  $bb92_db, (%esp)
3    call  annotation
4    movl  572(%esp), %eax
5    cmpl  %eax, 4328(%esp)
6    jb  .LBB1_39  # bb79.branch_penalty
7
8    ...
9
10   .type bb92_db,@object
11   .align  16
12 bb92_db:        # bb92_db
13   .size bb92_db, 12
14   .asciz  "\005\000\000\000\007\000\000\000\002\000\000\000"
```

L'annotation des arcs consiste à parcourir chacun des blocs de base successeurs du bloc courant et appeler la méthode `MBBBranchAnnotation()` (ligne 8 de l'algorithme 1). L'annotation de l'arc courant est ensuite réalisée (si nécessaire) par l'appel à la méthode `annotateLLVMArc()` (ligne 10). Elle consiste à insérer un nouveau bloc de base sur cet arc puis à insérer l'appel à la fonction `annotate()` comme pour l'annotation des blocs de base décrite auparavant.

Afin de faciliter la débogage de cette instrumentation, nous avons également implémenté une fonction permettant de représenter les CFG *LLVM* (celui annoté) et *Machine-LLVM* (celui du processeur cible) en faisant apparaître les liens entre eux. Nous générons pour cela un fichier texte au format *DOT* permettant de décrire des graphes qui peuvent ensuite être visualisés avec les outils *open source* disponibles dans GraphViz [GRA]. La figure 5.13 représente les CFG d'une fonction pour laquelle l'isomorphisme des graphes à été conservé malgré les optimisations. Chaque bloc de base *LLVM* est représenté dans le groupe *cluster_LLVM* et une connexion permet de visualiser le bloc de base correspondant dans la représentation *Machine-LLVM*. L'annotation des bloc n'est pas visible ici mais on peut remarquer l'insertion de trois bloc de base supplémentaires qui permettront de modéliser les pénalités de branchement (en vert ou gris foncé sur la figure).

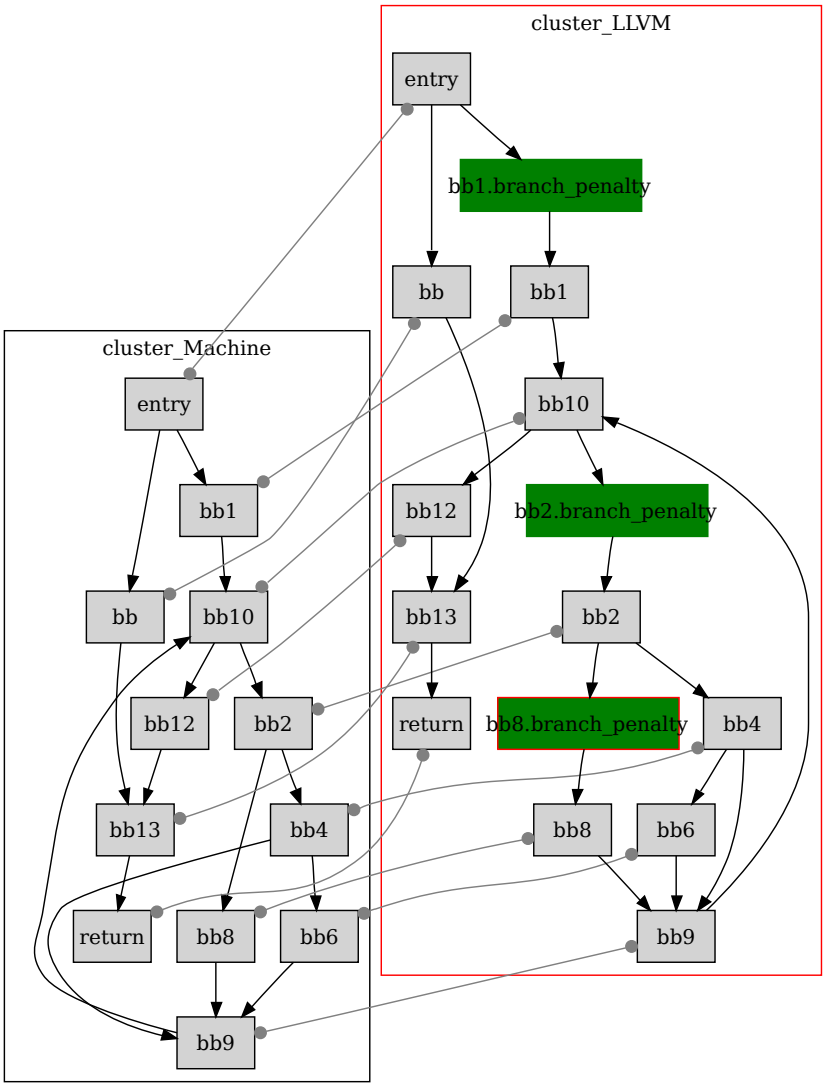


FIGURE 5.13 – Visualisation des CFG *LLVM* (à droite) et *Machine-LLVM* (à gauche) générés automatiquement lors de l'instrumentation

Les problèmes d'équivalence qui sont détectés dans la passe d'instrumentation, déclenchent systématiquement la génération des fichiers de débogage et les zones du CFG ayant posé des problèmes d'instrumentation sont mises en évidence. La figure 5.14 donne un exemple dans lequel le bloc nommé `bb` n'a pas pu être instrumenté. Celui-ci n'est pas équivalent au bloc de base `bb` de la représentation *Machine-LLVM* car le bloc `return` ne fait plus partie de ses successeurs dans le programme cible. L'implémentation actuelle de l'instrumentation n'annote pas les bloc qui ne sont pas équivalents.

5.6 Conclusion

La technique d'instrumentation que nous avons présentée dans ce chapitre permet d'insérer tout type d'informations relatives au programme compilé pour le processeur cible dans le CFG qui sera utilisé pour générer le code natif. Cela est possible en raison de l'indépendance entre le processus d'instrumentation et celui d'analyse des blocs de base et des

arcs du programme.

Bien que les cas des CFG non équivalent n'aient pas été complètement traités ici, cette approche est suffisamment flexible pour permettre d'utiliser d'autres solutions telles que les techniques WCET de manière très localisée dans le CFG, permettant ainsi de minimiser l'erreur dans le processus d'estimation.

Les expérimentations menées dans le chapitre 7 permettront de valider les capacités de cette approche à refléter l'exécution du programme cible sur le processeur de la machine hôte.

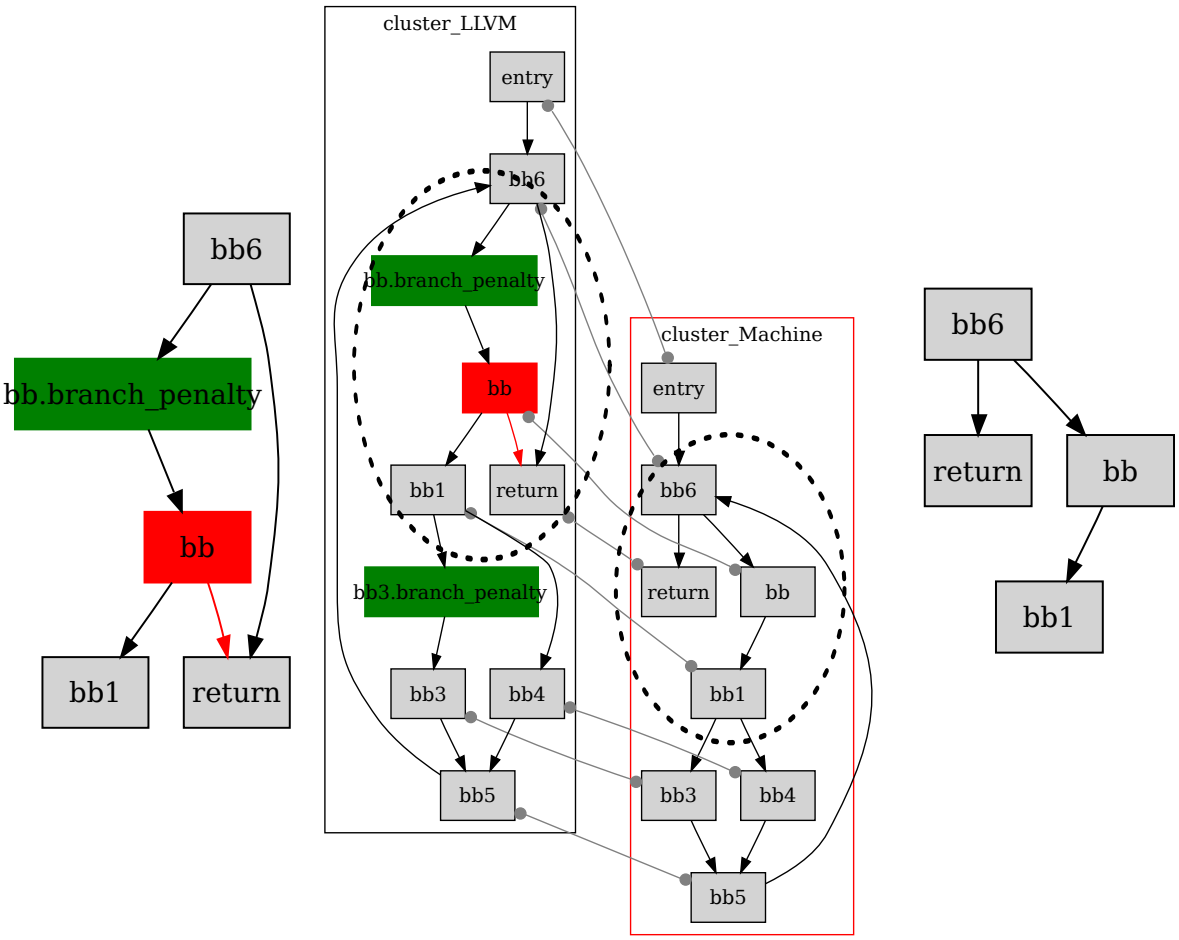


FIGURE 5.14 – Exemple de CFG non équivalent au niveau d'un bloc de base

Chapitre 6

Applications : Estimation des performances et profilage du logiciel

Sommaire

L'un des principaux objectifs de la technique d'instrumentation proposée dans le chapitre précédent est évidemment de permettre l'estimation des performances d'exécution du logiciel sur les plateformes de simulation d'architectures MPSoC natives.

La première application présentée dans la suite de ce document est une méthode d'estimation des performances, qui n'est pas une contribution à proprement dite de ce travail mais plutôt un exemple de mise en œuvre des deux approches proposées précédemment, dont le but est de montrer comment leurs utilisations conjointes peuvent être mises à profit pour la simulation d'architectures multiprocesseurs.

Un deuxième exemple d'application montre l'utilisation de la technique d'instrumentation pour insérer d'autres informations que celles dédiées à l'estimation du temps d'exécution dans le logiciel. Dans ce cas, il s'agit de faire le profilage d'un logiciel multi-tâches s'exécutant sur une architecture multiprocesseur symétrique.

6.1 Principes de base

Quelle que soit le type d'estimation effectuée, le principe d'exécution du logiciel instrumenté sur la plateforme de simulation native reste le même. Il est représenté sur la figure 6.1. Les appels aux fonctions de l'API HAL sont interceptés par l'unité d'exécution en cours de simulation et gérés comme nous l'avions décrit auparavant.

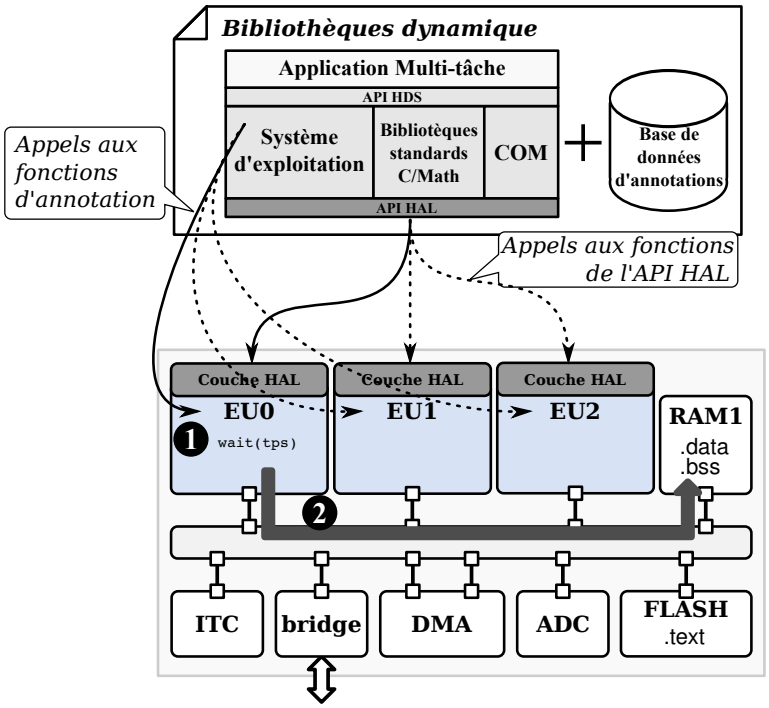


FIGURE 6.1 – Principe d'exécution du code annoté sur la plateforme de simulation native

A chaque exécution de bloc de base du programme instrumenté, les appels aux fonctions `annotateBB()` ou `annotateArc()` seront également interceptés par les EU. Celles-ci auront alors accès aux bases de données contenant les informations associées à chaque bloc de base ainsi qu'à toutes les ressources dont elles disposent et en particulier les interfaces de communication TLM. Les données d'annotations peuvent de cette manière être utilisées non seulement pour faire évoluer le temps de simulation ❶ mais aussi pour modéliser des accès vers le réseau de communication ❷ ou encore faire évoluer toutes grandeurs physiques contenues dans les annotations.

Les architectures multiprocesseurs restent « naturellement » supportées sur la plateforme de simulation malgré les annotations du logiciel. L'évolution des grandeurs mesurées (temps, consommation, etc.) est prise en compte de manière indépendante par chacune des EU, qui peuvent ou non exécuter une même pile logicielle.

6.2 L'estimation des performances

Les performances d'exécution d'une application dépendent de plusieurs facteurs pouvant être classés d'une manière générale en deux catégories : statique et dynamique.

La partie statique du temps d'exécution du logiciel est principalement liée au type de chaque instruction du programme, et peut être analysée sans aucune exécution.

L'aspect dynamique dépend quant à lui de multiples facteurs (congestion sur le réseau de communication, mémoires caches, pipeline du processeur, etc.) et ne peut être mesuré, dans un environnement multiprocesseur, qu'au moment de la simulation. Le lien avec les deux techniques proposées à la section précédente semble alors immédiat :

- La plateforme de simulation native doit permettre de prendre en compte le critère dynamique du temps d'exécution.
- L'instrumentation automatique du code permettra de modéliser le temps d'exécution du logiciel dépendant de l'architecture interne du processeur cible.

Cependant, plutôt que d'utiliser l'estimation du temps d'exécution statique du programme et le temps d'exécution dynamique de la simulation séparément, la solution proposée et mise en œuvre consiste à employer la technique d'instrumentation de code et la plateforme de simulation de manière conjointe. Les annotations sur le code logiciel ne contiendront donc pas uniquement des informations relatives au temps d'exécution du logiciel, mais également le maximum d'informations qui permettront à la plateforme de simulation de modéliser au mieux la partie dynamique du temps d'exécution global.

Le processus d'instrumentation avec lequel doit être intégré celui de l'estimation des performances se déroule en deux temps. La première phase consiste à insérer les annotations dans chacun des blocs de base et la seconde sur les arcs du CFG. La mise en œuvre de l'estimation consiste alors à fournir les méthodes d'analyse qui seront « appelées » lors de ces phases pour générer les informations d'annotation.

6.2.1 Analyse des blocs de base

Cette analyse sera effectuée sur chaque bloc de base de la représentation intermédiaire spécifique au processeur cible, et le résultat sera une structure de données contenant toutes les informations relatives à un bloc de base. Dans le cadre de l'estimation des performances, l'objectif de l'analyse est de déterminer le plus précisément possible le nombre de cycles d'horloge nécessaires à l'exécution de chaque instruction qu'ils contiennent. Pour une séquence d'instructions donnée, ce nombre est fonction de deux facteurs indépendants :

1. L'architecture interne du processeur : dans ce cas, une analyse statique doit permettre d'estimer le nombre de cycles d'horloge de manière relativement précise puisque toutes les instructions du programme cible sont connues à ce moment là.
2. L'architecture externe au processeur : l'analyse des instructions du programme qui accèdent à l'extérieur du processeur, typiquement les *load* et les *store* pour un RISC, ne peut fournir aucune estimation précise du nombre de cycles d'horloge dans ce cas. Nous déterminerons alors quelles informations sur le logiciel peuvent être utiles à la plateforme de simulation afin d'améliorer l'aspect dynamique du temps d'exécution

6.2.1.1 Analyse des dépendances liées à l'architecture interne

Au niveau de l'architecture interne du processeur, le nombre de cycles d'horloge nécessaires pour exécuter une séquence d'instructions dépend de la micro-architecture et peut être fort complexe. Dans le cas des processeurs RISC, cela dépend là encore de deux facteurs, l'un constant et l'autre variable. La partie constante provient uniquement du type

des instructions contenues dans les blocs de base et peut être très facilement déterminée à partir de la documentation sur le jeu d'instructions du processeur cible.

En ce qui concerne la partie variable, la précision avec laquelle celle-ci pourra être déterminée va dépendre de la complexité des mécanismes de l'architecture interne du processeur. Dans la phase d'analyse des blocs de base, nous avons considéré les deux sources de dépendance suivantes :

– Celles liées aux opérandes d'une instruction.
– Celles liées aux interdépendances entre instructions.

Dans la suite, les exemples d'analyse des blocs sont basés sur le jeu d'instructions du processeur *ARM9* (*ARM966ES*). La documentation sur le jeu d'instructions est issue du manuel de référence sur l'architecture du processeur.

Les opérandes d'une instruction :

Les opérandes d'une instruction peuvent affecter le nombre de cycles nécessaires à l'exécution de celle-ci en fonction de leurs valeurs. Lorsque ces valeurs ne sont pas connues lors de l'analyse statique, typiquement les valeurs des registres, il n'est pas possible de déterminer précisément le nombre de cycles que va prendre cette instruction. C'est le cas par exemple de la multiplication du processeur *ARM966ES* utilisée dans le bloc de base 1 de la figure 6.2 (instruction n°4) et dont le nombre de cycles peut être déterminé à partir du tableau 6.1, issu du manuel de référence *ARM*.

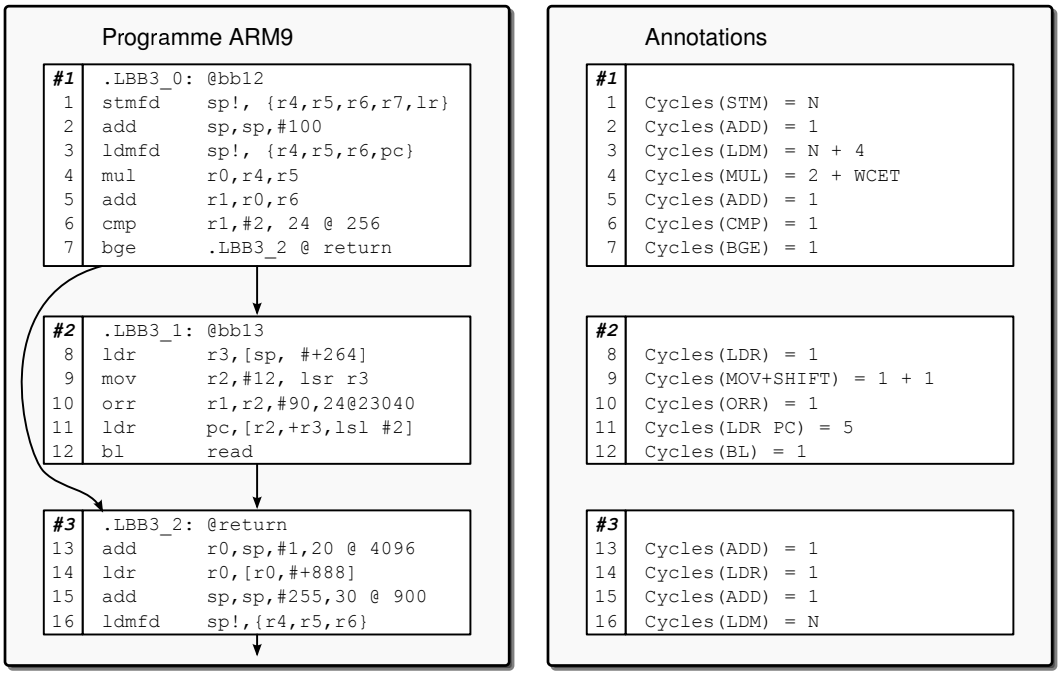


FIGURE 6.2 – Annotation des instructions avec prise en compte des opérandes

Dans la figure 6.2, l'instruction n°4 est annotée avec le WCET (« *le pire cas* ») de la multiplication, $Cycle(MUL) = 2 + WCET(M) = 6$.

De la même manière, le nombre d'opérandes peut directement affecter le nombre de cycles d'exécution comme c'est le cas pour les instructions 1, 3 et 16 de la figure 6.2. Il en

Syntaxe	$MUL< cond >SRd, Rm, Rs$
Sémantique	$Rd = Rm * Rs$
Cycles Instruction	$2 + M$
Où	$M = 1$ for $-2^8 \leq Rs < 2^8$
	$M = 2$ for $-2^{16} \leq Rs < 2^{16}$
	$M = 3$ for $-2^{24} \leq Rs < 2^{24}$
	$M = 4$ for $-2^{32} \leq Rs < 2^{32}$

TABLE 6.1 – Spécification de l'instruction *MUL* du processeur *ARM9*

est de même pour le type des opérandes utilisés dans l'instruction, qui peut également influencer le temps d'exécution, par exemple lorsque le registre *PC* (le compteur programme) est modifié lors d'un chargement, ce qui revient à exécuter un branchement.

Ainsi, le nombre de cycles d'exécution de l'instruction *LDM* permettant de faire un chargement multiple de registres sur le processeur *ARM* est déterminé par l'équation suivante :

$$Cycles(LDM) = N + 4 \times P + 1 \times S \tag{6.1}$$

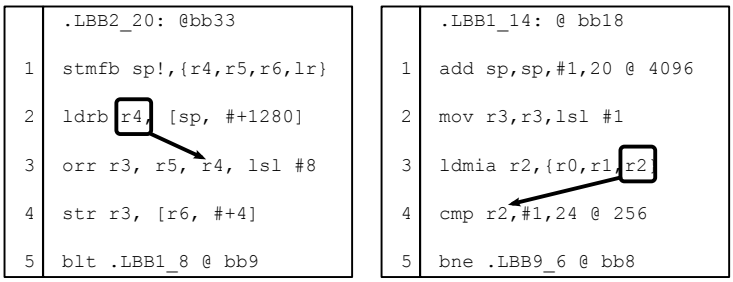
Où, N est le nombre de registres à charger, P un booléen représentant la présence du PC dans la liste des registres et S un booléen représentant le fait que N est égale à 1.

Interblocages d'instructions :

Afin d'atteindre des performances plus élevées, les architectures des processeurs utilisent des *pipelines* ou encore plusieurs unités fonctionnelles sur lesquelles plusieurs instructions peuvent être exécutées simultanément.

Dans ce cas, l'exécution d'une instruction ne peut démarrer que si tous ses opérandes sources ont été évalués, par la ou les instructions précédentes. Dans les processeurs embarqués classiques, les temps de latence dus à l'interblocage entre deux instructions peuvent être simplement déterminés de manière statique et correspondent à une constante.

Dans la figure 6.3(a), nous pouvons voir une relation de dépendance entre les instructions n°2 et n°3. Cette dépendance est due à l'utilisation d'un registre dans l'instruction `orr` directement après l'instruction de chargement `ldrb` de ce même registre. Une dépendance similaire existe également entre les instructions n°3 et n°4 de la figure 6.3(b). Afin de prendre en compte ces dépendances, des cycles d'horloges supplémentaires sont ajoutés à ceux déjà estimés par l'analyse des instructions, ce qui permet de modéliser les cycles de latences dus au pipeline du processeur.



(a) (b)

FIGURE 6.3 – Interblocages entre instructions

Dans des processeurs plus complexes, l'exécution d'instructions peut éventuellement démarrer même si les données ne sont pas encore disponibles. Des techniques de réservation sont également employées et permettent aux instructions de réserver des ressources matérielles pour des utilisations futures. Le nombre de cycles d'horloge lié à l'utilisation de ce type d'architecture complexe n'a pas été pris en compte dans cette approche. Cependant, l'utilisation de techniques dont l'objectif est le calcul du pire temps d'exécution de bloc de base sur des processeurs superscalaires ayant des ordonnancements dynamiques d'instructions, telle que celle décrite dans [RS09], peut aider à résoudre ce problème.

6.2.1.2 Dépendances liées à l'architecture externe

L'architecture externe au processeur, c'est à dire celle de la plateforme matérielle, est responsable d'un nombre de cycles d'horloge supplémentaires non négligeable par rapport à ceux dus uniquement à l'architecture interne. Ces cycles d'horloge additionnels sont dus aux différents accès en lecture ou en écriture effectués par le processeur vers les mémoires ou les périphériques du système, auxquels s'ajoutent les effets des mémoires caches. Deux types d'accès sont donc à prendre en compte :

- Les chargements d'instructions à partir de la mémoire.
- Les accès en mémoire ou en entrées/sorties des données.

Étant donné l'absence complète d'informations sur l'architecture externe au processeur dans l'environnement du compilateur utilisé pour réaliser l'instrumentation du code, aucune information temporelle ne peut être déduite directement à partir de la représentation intermédiaire dont nous disposons au moment de l'analyse. L'analyse de cette représentation intermédiaire peut cependant fournir certaines informations structurelles pouvant aider le modèle de la plateforme matériel sous-jacent dans son processus d'estimation du temps au moment de la simulation.

Chargement des instructions du programme :

C'est un contributeur important dans le temps d'exécution du logiciel. Dans de nombreuses approches, ce type d'information n'est pas pris en compte [LP02, **?**, PHS+04, KKW+06]. L'impact dans des architectures simples et en particulier faiblement parallèle peut certes être négligeable, mais il ne l'est plus dès lors que le nombre de processeurs augmente. En effet, l'exécution d'un programme sur un processeur donné peut affecter le temps d'exécution d'un programme sur un autre processeur en raison des congestions d'accès aux ressources communes.

Idéalement, l'adresse d'origine d'un bloc de base dans l'espace mémoire cible ainsi que sa taille sont deux informations suffisantes pour modéliser les accès que son exécution aura généré (au niveau d'abstraction de la simulation considérée) et ceci en tenant compte des mécanismes utilisés dans les mémoires caches. Lors de la phase d'analyse, les seules informations dont nous disposons sont la taille du bloc de base en mémoire cible, et un *label* qui nous permettra de connaître son adresse dans l'espace mémoire hôte lors de la simulation.

Ne connaissant pas l'adresse précise dans l'espace mémoire cible, il ne sera pas possible dans la plateforme de simulation d'implémenter un modèle de mémoire cache des instructions précis. Cependant, une implémentation simple peut être réalisée en ne considérant que la taille des blocs de base et une estimation du taux de défaut de cache, pouvant

être obtenu soit par simulation sur des plateformes plus précises, ou être un paramètre de configuration utilisé dans le cadre de l'exploration d'architecture. L'adresse du bloc de base dans l'espace mémoire hôte sera quant à elle utilisée par la plateforme de simulation pour modéliser les communications engendrées sur le réseau entre le processeur et la mémoire concernée.

Finalement la prise en compte des accès en lecture aux instructions du programme consiste à ajouter aux informations d'annotation des blocs de base uniquement la taille qu'ils occupent en mémoire, leurs adresses de base ne pourront être connues qu'au moment de l'exécution, lors des appels à la fonction d'annotation (qui, nous le rappelons, est implémentée dans la plateforme de simulation).

Les accès en lecture ou en écriture aux données :

Les accès vers les mémoires de données sont beaucoup plus difficiles à prendre en compte. Pour une partie importante des accès effectués en mémoire de données dans le programme cible, il n'est pas possible de déterminer une adresse équivalente dans l'espace mémoire hôte utilisé lors de la simulation, leurs adresses étant calculées lors de l'exécution du programme. Quoiqu'il en soit, en raison de leur diversité spatiale et temporelle, une instrumentation précise nécessiterait l'annotation de chacun des accès en mémoire, ce qui entrainerait une diminution importante des performances de simulation, comme c'est le cas dans [WH09] où tous les accès sont dans ce cas pris en compte.

Une première solution proposée dans [GHP09] part du principe que tous les accès en mémoire effectués dans le programme cible existent aussi d'une certaine manière dans la représentation intermédiaire étendue du compilateur (indépendante du processeur). La différence d'architecture entre le processeur cible et le processeur hôte résultera en un nombre différent d'accès pour une même variable. Typiquement, l'exécution native d'un programme sur un processeur hôte tel que le processeur Intel x86 génèrera un nombre d'accès différent en comparaison à un processeur de type RISC tels que les processeurs ARM, Mips ou Sparc, couramment utilisés dans les systèmes embarqués. Cette technique nécessite cependant une analyse supplémentaire qui est celle de la IR *étendue* compilée pour le processeur cible, alors que nous disposons uniquement de la IR *étendue* dans sa forme indépendante du processeur.

Une autre solution proposée ici consiste uniquement à extraire le nombre d'accès en lecture et en écriture aux mémoires de données effectués dans un bloc de base du programme cible (dont nous disposons au moment de l'analyse). Ne connaissant pas les adresses précises de ces accès sur le réseau de communication, celles-ci seront générées aléatoirement vers les mémoires accessibles dans la plateforme matérielle. Cette solution s'inspire de la technique employée dans [LP02], basée sur la génération de défauts de cache de manière aléatoire.

Le contenu d'une base de données d'annotation associée à un bloc de base sera donc finalement composé des informations suivantes :

- Nombre d'instructions du bloc de base
- Nombre de cycles estimés
- Taille du bloc de base
- Nombre d'accès à la mémoire en lecture
- Nombre d'accès à la mémoire en écriture

Plus de détails seront donnés sur la prise en charge des informations d'annotation et la modélisation des accès en mémoire par la plateforme de simulation dans la section 6.2.3.

6.2.2 Analyse des arcs du CFG

Cette phase d'analyse sera « appelée » pour chaque successeur de chaque bloc de base de la représentation intermédiaire. Elle consiste à déterminer si une pénalité de branchement doit être affectée à l'arc reliant les deux blocs concernés et auquel cas, retourner les informations nécessaires dans une structure de données comme cela est fait pour l'analyse des blocs de base présentée précédemment. Le processus d'instrumentation insèrera alors un bloc supplémentaire sur cet arc afin que cette annotation soit prise en compte lors de la simulation.

Les informations fournies par cette analyse dépendent de la politique de prédiction de branchement utilisée par le processeur cible. Dans le cas d'une politique simple, du type *pris/non-pris*, la base de données associée à l'arc ne contiendra que le nombre de cycles d'horloge de pénalité de branchement. Pour des politiques plus complexes (bimodales, locales, etc.), le mécanisme de prédiction de branchement pourra éventuellement être modélisé dans la plateforme de simulation et des informations supplémentaires ajoutées à l'annotation correspondante de l'arc du CFG.

Le processeur *ARM9* utilisé dans l'exemple qui suit considère que les branchements sont toujours *non-pris*, comme de nombreux processeurs embarqués. Dans ce cas, seuls les arcs correspondants aux branchements *pris* ont un coût supplémentaire et doivent être annotés.

La figure 6.4(a) représente un CFG non annoté, contenant deux instructions de branchement. Si l'on considère la politique de branchement du processeur *ARM9*, seuls les arcs $BB1 \rightarrow BB3$ et $BB3 \rightarrow BB1$ doivent être annotés. Les informations d'annotation seront donc retournées par cette phase d'analyse seulement pour ces deux arcs, qui seront alors les seuls annotés par le processus d'instrumentation. Le CFG obtenu est représenté sur la figure 6.4(b).

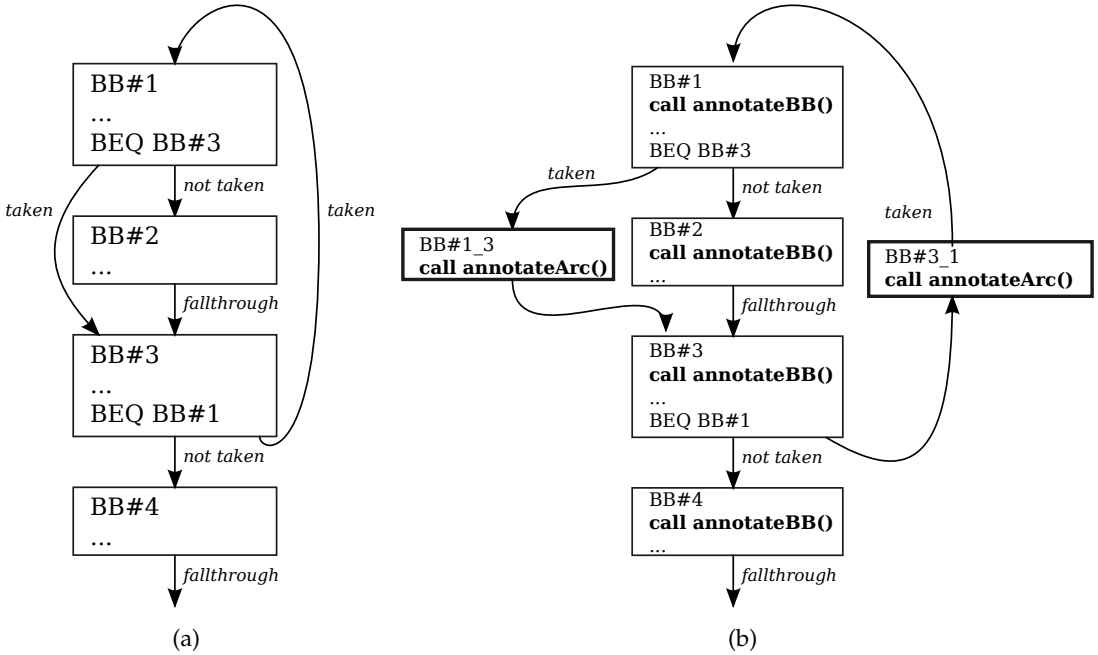


FIGURE 6.4 – (a) CFG du programme cible et (b) CFG du programme hôte équivalent avec les annotations des pénalités de branchement

6.2.3 Exécution du logiciel annoté sur la plateforme de simulation native

La figure 6.5 représente une portion de CFG d'un programme annoté. Ces annotations n'affectent en rien le principe d'exécution du logiciel natif sur la plateforme de simulation. Lorsque la simulation débute, chaque unité d'exécution va appeler la fonction servant de point d'entrée du logiciel, qui s'exécutera ensuite de manière atomique jusqu'à un appel à une fonction du HAL, et dans le cas du programme annoté, jusqu'à l'appel des fonctions `annotateBB()` et `annotateArc()`.

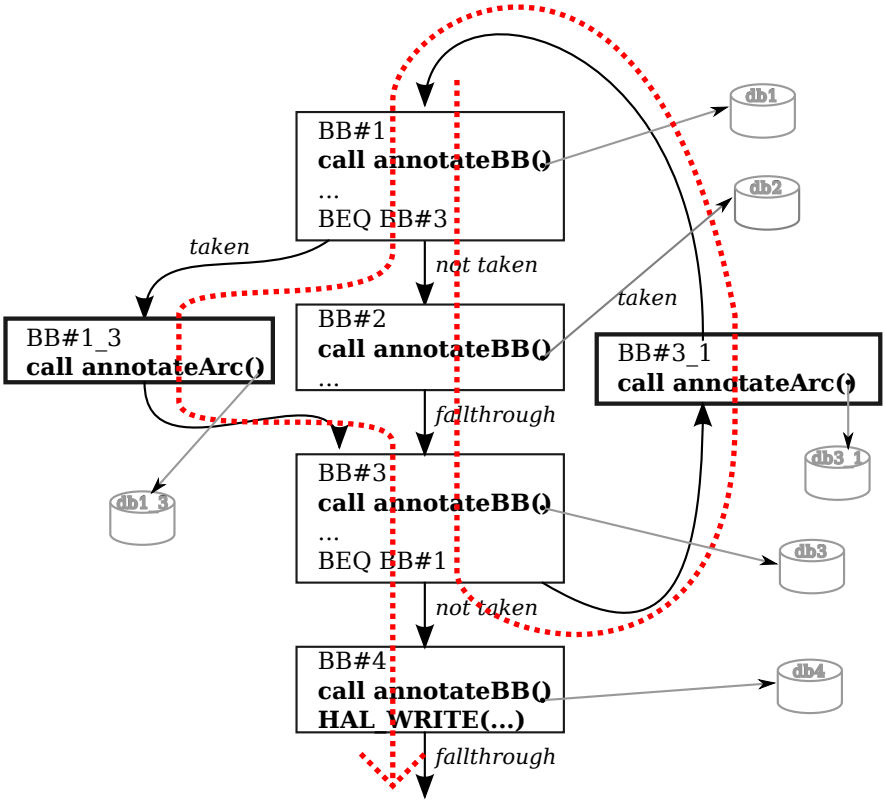


FIGURE 6.5 – Chemin d'exécution d'un programme annoté

Lorsque les fonctions `annotateBB()` et `annotateArc()` sont interceptées par une unité d'exécution, les informations relatives aux blocs de base ne sont pas interprétées immédiatement pour des raisons évidentes de performance de simulation, mais les pointeurs vers les bases de données passées en argument sont stockés dans une mémoire tampon et seront traités ultérieurement.

De la même manière que pour un programme non annoté, la synchronisation entre le matériel et le logiciel se fait lors des appels aux fonctions du HAL. C'est le cas dans cet exemple lorsque le bloc de base `BB\#4` contenant un appel à `HAL_WRITE` est exécuté. Ainsi, si l'exécution du CFG de la figure 6.5 débute sur le bloc de base `BB#1`, le contenu de la mémoire tampon avant la synchronisation sera `{db1, db2, db3, db3_1, db1, db1_3, db_3, db4}`.

La consommation du temps de synchronisation constant $TPS_{synchro}$ dans la fonction de synchronisation présentée dans la version fonctionnelle de la plateforme de simulation native est remplacée ici par l'analyse des bases de données contenues dans la mémoire tampon.

A l'issue de cette analyse, la modélisation du temps dans la plateforme de simulation doit être réalisée comme suit :

1. Modélisation des chargements d'instruction : après l'application du taux de défaut de cache en instruction configuré, l'unité d'exécution génère autant d'accès que nécessaires en lecture sur le réseau de communication vers la mémoire concernée (connue à partir des adresses des blocs de base exécutés).
2. Temps d'exécution statique : le temps d'exécution estimé des blocs de base est consommé par un appel à la fonction `wait()` de SystemC.
3. Accès en mémoire de données : les accès en lecture et écriture vers les mémoires de données sont générés après l'application du taux de défaut de cache en données. Aucune lecture ou écriture n'est réellement effectuées (celles-ci ont déjà été faites lors de la « vrai » exécution du programme natif), seuls les accès sont générés afin de modéliser la congestion sur le réseau de communication.

6.2.4 Synchronisation entre le matériel et le logiciel instrumenté

Comme nous l'avons dit précédemment, le rôle des fonctions `annotateBB()` et `annotateArc()` est de stocker les pointeurs sur les annotations des blocs de base en mémoire. Ces annotations seront traitées lors de la synchronisation entre le matériel et le logiciel, qui s'effectue sur la plateforme de simulation native à chaque appel à une fonction de l'API HAL. Cette granularité de synchronisation s'appuie sur le concept de *segment* introduit dans l'approche [Wol02]. D'après ce concept, la synchronisation dans les modèles de co-simulation est nécessaire uniquement lorsque l'application interagit avec le reste du système, c'est-à-dire uniquement lorsqu'une fonction du HAL est appelée. Cette condition n'est malheureusement pas applicable sur notre plateforme de simulation comme sur tout autre modèle de simulation dès lors que les mécanismes d'interruptions et le temps d'exécution du logiciel sont pris en compte.

Le problème est alors très similaire au choix du temps de synchronisation $Tps_{synchro}$ pour le support du logiciel natif non annoté. Le temps de simulation consommé entre deux synchronisations avait été calculé de manière à ne pas manquer d'interruptions. On pouvait cependant, dans le cas d'une simulation fonctionnelle, se permettre de ne pas modéliser les temps de communication dans la plateforme matérielle et ainsi supprimer la partie dynamique du temps d'exécution du logiciel afin de garantir le fonctionnement de l'application. Le problème est alors de synchroniser la plateforme de simulation « au bon moment » , afin que le temps dynamique, qui viendra s'ajouter au temps statique calculé à partir des annotations accumulées dans la mémoire tampon, ne rendent pas le temps de synchronisation supérieur à celui séparant deux occurrences consécutives d'un même évènement matériel, car, par exemple des interruptions pourraient être manquées.

La granularité de synchronisation la plus fine correspond à ce niveau à celle des appels aux fonctions d'annotation. Une synchronisation systématique à chaque annotation de bloc n'est cependant pas envisageable. Malgré le fait qu'elle offrirait dans notre contexte la meilleur précision pour la prise en compte des évènements matériels, les performances de simulation obtenues ne seraient certainement pas acceptables.

Une solution possible consiste à effectuer un pré-traitement dans les fonctions d'annotation afin d'anticiper le calcul du temps de synchronisation, en accumulant les temps statiques déjà disponibles dans les annotations et faire une première estimation de la partie

dynamique en fonction par exemple du nombre d'accès en mémoire qui devront être réalisés au moment de la synchronisation. En plus des pré-traitements nécessaires, cela suppose que les latences du réseau de communication sont connues et plus ou moins uniformes en fonction des adresses accédées. Il n'y a pas pour autant de garantie sur le fait qu'aucune interruption ne sera manquée.

La solution que nous proposons est empirique et peut même sembler simpliste. Elle consiste à choisir une fréquence de synchronisation (en nombre d'appels aux fonctions d'annotation) suffisamment petite afin d'avoir une précision de prise en compte des évènements matériels plus fine tout en ayant des performances de simulation acceptables. C'est donc au développeur de l'application utilisant cette plateforme de simulation que nous demandons de faire ce choix. Des mécanismes peuvent cependant être mis en place dans les Unités d'Exécution afin de détecter que des interruptions ont été manquées alors qu'elles n'étaient pas masquées.

6.3 Profilage du logiciel

Bien qu'indispensable à l'exploration d'architecture, l'estimation globale des performances d'exécution du logiciel n'est pas suffisante en tant que telle pour assister les concepteurs dans leurs choix architecturaux. Le profilage du logiciel permettant d'affecter un temps d'exécution à chacune des fonctions de l'application fait partie de ces informations pouvant être fort utiles dans les phases d'exploration. Une des applications possibles à la technique d'instrumentation mise en œuvre dans ce travail consiste à insérer d'autres informations dans les blocs de base lors de la phase d'analyse afin de permettre d'attribuer plus précisément le temps consommé aux différentes fonctions exécutées.

Les informations ajoutées dans le cadre du profilage du logiciel sont tout simplement des identificateurs, permettant de déterminer la nature des blocs de base parmi les trois types suivants :

- Entrée de fonction
- Sortie de fonction
- Standard (ni entrée, ni sortie)

Tous les traitements effectués dans les EU lors des appels aux fonctions d'annotations et de synchronisation décrits précédemment restent inchangés. Après traitement, la mémoire tampon contenant la liste de blocs de base exécutés n'est cependant pas vidée immédiatement. Elle doit, avant cela être soit sauvegardée dans un fichier, soit traitée immédiatement par l'outil de profilage. Les deux options sont d'ailleurs possibles et réalisées dans une tâche (un *thread*) concurrente à celle gérant la simulation, permettant ainsi, avec l'utilisation de plusieurs mémoires tampons, de minimiser l'impact sur les performances de simulation lorsque la machine hôte dispose de plusieurs processeurs.

Qu'il soit effectué hors simulation ou à la volée, le profilage du logiciel est réalisé de la même manière. L'analyse de chaque bloc de base permet de gérer l'évolution des appels de fonctions ainsi que leurs coûts, pouvant être un temps, un nombre de cycles ou d'instructions, ou encore une consommation, à partir d'un ensemble de structures de données dont une représentation simplifiée est donnée dans la figure 6.6.

Une première liste ❶ contient les descripteurs de chaque bloc de base ayant déjà été exécutés. Lorsqu'un bloc est traité, son descripteur est inséré dans cette liste (s'il n'existe pas déjà). Lorsqu'un bloc correspond à l'entrée d'une fonction, un descripteur d'appel est

créé et inséré dans la liste d'appels du bloc « appelant » ❷. Un nouveau descripteur de contexte est également créé et empilé dans la pile de contextes d'appels ❸. Lorsqu'un bloc correspond à un retour de fonction, le contexte d'appel en sommet de la pile est dépilé.

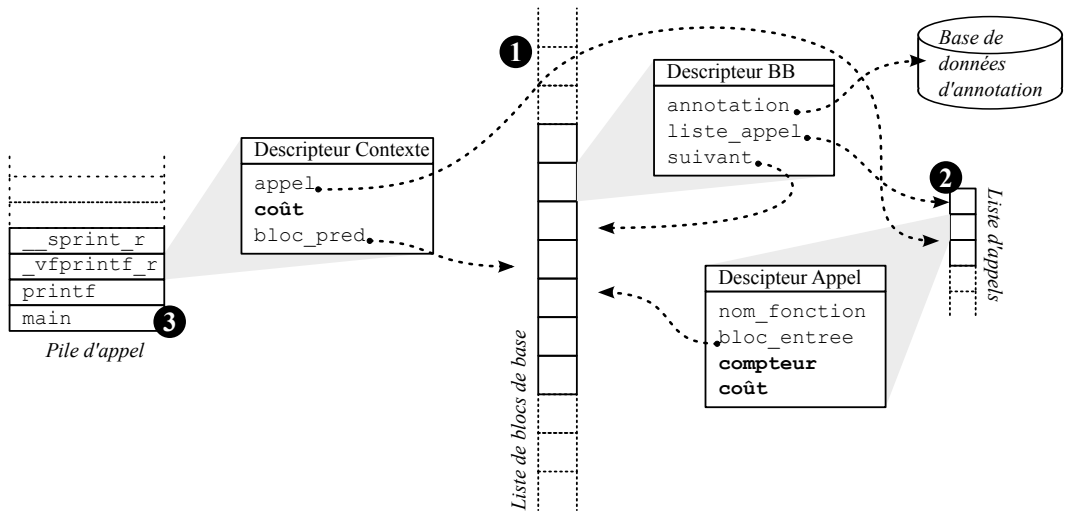


FIGURE 6.6 – Structure de données pour le profilage du logiciel

Le profilage de l'application logicielle est ensuite réalisé en calculant le cumul des coûts (temps, cycles, instruction, puissance, etc.) associés à chaque appel de fonction. Pour cela, un descripteur d'appel contient les champs *compteur* et *coût*. Le descripteur de contexte contient également un champ *coût*. Les cumuls sont ensuite gérés de la manière suivante :

1. Pour chaque bloc de base, le coût associé, qui sera tiré des informations d'annotations, est systématiquement ajouté au coût du contexte d'appel en sommet de pile.
2. Lors d'un appel de fonction, le compteur du descripteur associé est incrémenté.
3. Lors d'un retour de fonction, le contexte en sommet de pile est dépilé et son coût est ajouté à celui du nouveau sommet de pile ainsi qu'à celui du descripteur d'appel de fonction correspondant.
4. Lorsque la simulation se termine, la pile de contexte est vidée tout en continuant à gérer le cumul des coûts.

La construction du fichier au format *callgrind* consiste ensuite à parcourir la liste des blocs de base et des appels de fonctions pour en extraire tous les coûts. En pratique, on peut également gérer des coûts au niveau des descripteurs des blocs de base afin d'avoir une granularité plus fine. Il est également possible d'associer chaque fonction et bloc de base à une ligne dans un fichier source pour permettre aux développeurs de directement visualiser les coûts dans leur code d'origine. Durant la simulation, les Unités d'Exécution permettent de connaître les tâches de l'application en cours d'exécution, et ainsi de gérer une structure de données de profilage pour chacun d'entre elles. Ce profilage permet donc d'analyser en détail l'exécution d'un programme multi-tâches ayant été exécuté sur une architecture multiprocesseur de type SMP ou AMP.

La figure 6.7 représente le graphe d'appel de la fonction `IDCT()` utilisée dans une application de décodage JPEG ayant été exécutée sur deux processeurs. Dans cet exemple, le coût de chaque fonction est donné en pourcentage du nombre d'instructions totales exécutées (Les fonctions dont le coût est inférieur à 1% ne sont pas représentées ici).

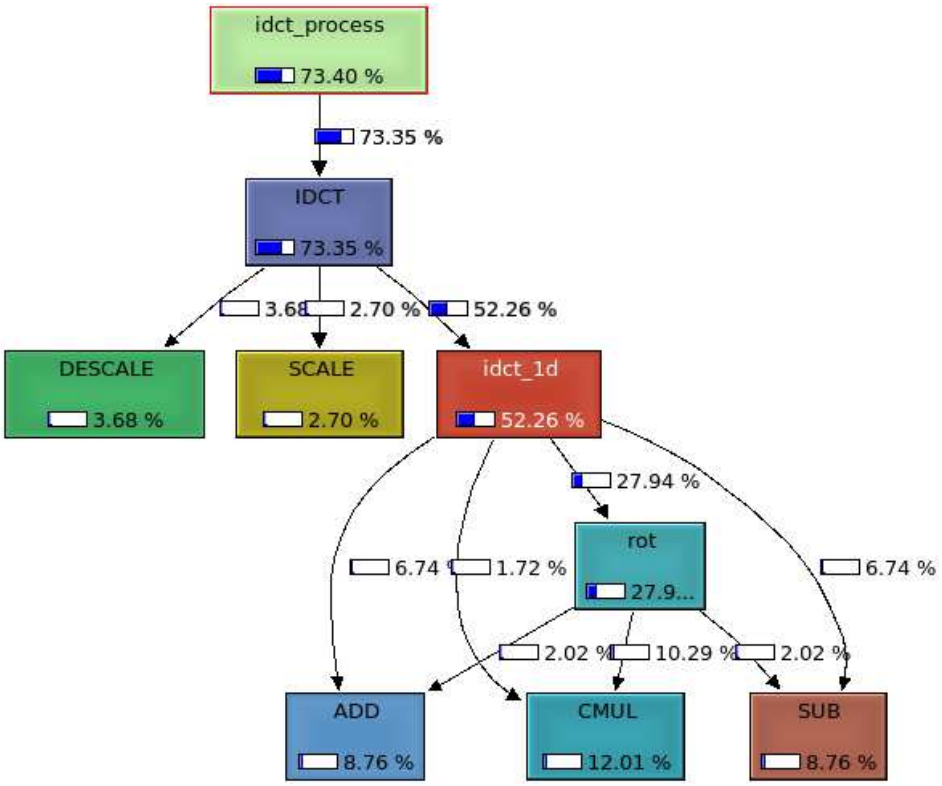


FIGURE 6.7 – Visualisation avec *kcachegrind* du profilage logiciel sur une application de décodage JPEG

Chapitre 7

Expérimentations

Sommaire

Les résultats des expérimentations menées sur les approches proposées dans cette thèse sont présentés dans ce chapitre. Dans la première partie, les méthodes de réalisation de plateformes matérielles au niveau TLM adaptées à la simulation native du logiciel seront mises en œuvre pour la modélisation d'architectures multiprocesseurs. Un exemple basé sur deux nœuds logiciels de type SMP servira de support d'expérimentation afin de mettre en évidence les capacités et les limitations de la solution proposée à modéliser des architectures complexes et réalistes. Les résultats obtenus dans cette partie seront donc principalement qualitatifs.

La deuxième partie présentera les résultats obtenus sur la technique d'instrumentation de code mise en œuvre, afin de valider la précision et la capacité de cette solution à refléter l'exécution du logiciel sur un processeur cible. Avant cela, nous commencerons par décrire les environnements logiciels et matériels sur lesquels se basent ces expérimentations.

7.1 Environnement matériel

Les environnements matériels utilisés dans le cadre de ces expérimentations sont basés sur deux bibliothèques de composants implémentées en SystemC. La bibliothèque SoCLib [SOC] permet de modéliser des architectures au niveau CABA. Cette modélisation nous servira de référence lorsque cela sera nécessaire. La bibliothèque fournit plusieurs simulateurs de processeur (ARM, SPARC, MIPS, MicroBlaze, etc.) ainsi qu'un réseau de communication « Generic Micro Network » (GMN), permettant d'interconnecter de nombreux processeurs, mémoires et autres périphériques utilisant le protocole Virtual Component Interface (VCI).

La deuxième bibliothèque a été développée dans le cadre de cette thèse afin de modéliser les architectures au niveau TLM en respectant les règles d'implémentation que nous avons proposées dans le chapitre 4.

7.1.1 Les composants de la bibliothèque TLM

Parmi les principaux composants mis à disposition dans cette bibliothèque se trouvent bien évidement les Unités d'Exécution permettant de modéliser les processeurs du système et implémentant le support du logiciel, ainsi que tous les périphériques classiques (mémoires, contrôleurs d'interruptions, etc.). Un composant d'interconnexion permet, en fonction de certains paramètres, de modéliser un bus, un *crossbar* ou encore un réseau sur puce NoC.

Chaque composant de cette bibliothèque dérive (au sens C++ du terme) de composants de type *maître* (`device_master`) et/ou *esclave* (`device_slave`), offrant des fonctionnalités de base, telle que la possibilité de paramétrer des cycles de latence en entrée ou sortie de chaque port TLM, permettant ainsi de modéliser finement les temps de communication sur une architecture matérielle. Ces composants sont schématisés sur la figure 7.1.

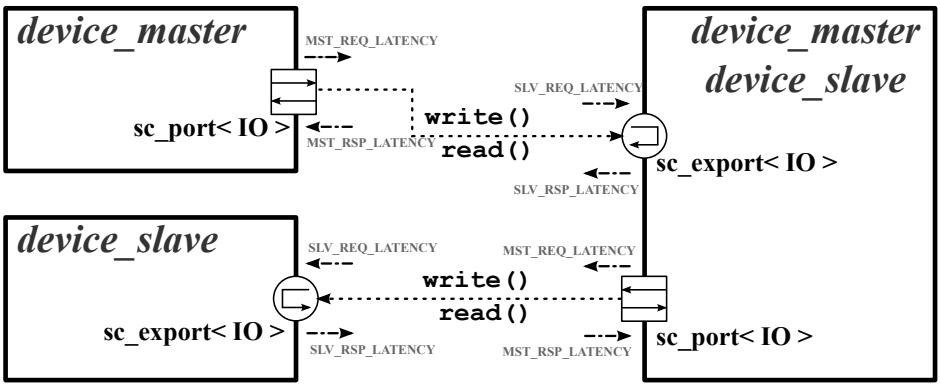


FIGURE 7.1 – Composants maîtres/esclave de base

Tout composant ayant au moins un port de communication esclave doit impérativement fournir deux méthodes utilisées lors de l'élaboration de la plateforme par les composants de communication et le *linker* décrits dans le chapitre 4. Ces deux méthodes sont les suivantes :

– `get_mapping()` : retournant une liste de régions de mémoire occupées par le composant esclave et définies en terme d'adresse de base, de taille et de nom. Ces méthodes seront appelées par les réseaux de communication, sur tous les composants esclaves connectés afin qu'ils puissent construire leur plan mémoire.

– `get_symbols()` : retournant une liste de symboles décrits par le couple nom et valeur. Ces symboles seront utilisés par le *linker* dynamique afin de résoudre les symboles non définis de l'application, lors de l'élaboration de la plateforme de simulation.

Une implémentation générique est fournie dans le composant de base `device_slave` avec un ensemble d'attributs permettant de modéliser et de gérer les registres internes pour la plupart des types de périphériques. Elles pourront être redéfinies dans les composants spécifiques pour lesquels elles ne conviennent pas.

7.1.2 L'interface de communication

Toutes les communications entre les différents composants se font uniquement par l'intermédiaire de canaux TLM offrant une vue volontairement bas niveau des accès en lecture et écriture (du type adresse plus donnée). Les ports de communication *maîtres* (`sc_port`) et *esclaves* (`sc_export`) sont tous basés sur l'interface SystemC `IO`, fournie dans la bibliothèque TLM mise en œuvre et définissant les méthodes de communications. Le listing 7.1 donne la définition de l'interface `IO` en SystemC.

Cette interface contient en plus des méthodes de lecture et d'écriture simples (lignes 3 à 9), des méthodes vectorisées (lignes 11 à 13) permettant de modéliser les mécanismes de *Burst* de certaines architectures. Ces méthodes peuvent être vues comme la modélisation des instructions *Load/Store* des processeurs. Certains processeurs fournissent également des instructions plus complexes pour réaliser des accès atomiques aux mémoires partagées dans des environnements parallèles. Les instructions *Load-Linked* et *Store-Conditional* permettent de réaliser ces accès atomiques, sans pour autant bloquer l'accès à la mémoire pour les autres processeurs jusqu'à la fin de l'opération [1]. L'interface `IO` fournit également ce type d'accès par les méthodes `load_linked()` (ligne 15) et `store_cond()` (ligne 16), et sont supportées par la plateforme de simulation TLM.

Listing 7.1 Interface de communication SystemC utilisée dans le modèle proposé

```
1  struct IO:public sc_interface
2  {
3    virtual void read (uint8_t  *addr, uint8_t  *data);
4    virtual void read (uint16_t *addr, uint16_t *data);
5    virtual void read (uint32_t *addr, uint32_t *data);
6
7    virtual void write (uint8_t  *addr, uint8_t  data);
8    virtual void write (uint16_t *addr, uint16_t data);
9    virtual void write (uint32_t *addr, uint32_t data);
10
11   virtual void vector_write (uint8_t  *to, uint8_t  *from, uint32_t len);
12   virtual void vector_write (uint16_t *to, uint16_t *from, uint32_t len);
13   virtual void vector_write (uint32_t *to, uint32_t *from, uint32_t len);
14
15   virtual uint32_t load_linked (uint32_t *addr);
16   virtual bool store_cond (uint32_t *addr, uint32_t data);
17 };
```

7.1.3 Configuration des composants

Les paramètres des composants sont passés par des fichiers de configuration comme celui présenté dans le listing 7.2. Ce fichier permet de configurer les paramètres de chaque

1. On parle alors de techniques *Lock-Free*. L'instruction *Load-Linked* charge un mot de la mémoire, après traitement, l'instruction *Store-Cond* sauvegarde la nouvelle valeur, seulement si la case mémoire concernée n'a pas été modifiée entre temps par un autre processeur.

composant individuellement ou de définir des paramètres globalement. Par exemple, les composants de base (et donc tous les composants de la bibliothèque) utilisent les paramètres `CLOCK_PERIOD` et `CLOCK_UNIT`, ce qui permet de définir les latences en terme de cycles d'horloge, plutôt qu'en temps. Les latences du composant *maître/esclave* nommé `crossbar` sont définies localement (lignes 10 à 13) alors que le composant *esclave* `timer_0` utilisera les paramètres de latence globaux (lignes 6 et 7).

Listing 7.2 Exemple de configuration d'un composant matériel

```
1  [] # Global parameters
2    unsigned long CLOCK_PERIOD = 10;
3    long          CLOCK_UNIT = 2;  #  0 => SC_FS    3 =>  SC_US
4                                    #  1 =>  SC_PS   4 =>  SC_MS
5                                    #  2 =>  SC_NS   5 =>  SC_SEC
6    long   SLV_REQ_LATENCY = 1;
7    long   SLV_RSP_LATENCY = 1;
8
9  [crossbar]
10   long   MST_REQ_LATENCY = 3;
11   long   MST_RSP_LATENCY = 1;
12   long   SLV_REQ_LATENCY = 5;
13   long   SLV_RSP_LATENCY = 2;
14   long   DEPTH = 3;
15
16 [timer_0]
17   char * IRQ_CONFIG = "1";
```

7.2 Environnement logiciel

Les applications utilisées dans les expérimentations reposent sur l'environnement logiciel APES [GP09], constitué de composants conçus afin de faciliter et d'accélérer le développement de logiciel embarqué sur des architectures multiprocesseurs hétérogènes. Les différents composants disponibles offrent un ensemble de fonctionnalités telles que la gestion de la mémoire, des communications ou encore des systèmes de fichiers. Les deux principaux éléments constituant APES sont le système d'exploitation DNA et son composant d'abstraction du matériel (la couche HAL).

DNA est un système d'exploitation dédié aux architectures multiprocesseurs embarqués offrant des mécanismes haut niveau tels que le support des files d'exécution (*thread*) compatible avec la norme POSIX, de sémaphores, de messages ou encore la gestion de la mémoire dynamique, tout cela avec une faible empreinte mémoire et un impact minimum sur les performances globales. L'intérêt d'utiliser l'environnement logiciel APES pour nos expérimentations est principalement lié à son système d'exploitation et plus particulièrement à l'API HAL sur laquelle il repose. L'utilisation stricte qui en est faite a grandement facilité le portage de ce système sur la plateforme de simulation.

Une seule modification du code source a été nécessaire afin de permettre son exécution native. La version initiale de DNA utilise directement les adresses de la zone de mémoire `.bss` (accessible aux adresses des symboles `__bss_start` et `_end`) contenant les données statiques ou globales à initialiser à zéro, définies par les outils de compilation lors de l'édition de lien. Ce code, incompatible avec la solution proposée (règle 4.6), a été remplacé par l'utilisation de pointeurs intermédiaires comme nous l'avions présenté dans la section 4.4. Le portage de DNA a ensuite consisté à implémenter une Unité d'Exécution et un *linker* dynamique spécifiques à DNA.

La pile logicielle construite à partir de APES pour nos expérimentations est représentée sur la figure 7.2. Elle nous permettra de supporter facilement des applications multi-tâches basées sur l'utilisation de *threads* POSIX. Les principaux composants qui la constituent sont la bibliothèque standard C Newlib de *Red-Hat* [NEW], les composant *VFS* permettant de gérer des systèmes de fichiers, une bibliothèque de communication *KPN* ainsi qu'un support pour la gestion de mémoire dynamique. La totalité de cette pile logicielle a été compilée pour le processeur *Intel x86* de la machine hôte utilisée pour les simulations.

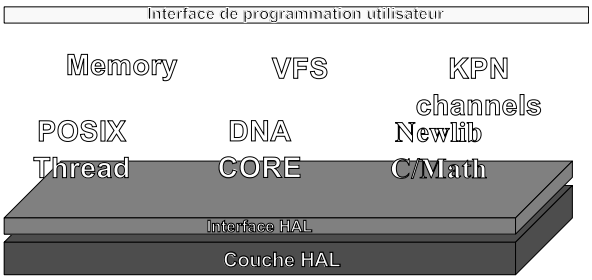


FIGURE 7.2 – Pile logicielle utilisée dans le cadre des expérimentations

7.2.1 L'Unité d'Exécution spécifique à DNA

Le portage du système d'exploitation DNA sur la plateforme native consiste à spécialiser le composant `eu_base` disponible dans la bibliothèque TLM mise en œuvre, afin d'y implémenter chaque fonction de la couche HAL. L'API HAL de DNA a été largement inspirée du HAL de eCos [ECO] mais adaptée aux architectures multiprocesseurs hétérogènes. Elle est composée de seulement 27 fonctions, contre 50 pour eCos et plus d'une centaine pour Linux.

Cette API est séparée en deux parties, l'une contenant les fonctions spécifiques à la plateforme et l'autre au processeur. Ces deux familles sont ensuite divisées en différentes catégories dont les principales fonctions sont décrites dans le tableau 7.1.

L'aspect très « bas niveau » de ce HAL est un élément clé permettant de maximiser la quantité de code logiciel qui pourra être simulé nativement. Le tableau 7.2 donne un aperçu du rapport entre le code qui ne sera pas exécuté nativement (colonne HAL) et le reste du logiciel, en terme de nombre de lignes de code source (mesuré avec l'outil *sloccount*). Pour la bibliothèque C, nous n'avons pris en compte que la partie indépendante des processeurs.

Processeur	HAL	DNA + pthread	NewlibC
ARM966	ansi C : 193 asm : 132	ansi C : 8152	ansi C : 55395
MIPS R3000	ansi C : 189 asm : 445		

TABLE 7.2 – Tailles des codes sources du HAL, de DNA et de NewlibC

7.2.2 Le *linker* dynamique pour DNA

Lorsqu'il est compilé pour un processeur cible, DNA a besoin d'un certain nombre d'informations permettant de le configurer. Parmi ces informations, on trouve par exemple le point d'entrée du système d'exploitation qui sera appelé après le démarrage du processeur ainsi que le point d'entrée de l'application. On trouve également la liste des pilotes de périphériques qui seront chargés lors de l'initialisation de DNA, ainsi que celle des systèmes

API Plateform		
Catégories	**Description**	**Fonction**
endian	Définition de l'*endianness* de référence de la plateforme	-
mp	Support multiprocesseur	`PLATFORM_MP_CPU_COUNT()`
API Processeur		
Catégories	**Description**	**Fonction**

context	Gestion des contextes d'exécution pour le support multi-tâches	`CPU_CONTEXT_INIT()`
		`CPU_CONTEXT_SAVE()`
		`CPU_CONTEXT_LOAD()`
		`CPU_CONTEXT_SWITCH()`

endian	Fonctions de conversion d'*endianness*	`CPU_DATA_IS_BIGENDIAN()`
		`CPU_DATA_IS_LITTLEENDIAN()`
		`CPU_PLATFORM_TO_CPU()`
		`CPU_CPU_TO_PLATFORM()`

io	Fonctions d'accès aux registres de contrôle des périphériques. Chaque fonction supporte divers types de données (8/16/32/64 bits, flottant)	`CPU_[READ|WRITE]()`
		`CPU_UNCACHED_[READ|WRITE]()`
		`CPU_VECTOR_[READ|WRITE]()`

mp	Support multiprocesseur, identification, synchronisation au démarrage	`CPU_MP_IS()`
		`CPU_MP_COUNT()`
		`CPU_MP_[WAIT|PROCEED]()`

synchro	Support de synchronisations atomiques	`CPU_TEST_AND_SET()` `CPU_COMPARE_AND_SWAP()`

trap	Gestion des interruptions et exceptions	`CPU_TRAP_ATTACH_[ESR|ISR]()`
		`CPU_TRAP_MASK_AND_BACKUP()`
		`CPU_TRAP_RESTORE()`
		`CPU_TRAP_[ENABLE|DISABLE]()`

TABLE 7.1 – Fonctions de l'API HAL utilisée par DNA

de fichiers. Dans l'environnement APES, ces paramètres de configuration sont fournis par l'intermédiaire du fichier de script utilisé lors de l'édition de liens par les outils de cross compilation.

Dans le cadre de la simulation native, ces informations ne peuvent pas être définies au moment de l'édition de liens faite par les outils de compilation car elles ne sont en partie connues qu'au moment de l'exécution (tout au moins celles relatives à des adresses de mémoire ou de périphériques). L'implémentation du *linker* dynamique spécifique à DNA a pour objectif de récupérer les paramètres du système d'exploitation à partir d'un fichier de configuration et d'affecter leurs valeurs directement dans l'application chargée dynamiquement en mémoire. Un exemple de configuration est donné dans le listing 7.3.

Chaque paramètre de ce fichier ainsi que la manière dont ils sont utilisés est spécifique au *linker* DNA. Le paramètre `CPU_OS_ENTRY_POINT` sera utilisé par exemple pour retrouver l'adresse du symbole `dna_kickstart` dans l'application, qui sera utilisée comme pointeur de fonction par l'Unité d'Exécution pour démarrer le système d'exploitation. D'autres paramètres permettent également de créer des régions de mémoire qui seront ensuite accessibles sur la plateforme de simulation par l'intermédiaire des composants modélisant les mémoires.

Le *linker* a également la responsabilité de récupérer tous les symboles définis par les composants de la plateforme matérielle à l'aide de la méthode `get_symbols()` décrite précédemment, puis d'affecter leurs valeurs aux symboles correspondant dans l'application logicielle. Ces symboles correspondent typiquement aux adresses de base des périphériques qui seront utilisés par les pilotes logiciels.

Listing 7.3 Exemple de configuration d'un *linker* dynamique DNA

```
1  [linker_dna_0]
2    char * APPLICATION = "./APP/app-native.x.0";
3    char * CPU_OS_ENTRY_POINT = "dna_kickstart";
4    char * APP_ENTRY_POINT = "_main";
5    long PLATFORM_N_NATIVE = 16;
6
7    char * OS_DRIVERS_LIST = "soclib_system_module rdv_module fdaccess_module";
8    char * OS_FILESYSTEMS_LIST = "devfs_module rootfs_module";
9    long OS_THREAD_STACK_SIZE = 0x8000;
10
11   char * OS_KERNEL_HEAP_SECTION = ".kheap";
12   char * OS_USER_HEAP_SECTION = ".uheap";
13   char * OS_KERNEL_BSS_SECTION = ".bss";
14
15   char * SECTIONS_DECL = ".kheap .uheap .sysstack
16   .global_mem";
17   char * SECTIONS_SIZE = "0x1000000 0x1000000 0x100000 0x10000";
18
19   long CHANNEL_RDV_NDEV = 16;
```

7.2.3 Applications : décodeur Motion-JPEG

Le Motion-JPEG (MJPEG) est un format multimédia ou chacune des trames d'une séquence vidéo sont indépendantes les unes des autres et compressées au format JPEG. L'application de décodage MJPEG utilisée ici est décomposée en trois tâches principales :

- *FETCH* lit les données à partir d'un fichier, répartit les informations vers les autres tâches et décompresse les blocs de pixels avant de les fournir aux tâches *IDCT*.
- *IDCT* effectue la transformée en cosinus discrète inverse. Le nombre de tâche IDCT est configurable afin d'augmenter le degré de parallélisme de l'application.
- *DISPATCH* est chargée de remettre en forme les données en provenance des *IDCT* afin de reconstruire l'image qui sera envoyée vers le composant matériel *framebuffer* permettant de visualiser la vidéo.

Chacune de ces tâches logicielles (représentées sur la figure 7.3) est implémentée dans un processus léger (thread) POSIX s'exécutant sur le système d'exploitation DNA.

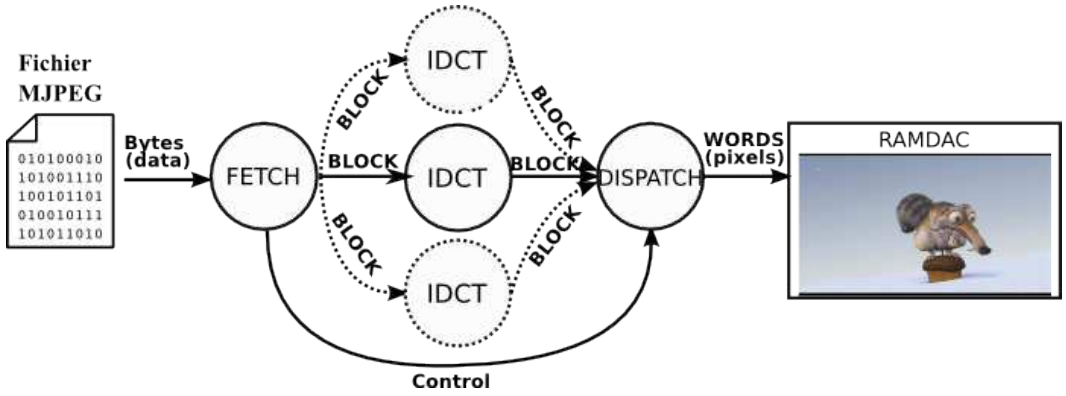


FIGURE 7.3 – Modèle fonctionnel de l'application MJPEG

7.3 La plateforme de simulation native : abstraire sans cacher

Le fait d'abstraire sans *cacher* les détails de l'architecture matérielle permet de faire une analogie directe entre la plateforme de simulation au niveau TLM proposée ici et des plateformes équivalentes réalisées au niveau CABA ou même réelles. Les expérimentations menées dans cette section visent à mettre en évidence la précision de la plateforme de simulation native en terme de « réalisme », et ce malgré l'absence totale d'annotation temporelle dans le code du logiciel.

Les similitudes avec un environnement réel débutent avant même la simulation, lors de l'élaboration de la plateforme de simulation.

7.3.1 Élaboration de la plateforme

L'architecture du système utilisé ici est représentée sur la figure 7.4. Elle est composée de deux nœuds logiciels, chacun contenant les périphériques classiques des sous systèmes processeurs (contrôleurs d'interruptions, mémoires, *timers*) ainsi que des périphériques plus spécialisés, tels qu'un contrôleur d'accès à un système de fichier (permettant d'utiliser des fichiers sur la machine hôte), un *framebuffer* permettant de visualiser une image en mémoire ainsi qu'un *bridge* pour les accès extérieurs au nœud logiciel. Le nombre de processeurs contenu dans chacun de ces nœuds est configurable et un réseau d'interconnexions externe (*crossbar*) offre un accès à une mémoire globale et un terminal d'affichage partagé.

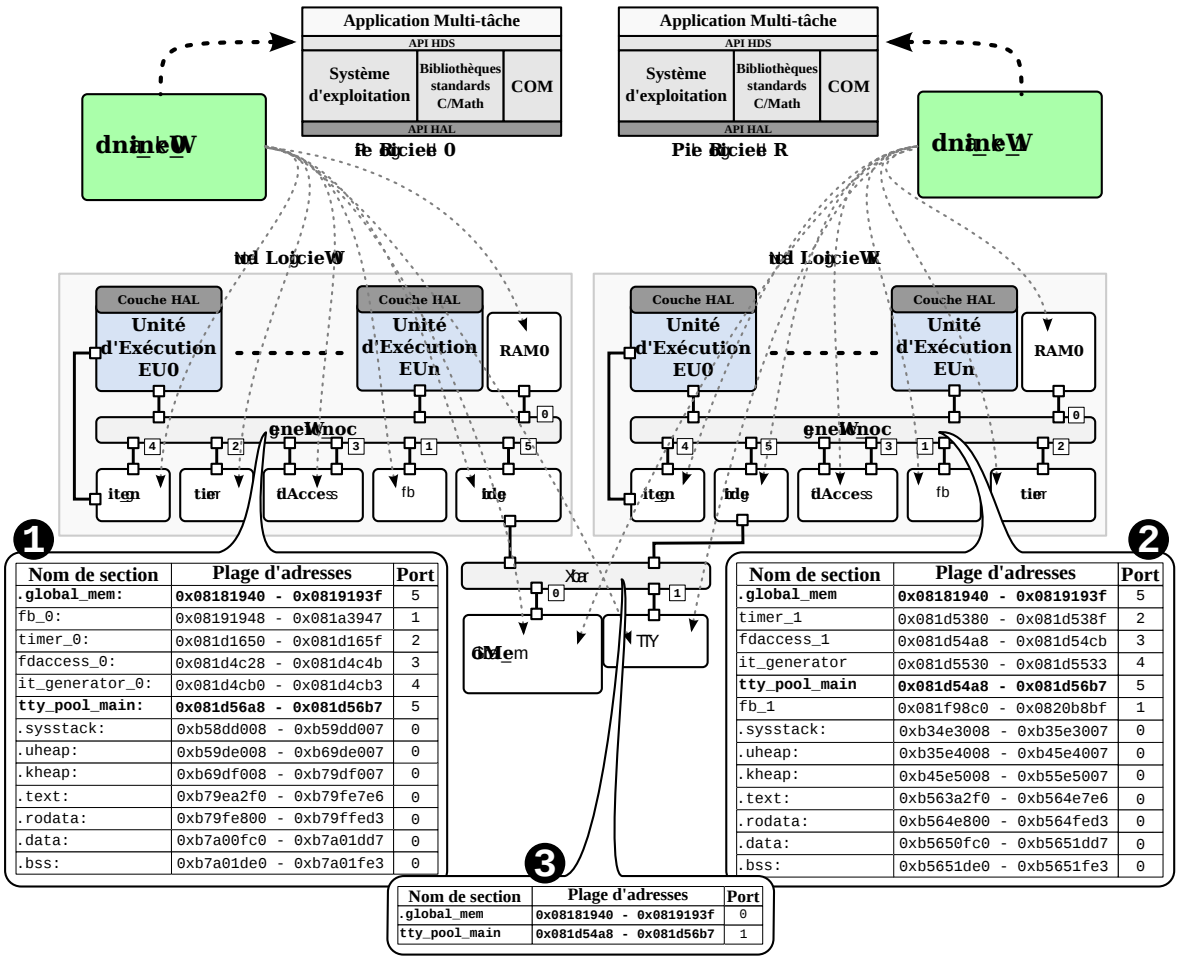


FIGURE 7.4 – Exemple d'une architecture à deux nœuds logiciels SMP

La première similitude que nous souhaitons mettre en évidence ici est celle de la représentation mémoire uniforme entre l'application logicielle et la plateforme matérielle. Lors de l'élaboration de la plateforme, chaque composant ayant un rôle de communication est chargé de construire lui même son propre plan mémoire, à partir des informations fournies par les périphériques connectés. Dans cette plateforme, les trois plans mémoires associés aux trois réseaux de communication sont représentés. Dans les plans mémoire ❶ et ❷, on peut voir la présence des sections de l'application `.text`, `.data` ou encore `.bss` ainsi que les zones occupées par les différents périphériques. On notera également que les zones occupées par la mémoire globale et le terminal dans le plan mémoire ❸ sont également visibles dans les plans mémoires ❶ et ❷, mais dans lesquels ils correspondent bien au port d'accès vers le composant *bridge*. Chacune des plages d'adresses décrites ici sont valides du point de vue des applications logicielles, qui pourront y accéder sans aucune contraintes particulières. Rappelons toutefois que, sans instrumentation du code logiciel, seules les communications engendrées par des appels aux fonctions HAL dans le logiciel peuvent être visibles sur la plateforme matérielle. Les accès directs à la mémoire dans le logiciel sont supportés mais ne génèrent aucune communication sur les réseaux.

La deuxième phase de l'élaboration correspond au chargement des applications et à la résolution de symboles non définis. On peut faire l'analogie avec l'édition de lien qui aurait été réalisée par les outils de compilation dédiés au processeur cible. Tous les symboles et régions des mémoires définies dans le fichier de script utilisé par l'éditeur de liens du processeur cible devront également êtres résolues par le *linker* dynamique de *DNA* sur la plateforme de simulation native.

Un *linker* DNA est associé à chaque nœud logiciel et connecté à chaque composant susceptible de définir des symboles pour l'application (ces connections représentées par des pointillés ne sont utilisées que lors de l'élaboration). L'utilisation du terminal *TTY* donne un exemple simple de définition de symbole attendu par la pile logicielle. Dans le listing 7.3 du fichier de configuration de DNA, nous avons spécifié au système de charger le pilote de périphérique `soclib_system_module`, permettant de contrôler certains périphériques disponibles dans la bibliothèque CABA SoCLib, dont un terminal *TTY*. L'implémentation du composant *TTY* utilisé ici est donc équivalente du point de vue des registres modélisés et du comportement au *TTY* disponible dans SoCLib, ce qui permet d'utiliser le même pilote logiciel sans aucune modification du code source.

Sur une architecture cible, le pilote associé au composant *TTY* utilise une adresse de base définie par le pointeur `SOCLIB_TTY_DEVICES`. Ce symbole sera donc défini au niveau TLM par le composant *TTY* et sa valeur correspondra à l'adresse de base de la zone mémoire allouée pour ces registres. Cette zone, nommée `tty_pool_main`, est visible dans les trois plans mémoires car elle est accessible par tous les réseaux de communications. L'adresse du symbole `SOCLIB_TTY_DEVICES` sera donc égale à $0x081d54a8$ (pour cette simulation uniquement). Une fois l'application chargée, le *linker* utilisera les symboles définis par les composants pour affecter leurs valeurs dans le code logiciel. Le listing 7.4 correspond à la trace d'affichage générée par le *linker* DNA du nœud logiciel 0 lors de l'élaboration de la plateforme utilisée en exemple. Parmi les symboles de l'application mis à jour, on peut noter (lignes 9 et 10) que le pointeur dont le symbole est `SOCLIB_TTY_DEVICES` se trouvant à l'adresse $0xb7a01444$ de l'application est bien affecté avec l'adresse de base $0x081d54a8$ de la zone mémoire occupée par le *TTY*.

Au final, toutes les zones mémoires utilisées sur une plateforme réelle trouveront une zone correspondante dans la plateforme de simulation native. Si tous les composants sont implémentés de manière fidèle par rapport à leur équivalent CABA ou réel, alors ces zones mémoire seront de même taille, seules les adresses de base seront différentes.

Listing 7.4 Trace de résolution des adresses par le *linker* dynamique de DNA

```
1  linker_dna_0 start dynamic link
2  link: SOCLIB_FB_BASE
3  *0xb7a01564 = 0x8191948
4  ...
5  link: OS_GLOBAL_MEM_ADDRESS
6  *0xb7a01370 = 0x8181940
7  link: SOCLIB_TTY_NDEV
8  *0xb7a01420 = 0x1
9  link: SOCLIB_TTY_DEVICES
10 *0xb7a01444 = 0x81d54a8
```

7.3.2 Mécanismes de communication

Le réalisme de la plateforme matérielle obtenue lors de l'élaboration se transmet naturellement à l'exécution de la simulation. Le premier aspect que nous souhaitons présenter ici est la capacité de la plateforme à modéliser précisément les mécanismes de communication. Pour cela, nous avons utilisé une simple application écrivant la fameuse chaine de caractère « *Hello World* » (`printf("Hello World from proc[\%d]\n",CPU\_MP\_ID());`) sur le *TTY*. Cette application est exécutée simultanément sur les deux nœuds logiciel contenant un seul processeur chacun.

Les tracés de signaux sur la figure 7.6 sont pris sur les ports identifiés de la plateforme représentée sur la figure 7.5 et correspondent à l'écriture d'un caractère effectuée par les deux processeurs sur le *TTY*. Bien que les communications entre les composants se fassent au niveau TLM, ceux-ci supportent la génération de trace dans SystemC, permettant ainsi d'afficher les arguments des fonctions d'accès en lecture ou écriture sous une forme familière du point de vue matériel. Nous avons associé à chacun des ports des composants différentes latences d'accès de manière plus ou moins arbitraire.

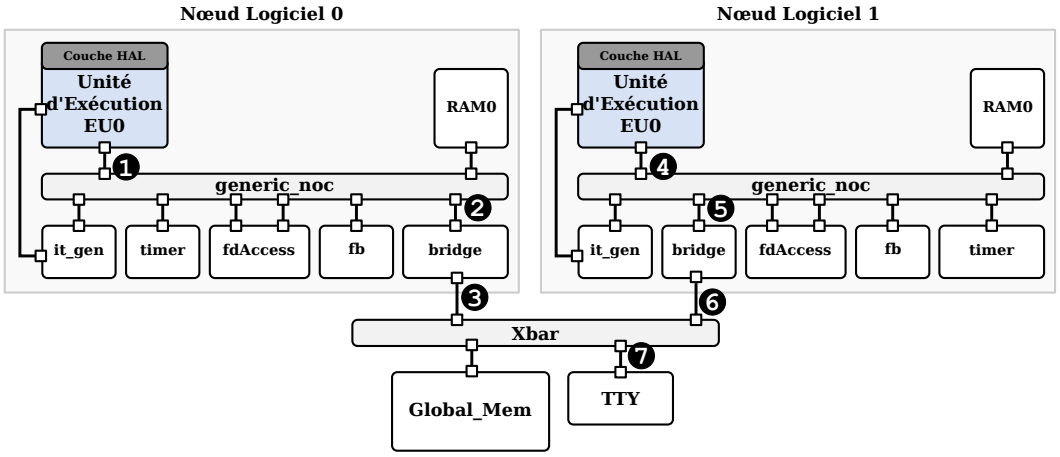


FIGURE 7.5 – Architecture de la plateforme matérielle

Le processeur du nœud logiciel 1 est le premier à faire une requête d'écriture ❹ à l'adresse $0x08209620$ du *TTY* (l'adresse est différente de celle de l'exemple précédent car elle change potentiellement à chaque exécution de la simulation). On peut ensuite suivre le cheminement de cette requête en ❺, jusqu'au port d'accès du *crossbar* en ❻. Celle-ci se trouve alors mise en attente car la requête issue du processeur du nœud logiciel 0 a traversé les différents composants (❶, ❷ et ❸) plus rapidement en raison des latences configurées. On peut ensuite voir la prise en compte des deux requêtes l'une après l'autre en ❻.

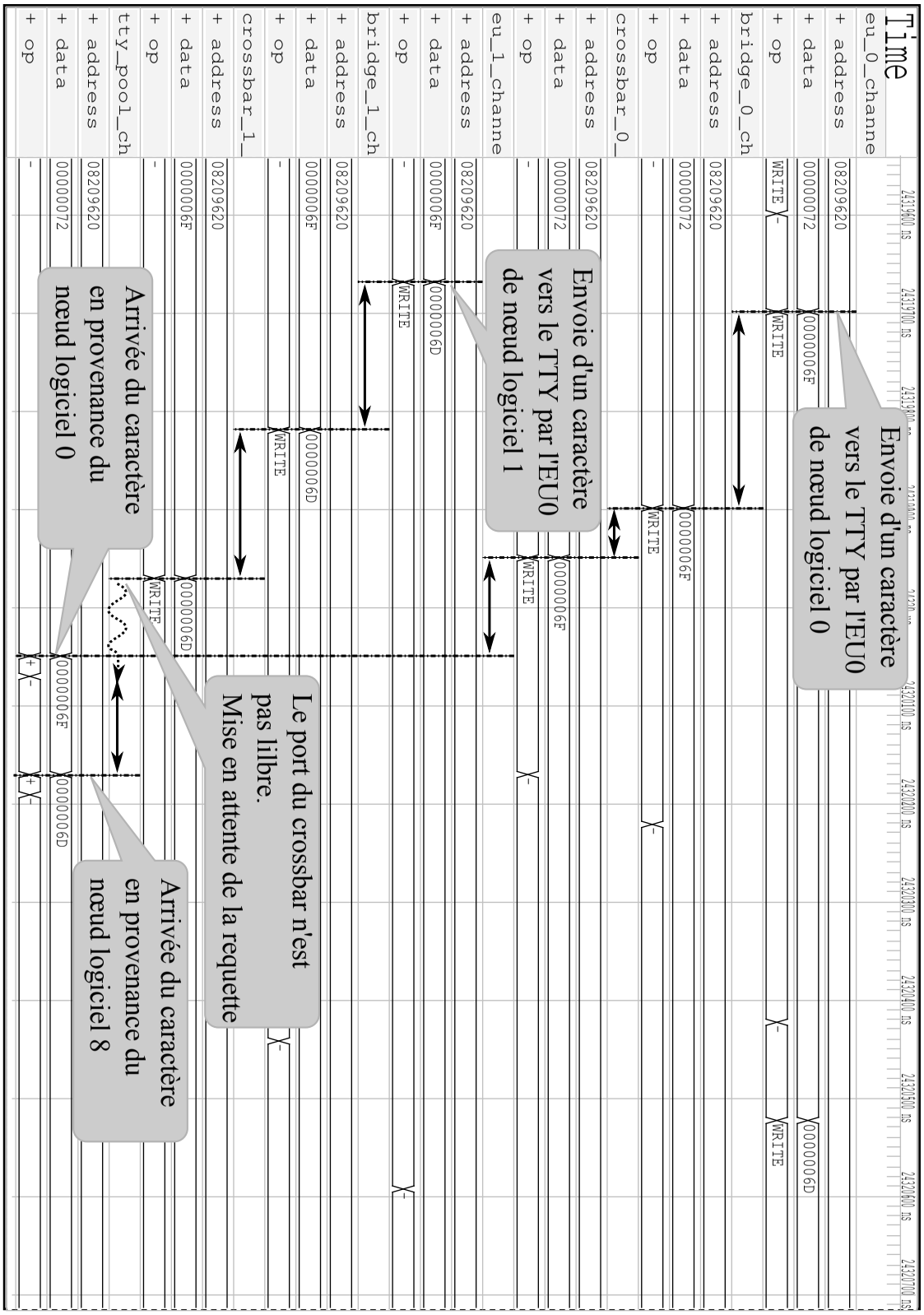


FIGURE 7.6 – Modélisation des communications sur les réseaux de la plateforme TLM

Cette simulation nous permet de mettre en évidence deux aspects importants de la solution proposée. Le premier en terme de précision de la modélisation aussi bien du point de vue du matériel et notamment des communications dans la plateforme, que du code logiciel exécuté puisque malgré sa simplicité, cet exemple met en œuvre une quantité relativement importante de code puisqu'il s'exécute tout de même sur le système d'exploitation DNA et nécessite le support de la bibliothèque C pour afficher la chaine de caractère.

Le deuxième aspect concerne les capacités de validation offertes par cette plateforme. Nous avons vu ici l'utilisation des traces de signaux, pouvant aider au débogage des modèles SystemC des composants matériels utilisés dans la plateforme. Ces signaux peuvent bien entendu être utilisés également pour visualiser les ressources internes des composants, telles que les registres. La partie comportementale ainsi validée pourra être réutilisée de manière assez directe sur des plateformes telles que SoCLib au niveau CABA (sous réserve d'une implémentation « propre », faisant la distinction entre communication et comportement).

Étant donné que l'ensemble de la plateforme et des piles logicielles s'exécute du point de vue de la machine hôte dans un même processus, nous pouvons bénéficier très simplement des outils de débogage et de validation utilisés dans le domaine du logiciel. Il est possible de cette manière de mettre un point d'arrêt à l'aide d'un outil de débogage tel que GDB, n'importe où dans le code de l'application (système d'exploitation, application, etc.), mais aussi dans le code implémentant les composants matériels de la plateforme.

Nous avons ainsi mis un point d'arrêt dans le composant *TTY*, à l'endroit où celui-ci traite la réception d'un caractère. L'ensemble de la pile d'appel des fonctions jusqu'à ce point d'arrêt a ensuite été obtenu (commande *backtrace* de GDB) et elle est donnée dans le listing 7.5. La ligne 1 correspond à celle où a été placé le point d'arrêt, dans le code source du composant *TTY*. Cette pile d'appel nous donne le cheminement de la requête d'écriture dans le *TTY* du point de vue logiciel. En « remontant » cette pile, on peut suivre le chemin inverse de la requête dans la plateforme matérielle (sans les informations temporelles). On peut par exemple voir son passage dans le *crossbar* à la ligne 5, dans le *bridge* à la ligne 9 et enfin dans le réseau de communication à la ligne 13. Toutes ses fonctions, bien que visibles du point de vue du logiciel font partie du domaine matériel de la plateforme. Si l'on continue de remonter dans cette pile d'appels, on trouve à la ligne 16 la fonction `__dnaos_hal_write_uint8()` correspondant à l'implémentation de la primitive HAL `CPU_WRITE_UINT32()` réalisée dans les EU spécifiques à DNA (le nom de la primitive HAL n'apparaît pas car il s'agit d'une *macro* C.

Cette fonction marque la frontière entre le matériel de la plateforme et le logiciel de l'application. Ainsi, si l'on poursuit dans la pile d'appel, on se trouve maintenant dans le logiciel en cours d'exécution. La fonction `soclib_tty_write()` à la ligne 17, est celle fournie par le pilote de périphérique associé au *TTY*, sur laquelle a été redirigé la demande d'écriture par les fonctions `devfs_write()` et `vfs_write()` de DNA. La fonction `write()` correspond à la dernière étape du traitement de la chaine de caractères dans la bibliothèque C (lignes 20 à 26). L'application n'est composée ici que de la fonction `main()` (ligne 27), qui est gérée dans le contexte d'un *thread* (`thread_wrapper`) de DNA. La première fonction de la pile d'appel (ligne 30) a été appelée par la primitive d'initialisation de contexte implémentée dans la partie HAL des EU. La fonction POSIX `makecontext` est fournie par le système d'exploitation de la machine hôte. Nous avions justement pris cet exemple d'utilisation de ressources de la machine hôte pour implémenter les EU, dans la section 4.2.1.1 à la page 44.

Listing 7.5 Pile d'appel complète d'un *printf* jusqu'au composant *TTY*

```
 1  libta::tty_pool::slv_write ()                \ <-- Point d'arret dans le TTY
 2  libta::device_slave::write ()                |
 3  libta::dev_channel::write ()                 |
 4  libta::device_master_slave::mst_write ()     |
 5  libta::generic_noc::slv_write ()             | <-- port d'entree du crossbar
 6  libta::device_master_slave::write ()         |
 7  libta::dev_channel::write ()                 |
 8  libta::device_master_slave::mst_write ()     |
 9  libta::bridge::slv_write ()                  | <-- Port d'entree du brigde
10  libta::device_master_slave::write ()         |
11  libta::dev_channel::write ()                 |
12  libta::device_master_slave::mst_write ()     |
13  libta::generic_noc::slv_write ()             | <-- Port d'entree du reseau
14  libta::device_master_slave::write ()         |
15  libta::dev_channel::write ()                /
16  __dnaos_hal_write_uint8 () _________________> <-- Fonction HAL de l'EU
17  soclib_tty_write ()                          \ <-- Pilote peripherique TTY
18  devfs_write ()                               |
19  vfs_write () _______________________________/
20  write ()                                     \
21  _write_r ()                                  |
22  _fflush_r ()                                 |     Fonction
23  __sfvwrite_r ()                               >    de la bibliothÃ¨que
24  __sprint_r ()                                |     standard C
25  _vfprintf_r ()                               |
26  printf ("Hello World from proc[%d]\n")_______/
27  main () ____________________________________> <-- L'application
28  __libthread_start ()                         \
29  thread_wrapper (p_signature=0xb69b3fd4) _____/     Systeme d'exploitation
30  makecontext () from /lib/tls/i686/cmov/libc.so.6   Creation d'un contexte
31  ?? ()                                              dans le HAL
```

7.3.3 Représentation de la mémoire uniforme

Un des objectifs de l'uniformisation de la représentation mémoire présentée dans le chapitre 4 était de permettre la modélisation des interactions entre le logiciel et le matériel sans aucune contrainte sur l'écriture du code source de l'application. Nous avions alors donné l'exemple d'un transfert DMA, qui n'est pas réalisable sans modification du code source sur une plateforme de simulation native utilisant deux espaces mémoire distincts.

La plateforme utilisée ici ne dispose pas de DMA, mais le contrôleur de fichier *fdaccess* permettant de lire l'entrée vidéo est confronté au même problème. L'ouverture d'un fichier se fait en donnant directement le pointeur de la chaine de caractère contenant le nom au contrôleur, qui ira lui-même lire cette chaine en mémoire. Cette adresse doit donc être valide aussi bien dans l'espace mémoire vu par le logiciel que celui vu par le matériel, ce qui est le cas sur notre plateforme. Les signaux représentés sur la figure 7.7 sont ceux obtenus lors de l'ouverture du fichier « *movie.mjpeg* » par l'appel à la fonction `fopen("movie.mjpeg","r");` de la bibliothèque C standard, sans avoir eu recourt à aucune fonction de conversion d'adresse.

Quand il reçoit la demande d'ouverture, le pilote du périphérique *fdaccess* configure les registres internes du contrôleur avec l'adresse, la taille de la chaine de caractères contenant le nom du fichier et enfin le registre de commande pour démarrer l'opération ❶. Ces accès sont visibles sur le port *esclave* du contrôleur (`fdaccess_0_channel_slv`), on peut ainsi voir que cette chaine de caractères se trouve en mémoire à l'adresse $0xb79ed492$. Il va ensuite lui-même faire les accès en lecture sur son port *maître* ❷ directement vers l'adresse mémoire configurée. On peut ainsi voir que les accès entre le port *maître* du contrôleur et

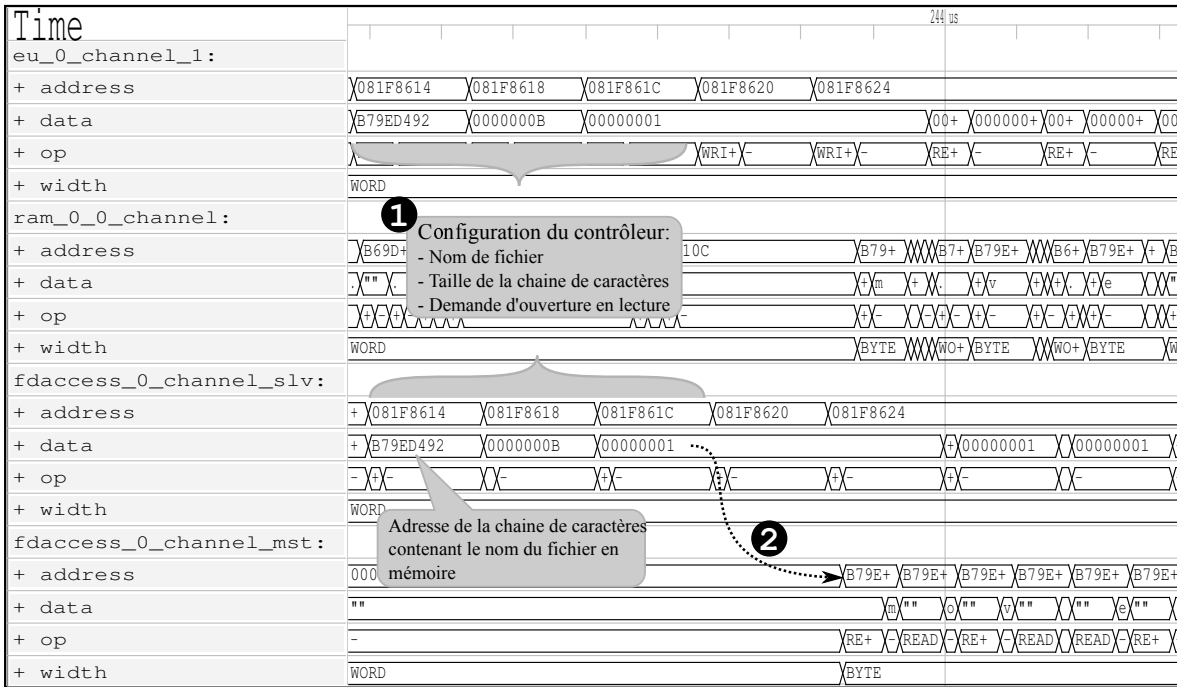


FIGURE 7.7 – Adresses mémoire unifiées entre le logiciel et le matériel

le port *esclave* de la mémoire sur l'agrandissement des signaux ❸ (figure 7.8) se font bien à l'adresse $0xb79ed492$. Ceux-ci sont d'ailleurs entrelacés avec les accès concurrents à la mémoire effectués par les processeurs de la plateforme.

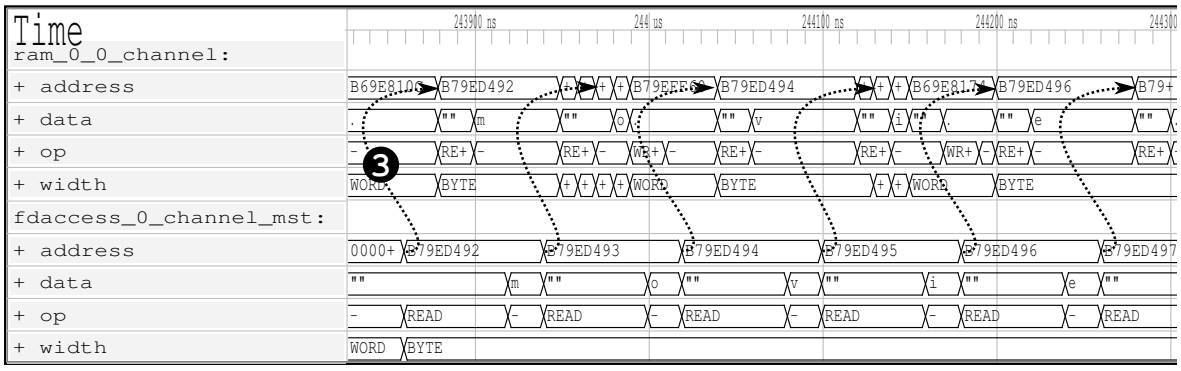


FIGURE 7.8 – Agrandissement de la figure 7.7

Cet exemple valide l'utilisation d'une seule et même représentation mémoire pour le logiciel et le matériel, et met une nouvelle fois en évidence la capacité à utiliser directement le code source d'origine du logiciel.

7.3.4 Support multiprocesseur SMP et interruptions

La représentation uniformisée de la mémoire ainsi que les mécanismes de synchronisation bas niveau (*Load-Linked*, *Store-Conditional*) supportés par la plateforme matérielle permettent également de modéliser précisément des architectures multiprocesseurs symé-

triques.

La plateforme de simulation utilisée jusqu'à présent est maintenant configurée avec 4 processeurs sur le nœud logiciel 0 et deux processeurs sur le nœud 1, chacun exécutant l'application MJPEG, l'une cadencée à 25 images par secondes et l'autre à 50 (l'utilisation d'évènements temporisés avec du logiciel non instrumenté est présentée dans la suite). Afin de fournir suffisamment de charge de calcul au nœud 0, le nombre de tâches IDCT de l'application MJPEG est passé à trois.

La norme POSIX prévoit la possibilité de nommer un *thread* en utilisant le champ *name* de la structure `pthread_attr_t`. Étant donné que la gestion des contextes d'exécution est prise en charge dans la couche HAL (implémentée ici par les EU), il est possible d'instrumenter de manière non intrusive l'exécution de ces *threads* sur les EU pour récupérer leurs noms (s'il sont fournis par l'utilisateur) ou toutes autres informations. Dans la figure 7.9, le nom des *threads* en cours d'exécution sur chacune des EU est représenté sur les mêmes chronogrammes utilisés pour tracer des signaux matériels.

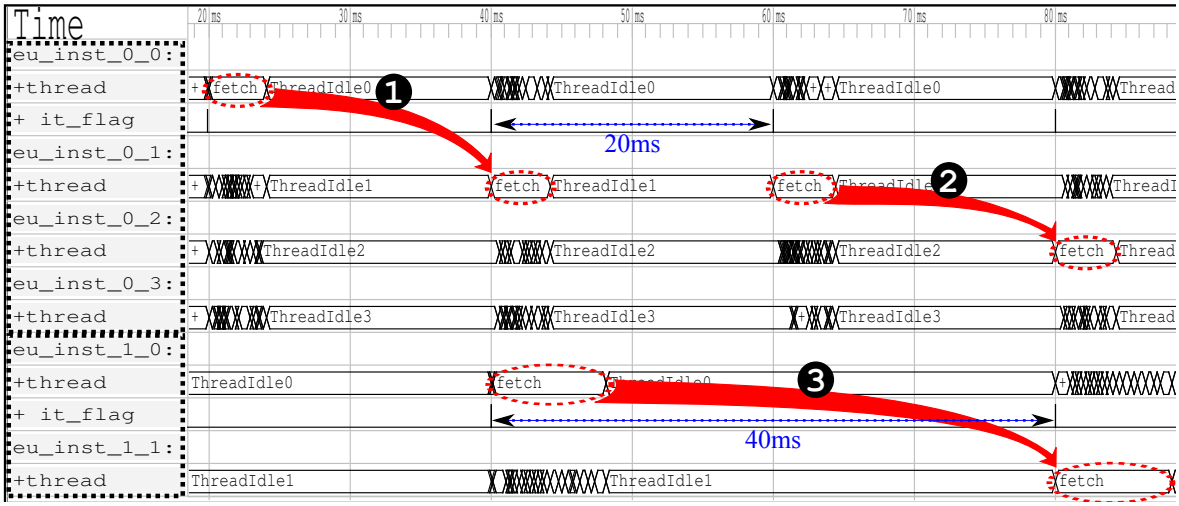


FIGURE 7.9 – Migration des *threads* entre les EU d'un même nœud logiciel

Ce nouvel aspect des capacités de débogage offertes par la plateforme de simulation native nous permet également de montrer le réalisme d'exécution de l'application multi-tâches sur une architecture SMP. Il est en effet possible de suivre sur la figure 7.9 la migration des tâches entre les différents processeurs d'un même nœud logiciel du système. La tâche la plus visible à cette échelle et dans cet exemple est la tâche *fetch* car elle nécessite un temps de traitement plus important et s'exécute donc entièrement sans qu'elle ne soit mise en attente. Elle est mise en attente sur les interruptions permettant de cadencer le taux de rafraichissement de l'image et n'est donc susceptible de migrer entre les processeurs qu'à ce moment là. C'est le cas sur la figure en ❶ et ❷ ou elle migre du processeur 0 au processeur 1, puis du processeur 1 au processeur 2 du nœud logiciel 0. On peut aussi voir cette migration en ❸ sur le nœud logiciel 1.

7.3.5 Temps de synchronisation en simulation non instrumentée

Dans le chapitre 4, nous avons présenté une solution permettant à une application non instrumentée de prendre en compte des évènements matériels (et donc dépendants

du temps). Cette solution est basée sur le choix d'un temps de synchronisation $Tps_{synchro}$ utilisé à chaque appel de fonction du HAL afin de faire évoluer le temps de simulation. Nous avions alors présenté le choix de cette constante à partir de l'équation suivante :

$$0 < Tps_{synchro} \leq \frac{1}{\sum_{i \in I} \frac{nb\_appels\_hal_i}{T_i}} \tag{7.1}$$

L'objectif ici est de montrer par des expérimentations comment cette valeur peut influencer et être influencée par l'application et la plateforme de simulation. Pour cela, nous utiliserons l'application MJPEG sur une architecture à un seul processeur. Un *Timer* configurable permet de déclencher une interruption à intervalles réguliers, qui elle même déclenchera à son tour le décodage d'une image complète. Cette interruption sera déclenchée toutes les $40ms$ afin de simuler un taux de rafraichissement classique de 25 images par secondes.

La première étape consiste donc à déterminer la valeur maximale du nombre d'appels au HAL réalisés par le logiciel pour décoder une image. Nous avons obtenu ce nombre par profilage de l'application sans utiliser les interruptions en prenant une valeur arbitraire pour $Tps_{synchro}$ (ce qui est possible lorsqu'il n'y a aucun évènement matériel), sur un ensemble de 10 images. La valeur maximale obtenue est pour ce jeu de mesures de 66780 appels. On peut donc déterminer la valeur maximale de $Tps_{synchro}$ comme suit :

$$\begin{aligned} Tps_{synchro} &= \frac{1}{\sum_{i \in I} \frac{nb\_appels\_hal_i}{T_i}} \\ &= \frac{1}{\frac{66780}{40 \times 10^{-3}}} = 598.98 \times 10^{-9} \end{aligned} \tag{7.2}$$

La première expérimentation consiste ici à faire varier le temps de synchronisation $Tps_{synchro}$ jusqu'à ce que celui-ci ne permette plus à l'application de fonctionner correctement, c'est-à-dire lorsqu'elle ne parvient pas à traiter toutes les interruptions en temps voulu. Sur la figure 7.10, chaque mesure en fonction de $Tps_{synchro}$ est faite sur une séquence de 10 images JPEG. Les premières interruptions non traitées apparaissent à partir de $Tps_{synchro} = 600$, ce qui correspond effectivement avec la valeur théorique déterminée précédemment.

Les performances de simulation pour décoder les 10 images de la séquence utilisée en fonction de la valeur de $Tps_{synchro}$ sont également représentées sur cette figure. On voit très nettement l'intérêt de ne pas choisir une valeur de synchronisation trop petite si l'on souhaite obtenir de bonnes performances de simulation. Cependant, ces premiers résultats pourraient mener à penser qu'il suffit de choisir une valeur légèrement inférieure à la valeur maximale de $Tps_{synchro}$ pour garantir le fonctionnement correct de la simulation.

Ceci reste vrai seulement si la plateforme matérielle ne modélise pas le temps. En effet, comme nous l'avons déjà évoqué dans la section 4.2.1.2, le temps engendré par les latences de communication sur le réseau d'interconnexion de la plateforme n'est pas pris en compte dans le calcul de $Tps_{synchro}$. Il est par ailleurs difficile à estimer en pratique, car il ne dépend pas seulement des communications effectuées par une EU, mais de toutes les communications engendrées aussi bien par les EU que par les autres composants matériels du système.

La figure 7.11 montre la dérive de la valeur maximale du temps de synchronisation en fonction de la latence configurée sur le réseau de communication. Ces valeurs sont obtenues en faisant varier $Tps_{synchro}$ pour chaque configuration du réseau, jusqu'à ce que des interruptions ne soient pas traitées. Lorsque les communications ne consomment aucun

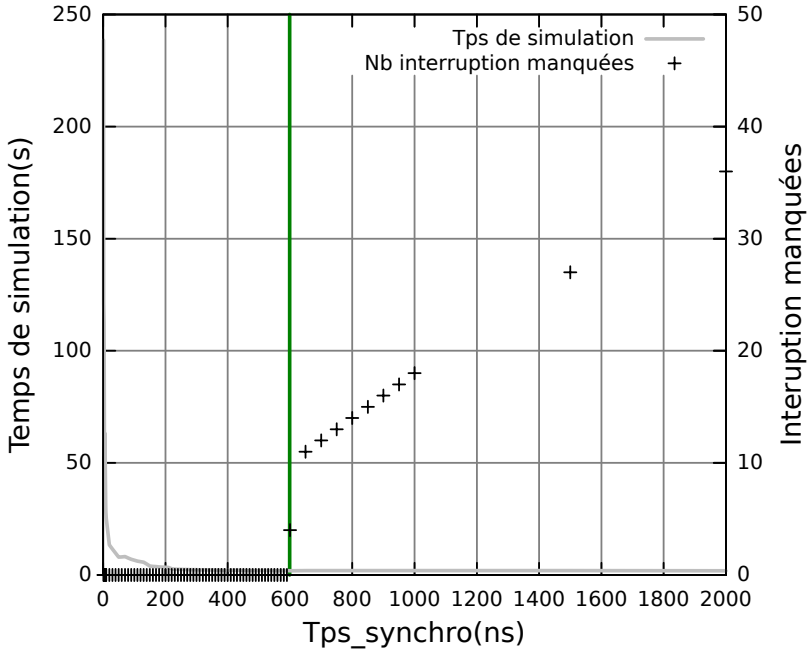


FIGURE 7.10 – Mesure du temps de synchronisation sur une plateforme sans latences matérielles

cycle d'horloge, la valeur maximale de $Tps_{synchro}$ permettant à l'application de s'exécuter correctement est de $590ns$, ce qui reste cohérent avec la valeur théorique trouvée précédemment (la granularité de mesure étant ici de $10ns$). Plus la part de temps consommé par le réseau de communication représente une part importante du temps de traitement d'une image, plus la valeur maximale de $Tps_{synchro}$ permettant à l'application de s'exécuter correctement diminue.

On peut également remarquer que même pour des temps de synchronisation très petits, les temps de simulation restent faibles par rapport au pic que nous pouvions constater dans la figure 7.10, et quasi constants. Cela s'explique par la méthode utilisée pour mesurer l'effet de la latence sur le temps de synchronisation. Contrairement à des modèles au niveau CABA où le temps de simulation est directement lié au temps simulé, les performances de simulation au niveau TLM natif peuvent varier de manière très importante en fonction de l'application. Par exemple, si toute l'application est en attente (*thread idle*), aucune communication n'est engendrée sur le réseau de communication et le temps simulé n'avance que par pas de $Tps_{synchro}$. Si ce temps est faible, ce qui est nécessaire lorsque la latence du réseau augmente, le temps simulé dans la boucle *idle* nécessitera un temps de simulation pouvant être très supérieur à celui nécessaire pour simuler des tâches « *actives* ».

Les mesures effectuées dans la figure 7.11 sont prises pour des configurations « *limites* » ou aucune interruption n'a été manquée. Entre chaque traitement d'interruption, l'application reste donc très peu de temps en attente dans le *thread idle* et les performances de simulation sont donc très raisonnables.

7.3.6 Performances de simulation

Comme nous venons de l'évoquer, les performances de la plateforme de simulation native sont très dépendantes à la fois de l'application simulée, de la modélisation du temps dans la plateforme matérielle ainsi que de la configuration du temps de synchronisation et

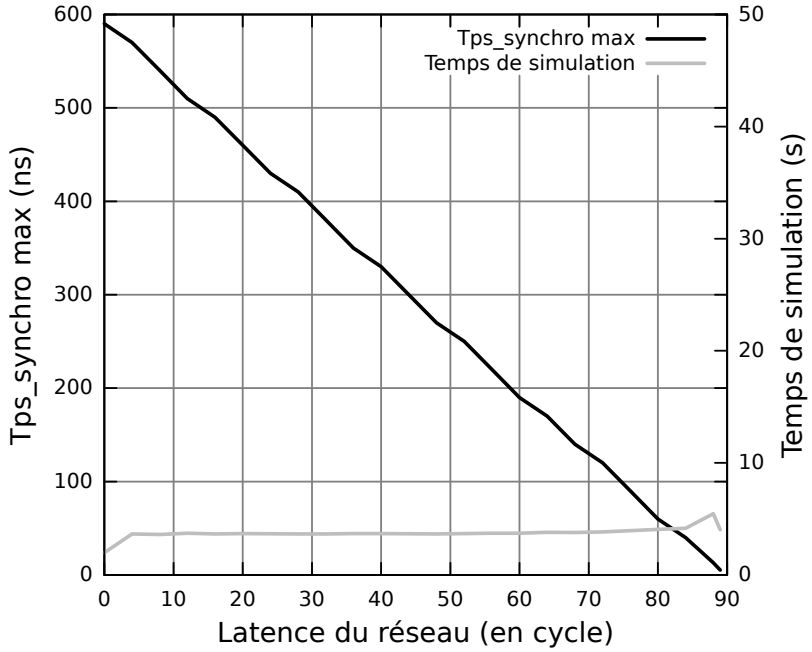


FIGURE 7.11 – Temps de synchronisation maximum en fonction de la latence du réseau de communication

peuvent varier de manière très importante. Les performances obtenues sur une plateforme de simulation qui ne serait pas purement fonctionnelle sont donc difficilement comparable avec d'autres méthodes ou modèles de simulation.

Le seul moyen de garantir une estimation assez stable des performances de la plateforme est donc d'utiliser un modèle matériel ne faisant pas apparaitre les temps de communication dans le système, et ne prenant pas en compte les évènement tels que les interruptions. Dans ce cas, il nous est possible d'obtenir des performances de simulation relativement stables, quelque soit la valeur du dernier paramètre configurable de la simulation qui est le temps de synchronisation. Le tableau 7.3 donne les performances obtenues sur l'application MJPEG exécutée sur un seul nœud logiciel dont le nombre de processeur varie entre 1 et 8. Ces temps de simulation sont donnés pour un temps de synchronisation de 1ns, sachant qu'ils restent similaires quelque soit sa valeur. Ces résultats doivent cependant être considérés comme des performances maximales pouvant être atteintes avec la plateforme de simulation. Elles sont comparées avec celles obtenues au niveau CABA, en utilisant le simulateur SystemCASS [SYSb], offrant un facteur d'accélération de simulation d'environ $\times 7$ par rapport à SystemC.

	Nombre de processeurs/EU					
Plateforme	1	2	3	4	6	8
Native	1.984	3.201	4.356	5.544	8.161	10.629
CABA (Mips)	359.8	251.0	264.0	346.5	480.5	618.1
Accélération	$\times 181$	$\times 78$	$\times 60$	$\times 62$	$\times 58$	$\times 58$

TABLE 7.3 – Temps de simulation en seconde pour la plateforme native et la plateforme CABA utilisant SystelCass

7.4 Instrumentation du code logiciel

Pour ces expérimentations, nous nous focaliserons principalement sur la précision de la technique d'instrumentation relative à l'architecture interne du processeur pour le *ARM966*. Le choix de ce processeur s'est imposé de lui même en raison d'un certain nombre de contraintes liées à la fois à la chaine de compilation LLVM et aux processeurs disponibles au niveau CABA afin de réaliser nos plateformes de référence.

LLVM est aujourd'hui un outil relativement jeune et peu d'architectures sont pleinement supportées. La compilation de l'ensemble de la pile logicielle pour les processeurs *MIPS* et *SPARC* s'est soldée par une quantité importante de plantage du compilateur LLVM, notamment lors de la compilation de la bibliothèque standard C. Seul la compilation pour le processeur *ARM* de l'ensemble de la pile logicielle a été réalisée. Les quelques fichiers qui n'ont pas pu être compilés avec LLVM pour le processeur *ARM* ont été compilés avec la chaine de compilation GCC GNU et ne sont donc pas instrumentés. Ils correspondent à des fonctions non utilisées dans notre application et ne viendront donc pas entamer la précision des résultats obtenus.

Le modèle de processeur *ARM966* au niveau CABA de la bibliothèque SoCLib est basé sur un ISS fourni par UniSim [ACG+07]. Celui-ci n'offrant au moment des expérimentations qu'une précision au niveau instruction, la prise en compte des cycles d'horloge liés à l'architecture interne du processeur a donc du être implémentée et intégrée dans ce simulateur. Le simulateur a été également instrumenté afin de pouvoir obtenir des traces de simulation nous permettant d'extraire les informations de comparaison dont nous avons besoin (nombre d'instructions exécutées, nombre de cycles, etc.).

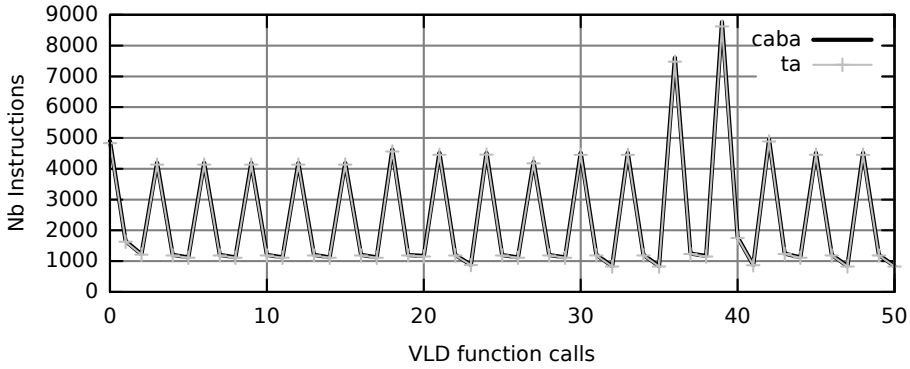
Afin de fournir un environnement logiciel complet, nous avons fait le choix de construire une chaine de compilation destinée à l'instrumentation du code, ce qui nécessite l'instrumentation de la bibliothèque standard C (NewlibC) ou des fonctions *builtins* de GCC. Malheureusement, ces bibliothèques contiennent du code assembleur spécifique au processeur cible et donc non compatible avec notre solution. En ce qui concerne la bibliothèque standard C, l'utilisation de code spécifique au processeur cible a été désactivée et remplacée par l'implémentation générique disponible en langage C. Les fonctions *builtins* de GCC n'ont quant à elles pas été instrumentées.

De manière à mettre en évidence la capacité de l'approche d'instrumentation proposée à refléter l'exécution du CFG cible réel, nous utiliserons la fonction VLD (pour Variable Length Decoder) utilisée dans la décompression de l'application MJPEG. Cette fonction est tout à fait pertinente pour cette expérimentation car son flot d'exécution est très fortement dépendant des données traitées.

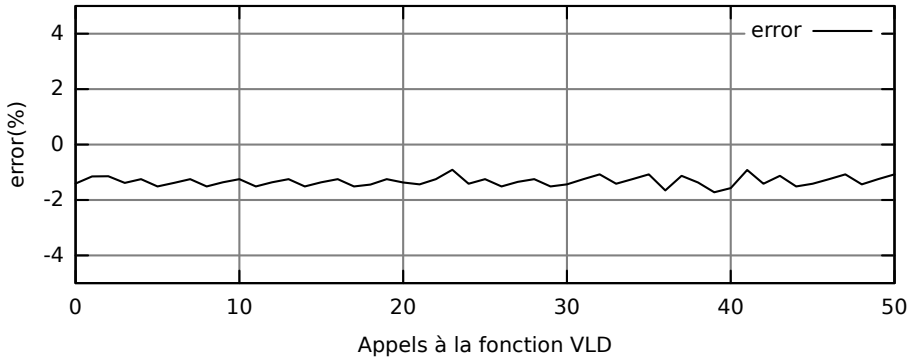
7.4.1 Estimation du nombre d'instruction

La première expérimentation présentée ici consiste à comparer l'estimation du nombre d'instructions exécutées pour chaque appel à la fonction VLD obtenue sur la plateforme de simulation native avec la valeur de référence obtenue sur la plateforme CABA. La figure 7.12(a) trace ce nombre d'instructions en fonction de l'ordre d'appel à la fonction VLD (l'appel n°1 correspond au traitement du premier macro bloc de l'image, l'appel i correspond au traitement du i^{me} macro bloc). Les résultats obtenus sont très proches et parfaitement superposés sur la courbe. L'erreur d'estimation donnée sur la courbe 7.12(b) montre que celle-ci est inférieure à 2%

En théorie, l'erreur d'estimation devrait être nulle en ce qui concerne le nombre d'instructions exécutées car nous disposons du code assembleur du processeur cible final lors



(a) Nombre d'instruction exécutées à chaque appel à la fonction VLD



(b) Erreur d'estimation par rapport au modèle CABA

FIGURE 7.12 – Estimation du nombre d'instruction

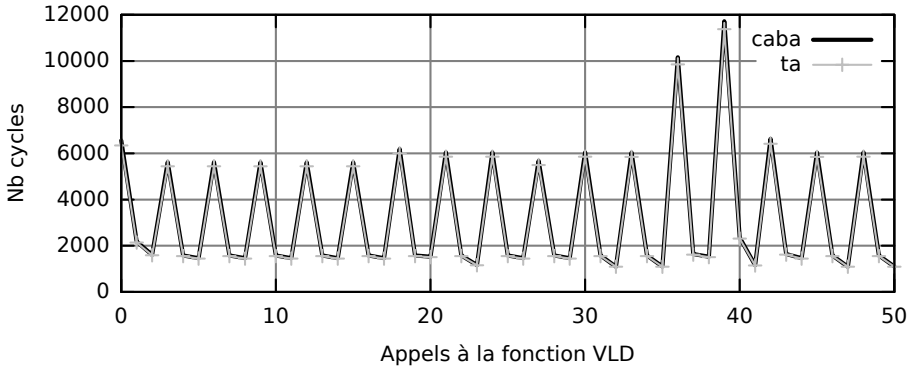
de la passe d'annotation. Cette erreur est due au fait que malgré tout, la totalité du code logiciel n'est pas instrumenté. La première source de code logiciel non instrumenté se trouve dans la couche HAL spécifique au processeur *ARM966*, qui n'est pas disponible car elle est fournie par les EU de la plateforme de simulation. Bien que cette erreur soit inévitable dans notre approche de simulation, elle ne représente qu'un infime partie du code logiciel et reste donc relativement faible. Il serait cependant préférable de la minimiser en associant à chaque fonction HAL implémentée dans les EU un nombre pré-calculé d'instructions obtenu en fonction du processeur cible. Ces valeurs pourraient être fournies très simplement aux EU en utilisant les fichiers de configuration présentés en début de ce chapitre (un par modèle de processeur).

La seconde source d'erreurs provient des fonctions *builtins* de GCC. Cette erreur est difficilement évitable dans notre approche.

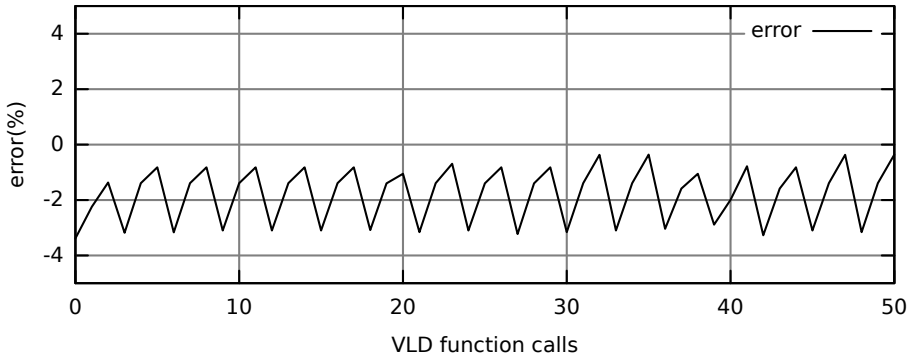
7.4.2 Estimation des cycles d'horloges

Bien que l'estimation précise du nombre de cycles nécessaires à l'exécution du programme ne fasse pas partie des contributions à proprement parler de cette thèse, nous tenions à faire quelques expérimentations afin d'avoir un premier aperçu des résultats que l'on pourrait espérer obtenir dans de futurs travaux.

Nous avons ainsi implémenté une première fonction d'analyse ne prenant en compte que les cycles de latences supplémentaires liés aux opérandes et aux interdépendances



(a) Nombre de cycles exécutés à chaque appel à la fonction VLD



(b) Erreur d'estimation par rapport au modèle CABA

FIGURE 7.13 – Estimation du nombre de cycles d'horloge sans prendre en compte les pénalités de branchement

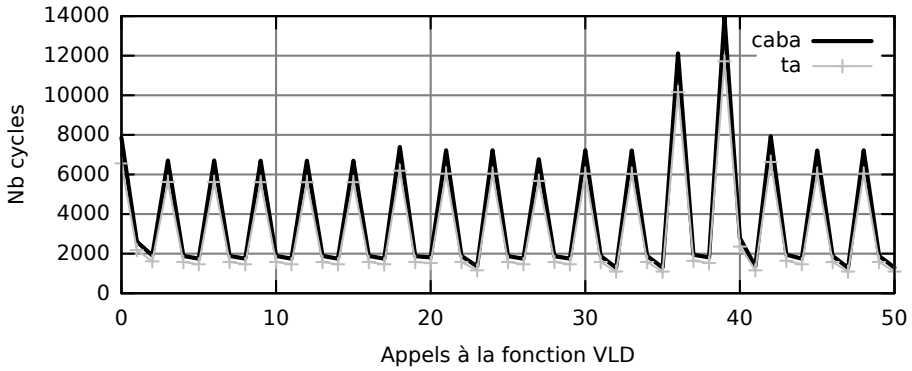
entre instructions. Afin d'obtenir dans un premier temps la précision de cette analyse uniquement, nous avons modifié le simulateur du processeur *ARM* au niveau CABA afin de ne pas prendre en compte les cycles de pénalités dues au branchement qui devraient s'ajouter à ceux pris en compte dans l'analyse. La figure 7.13(a) donne cette fois ci l'estimation en nombre de cycles d'horloge nécessaires à l'exécution de chacun des appels à la fonction VLD. Là encore, les deux courbes correspondent relativement bien, et l'erreur d'estimation, donnée dans la figure 7.13(b) se situe autour de 2%.

La dernière expérimentation a simplement pour objectif de mettre en évidence l'erreur due à l'absence de prise en compte des pénalités de branchement. Sur la figure 7.14(a), on voit maintenant que les deux courbes ne se superposent plus parfaitement et que le fait d'ignorer ces pénalités dans l'implémentation actuelle entraine une sous estimation du nombre de cycles d'horloge de l'ordre de 15% à 20%.

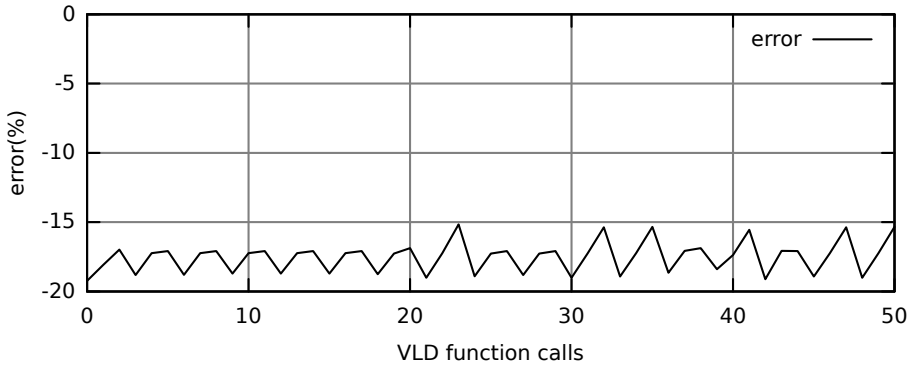
Les expérimentations que nous avons menées nous ont permis de montrer la capacité de la technique d'instrumentation proposée à suivre l'exécution du flot de contrôle du programme comme si celui ci c'était réellement exécuté sur le processeur cible.

Bien que l'exemple d'application proposé dans le chapitre 6 n'ait pu être implémenté qu'en partie dans le cadre de cette thèse, les expérimentations effectuées en ce sens peuvent nous laisser penser que la prise en compte de pénalités de branchement devraient nous mener à des précisions proches de celles que nous avons obtenues en les ignorants. En effet, les informations sur le CFG dont nous disposons au moment de l'analyse sont relativement

complètes et devraient permettre d'instrumenter la quasi totalité des branchements.



(a) Nombre de cycles exécutés à chaque appel à la fonction VLD



(b) Erreur d'estimation par rapport au modèle CABA

FIGURE 7.14 – Estimation du nombre de cycles d'horloge avec prise en compte des pénalités de branchement sur le modèle CABA

Chapitre 8

Conclusion

L'objectif global de cette thèse par lequel nous avions conclu le chapitre de problématique était de proposer une méthode de simulation pour les systèmes multiprocesseurs sur puces, alliant la précision des approches TLM pour la modélisation des plateformes matérielles avec la vitesse de simulation offerte par les approches basées sur l'exécution native du logiciel.

Nous avions alors présenté un tableau récapitulatif du classement des différents niveaux d'abstraction du matériel en fonction de leurs compromis entre vitesse de simulation et précision de modélisation, ainsi que la quantité de code logiciel pouvant être validé sur chacun de ces modèles.

Ce tableau, représenté à nouveau ci-dessous, peut aujourd'hui être enrichi d'une nouvelle colonne intermédiaire réunissant la simulation native du logiciel et la modélisation TLM *classique* du matériel.

	Précision				*Performances*
Niveau de modélisation du matérielle	CA	TLM-classique	***TLM-classique***	TLM-natif	SL
Validation de la partie logicielle	Pile logicielle complète	Pile logicielle complète	***Pile logicielle indépendante du matériel***	Pile logicielle indépendante du matériel	Tâches applicatives

FIGURE 8.1 – Positionnement de l'approche proposée par rapport aux autres modèles de simulation

Atteindre cet objectif global a été possible en apportant des réponses à un certain nombre de questions que nous avions soulevées lors de l'analyse des problèmes liés à l'exécution native du logiciel. Dans la première partie de ce chapitre de conclusion, nous dresserons le bilan de ce travail en reprenant chacune des questions auxquelles nous souhaitions répondre pour en faire le bilan. Nous terminerons ce manuscrit par les perspectives que ce travail de recherche a permis de définir.

8.1 Bilan

La première série de questions concernait principalement la plateforme de simulation native et plus particulièrement les problèmes liés à l'exécution du logiciel :

- Comment exécuter une pile logicielle complète de manière native sur une plateforme de simulation TLM ?
- Comment faire en sorte de maximiser la quantité de code source final réutilisé tout en minimisant les contraintes de codage ?
- Dans quelle mesure est-il possible de prendre en compte les détails architecturaux au niveau du logiciel natif, tels que l'exécution parallèle du logiciel, les représentations mémoire ainsi que les évènements matériels ?

Les deux premières questions avaient partiellement trouvées des réponses dans les travaux basés sur la simulation native du logiciel au dessus de la couche HAL et dont nous nous sommes inspirés. Nous avons mis en évidence que l'utilisation de cette API HAL n'est pas une condition suffisante pour permettre l'exécution complète de la pile logicielle sans aucune modification de code. C'est finalement en répondant à la troisième question que nous avons pu, dans ce travail, apporter une solution permettant de prendre en charge la totalité de la pile logiciel (au dessus de la couche HAL) tout en minimisant les contraintes d'écriture de code. En effet, le manque de "réalisme" des approches de simulation native existantes ne permet pas de réutiliser directement le code logiciel et plus particulièrement les couches situées juste au dessus du HAL (système d'exploitation, pilotes de périphériques, bibliothèque standard C, etc.) en raison des problèmes d'incompatibilité entre les espaces mémoires vues pas le logiciel et le matériel.

L'utilisation d'une représentation mémoire unifiée nous permet de réutiliser la totalité du code logiciel au dessus de la couche HAL. Nous avons pour cela défini une méthode de conception des composants TLM modélisant le matériel adaptée à la simulation native, ainsi qu'une technique d'édition dynamique de liens similaire à celle effectuée dans les outils de compilation, permettant d'initialiser les adresses non résolues au moment de l'élaboration de la plateforme.

Les contraintes de codages du logiciel concernent uniquement la manipulation d'adresses relative au matériel et sont résolues par l'utilisation de pointeurs intermédiaires. Il est peu probable que le portage d'un système d'exploitation existant ne requière aucune modification. Ceci dit, les contraintes de codage du logiciel nécessaires à l'exécution native restent valides dans le code compilé pour le processeur cible. Ainsi une fois les adaptations faites, le même code source du logiciel peut directement être réutilisé.

L'espace mémoire vu par le logiciel correspond à celui modélisé par la plateforme matérielle. Ainsi, si l'on souhaite réellement utiliser le même code logiciel, il est nécessaire de modéliser une plateforme matérielle réaliste vis-à-vis de l'architecture réelle. Cela est possible en utilisant les Unités d'Exécution comme *unique* support pour le logiciel, permettant ainsi de modéliser le matériel avec des techniques TLM classiques pouvant être relativement précises en terme d'architecture.

Nos expérimentations ont montrés les capacités de cette approche à modéliser de manière fidèle l'architecture matérielle. Nous avons également montré comment cette plateforme peut grandement améliorer la validation fonctionnelle des applications en permettant de modéliser les évènements matériels tels que les interruptions, et ce malgré l'absence de temps dans le logiciel exécuté.

La deuxième contribution de cette thèse avait pour objectif de définir une méthode d'instrumentation automatique du code logiciel. Nous avons pour cela fait un certain nombre de choix techniques afin de répondre aux questions posées initialement :

- À quel niveau instrumenter le code logiciel (source, binaire, ...) ?
- Comment refléter l'exécution du logiciel sur le processeur cible ?
- Comment prendre en compte ces informations dans une plateforme de simulation native ?

L'instrumentation au niveau de la représentation intermédiaire des compilateurs est une solution idéale permettant à la fois de refléter l'exécution du logiciel sur le processeur cible, puisqu'elle permet de suivre de manière relativement précise le flot d'exécution du programme, sans pour autant affecter le code source d'origine. Contrairement à une approche d'instrumentation au niveau source, cette solution non intrusive permet donc aux développeurs de valider directement leurs codes source.

La prise en compte des informations d'annotations dans la plateforme de simulation est réalisée très simplement par l'appel d'une fonction au début de chaque *Basic Block* du programme ou sur les arcs du CFG le nécessitant. Cette fonction prend en argument l'adresse en mémoire où se trouvent les informations d'annotation qui pourront ensuite être interprétées par la plateforme de simulation.

Les expérimentations nous ont permis principalement de montrer la capacité de cette approche à suivre le flot d'exécution du programme. L'aspect temporel a été abordé ici de manière superficielle et fera l'objet de travaux futurs.

8.2 Perspectives

Nous commencerons tout d'abord par les perspectives concernant l'instrumentation du code logiciel. Cette contribution ne représente en effet encore qu'une partie d'un travail plus conséquent dont l'objectif est d'estimer les performances d'exécution du logiciel. Nous avons simplement présenté dans le chapitre 6 une première ébauche d'analyse et comment les annotations insérées dans le programme peuvent être utilisées sur la plateforme de simulation. Le principal travail qui va suivre cette thèse sera donc tout naturellement consacré à l'estimation des performances d'exécution du logiciel, notamment sur des architectures de processeurs plus complexes telles que les architectures superscalaires ou VLIW.

L'implémentation de l'instrumentation au sein de LLVM nous a permis de prendre conscience d'un certain nombre de limitations, la principale étant liée au maintien de la représentation LLVM tout au long du *back-end* du compilateur. Une des améliorations possible consisterait donc à attendre la fin du *back-end* spécifique au processeur cible sans maintenir la représentation intermédiaire à jour, puis de faire une conversion de la représentation *Machine-LLVM* dépendante du processeur vers la représentation *LLVM* indépendante afin de bénéficier naturellement des optimisations effectuées sur le CFG, sans pour autant avoir à les maîtriser.

En ce qui concerne la plateforme de simulation native, nous nous sommes principalement focalisés sur la méthode permettant de supporter l'exécution du logiciel plutôt que sur les précisions dans les estimations temporelles. Cela explique entre autre que nous n'ayons présenté dans les expérimentations qu'une seule application. Celle-ci était en effet suffisante pour valider les différents concepts que nous avons proposés. Au cours de ce

travail nous avons bien entendu utilisé d'autres applications, tels que celles fournies dans *SPLASH-2* [WOT+], mais que nous n'avons pas jugé utile de présenter car elles n'apportent ici pas plus d'information sur les capacités de la plateforme de simulation.

Dans les travaux futurs, nous approfondirons la modélisation du temps dans la plateforme de simulation, et plus particulièrement l'aspect dynamique qui devra s'ajouter à l'estimation faite lors de l'analyse du code logiciel. Il reste également des améliorations à apporter à la plateforme de simulation native, afin de permettre l'exécution de systèmes d'exploitation plus complexes, mettant en œuvre par exemple les unités de gestion de mémoires virtuelles (MMU) actuellement non supportées. Des solutions envisageables sont l'utilisation d'exception du système d'exploitation de la machine hôte pour modéliser les défauts de page par exemple ou encore implémenter un chargeur d'application spécifique à la simulation native. Il ne s'agit là que de suggestion d'idées qui devront être étudiées plus en détail afin de déterminer leur faisabilité.

Glossaire

ADC	Analog to Digital Converter
AMP	Asymmetric MultipleProcessor
APES	APplication Elements for System-on-chips
API	Application Programming Interface
BLS	Binary Level Simulation
CA	Cycle Accurate
CABA	Cycle Accurate Bit Accurate
CFG	Control Flow Graph
DMA	Direct Memory Access
DNA	DNA is Not just Another OS
DSP	Digital Signal Processor
EU	Execution Unit
FPGA	Field Programmable Gate Array
HAL	Hardware Abstraction Layer
HDL	Hardware Description Language
HDS	Hardware Dependent Software
HPC	High-Performance Computing
GCC	GNU Compiler Collection
GDB	GNU DeBugger
GMN	Generic Micro Network
GNU	GNU's Not Unix
GPP	General Purpose Processor
IDCT	Inverse Discrete Cosine Transform
IPC	Inter Processus Communication
IR	Intermediate Representation
IRLS	Intermediate Representation Level Simulation
ISA	Instruction Set Architecture
ISC	Intermediate Source Code
ISS	Instruction Set Simulator
JiT	Just in Time
JPEG	Joint Photographic Experts Group
JVM	Java Virtual Machine
LLVM	Low Level Virtual Machine
LTO	Link Time Optimization
MJPEG	Motion-JPEG
MMU	Memory Managment Unit
MPSoC	MultiProcessors System On Chip
NoC	Network On Chip
POSIX	Portable Operating System Interface
PV	Programmer's View
PVT	Programmer's View with Time
RAMDAC	Random Access Memory Digital-to-Analog Converter
RISC	Reduced Instruction Set Computer
RPC	Remote Procedure Call
RTL	Register Transfer Level
SAD	Sum of Absolute Differences
SE	SystÃ¨me d'Exploitation
SL	System Level
SLS	System Level Simulation
SMP	Symmetric Multiple Processor
SoC	System On Chip
TA	Transaction Accurate
TLM	Transaction Level Modeling
UML	Unified Modeling Language
VCI	Virtual Component Interface
VHDL	VHSIC Hardware Description Language
VLD	Variable Length Decoder
VLIW	Very Long Instruction Word
WCET	Worst Case Execution Time

Bibliographie

[ACG+07] David August, Jonathan Chang, Sylvain Girbal, Daniel Gracia-Perez, Gilles Mouchard, David A. Penry, Olivier Temam, and Neil Vachharajani. Unisim : An open simulation environment and library for complex architecture design and collaborative development. volume 6, pages 45–48, Washington, DC, USA, 2007. IEEE Computer Society. 7.4

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. 5.1

[BBY+05] Aimen Bouchhima, Iuliana Bacivarov, Wassim Youssef, Marius Bonaciu, and Ahmed Amine Jerraya. Using abstract cpu subsystem simulation model for high level hw/sw architecture exploration. In *ASP-DAC*, pages 969–972, 2005. 2.4, 2.4.2, 3.1.2, 4.1, 4.2

[BK02] G. Bontempi and W. Kruijtzer. A data analysis method for software performance prediction. In *DATE '02 : Proceedings of the conference on Design, automation and test in Europe*, page 971, Washington, DC, USA, 2002. IEEE Computer Society. 3.2

[BKL+00] Jwahar R. Bammi, Wido Kruijtzer, Luciano Lavagno, Edwin Harcourt, and Mihai T. Lazarescu. Software performance estimation strategies in a system-level design tool. In *CODES '00 : Proceedings of the eighth international workshop on Hardware/software codesign*, pages 82–86, New York, NY, USA, 2000. ACM. 3.2.1, 3.2.2

[BPN+04] Alex Bobrek, Joshua J. Pieper, Jeffrey E. Nelson, JoAnn M. Paul, and Donald E. Thomas. Modeling shared resource contention using a hybrid simulation/analytical approach. In *DATE '04 : Proceedings of the conference on Design, automation and test in Europe*, page 21144, Washington, DC, USA, 2004. IEEE Computer Society. 3.1.3

[BYJ02] M. Bacivarov, Sungjoo Yoo, and A. A. Jerraya. Timed hw-sw cosimulation using native execution of os and application sw. In *HLDVT '02 : Proceedings of the Seventh IEEE International High-Level Design Validation and Test Workshop*, page 51, Washington, DC, USA, 2002. IEEE Computer Society. 3.1.1

[BYJ04] Aimen Bouchhima, Sungjoo Yoo, and Ahmed Jerraya. Fast and accurate timed execution of high level embedded software using hw/sw interface simulation model. In *ASP-DAC '04 : Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, pages 469–474, Piscataway, NJ, USA, 2004. IEEE Press. 3.1.2

[CBR+04] Jerome Chevalier, Olivier Benny, Mathieu Rondonneau, Guy Bois, El Mostapha Aboulhamid, and Francois-Raymond Boyer. Space : a hardware/soft-

ware systemc modeling platform including an rtos. pages 91–104, Norwell, MA, USA, 2004. Kluwer Academic Publishers. 3.1.1

[CG03] Lukai Cai and Daniel Gajski. Transaction level modeling : an overview. In *CODES+ISSS '03 : Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24, New York, NY, USA, 2003. ACM. 2.3.3

[CHB09a] Eric Cheung, Harry Hsieh, and Felice Balarin. Fast and accurate performance simulation of embedded software for mpsoc. In *ASP-DAC '09 : Proceedings of the 2009 Conference on Asia and South Pacific Design Automation*, pages 552–557, Piscataway, NJ, USA, 2009. IEEE Press. 3.2, 3.2.3, 4.5, 5.3

[CHB09b] Eric Cheung, Harry Hsieh, and Felice Balarin. Memory subsystem simulation in software tlm/t models. In *ASP-DAC '09 : Proceedings of the 2009 Conference on Asia and South Pacific Design Automation*, pages 811–816, Piscataway, NJ, USA, 2009. IEEE Press. 3.1.1, 3.1.1

[CMMC08] Jérôme Cornet, Florence Maraninchi, and Laurent Maillet-Contoz. A method for the efficient development of timed and untimed transaction-level models of systems-on-chip. In *DATE '08 : Proceedings of the conference on Design, automation and test in Europe*, pages 9–14, New York, NY, USA, 2008. ACM. 2.3.3.1

[DBDSVP+04] Bruno De Bus, Bjorn De Sutter, Ludo Van Put, Dominique Chanet, and Koen De Bosschere. Link-time optimization of arm binaries. In *LCTES '04 : Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 211–220, New York, NY, USA, 2004. ACM. 5.4

[DGM09] Rainer Dömer, Andreas Gerstlauer, and Wolfgang Müller. Introduction to hardware-dependent software design hardware-dependent software for multi- and many-core embedded systems. In *ASP-DAC '09 : Proceedings of the 2009 Conference on Asia and South Pacific Design Automation*, pages 290–292, Piscataway, NJ, USA, 2009. IEEE Press. 2.1.2.2

[DM99] Giovanni De Micheli. Hardware synthesis from c/c++ models. In *DATE '99 : Proceedings of the conference on Design, automation and test in Europe*, page 80, New York, NY, USA, 1999. ACM. 2.2

[Don04] Adam Donlin. Transaction level modeling : flows and use models. In *CODES+ISSS '04 : Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 75–80, New York, NY, USA, 2004. ACM. 2.3.3

[ECO] eCos homepage, http ://ecos.sourceware.org/. 7.2.1

[ESE] ESE : Embedded Software Environment. http ://www.cecs.uci.edu/ẽse/. 3.2.3

[Ghe06] Frank Ghenassia. *Transaction-Level Modeling with Systemc : Tlm Concepts and Applications for Embedded Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 2.3.3

[GHP09] Patrice Gerin, Mian Muhammad Hamayun, and Frédéric Pétrot. 2009. 6.2.1.2

[GKK+08] Lei Gao, Kingshuk Karuri, Stefan Kraemer, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Multiprocessor performance estimation using hybrid

simulation. In *DAC '08 : Proceedings of the 45th annual conference on Design automation*, pages 325–330, New York, NY, USA, 2008. ACM. (document), 3.1.3, 3.4, 3.1.3

[GKL+07] Lei Gao, Stefan Kraemer, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. A fast and generic hybrid simulation approach using c virtual machine. In *CASES '07 : Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 3–12, New York, NY, USA, 2007. ACM. 3.1.3

[GL97] Rajesh K. Gupta and Stan Y. Liao. Using a programming language for digital system design. volume 14, pages 72–80, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press. 2.2

[GMH01] P. Giusto, G. Martin, and E. Harcourt. Reliable estimation of execution time of embedded software. In *DATE '01 : Proceedings of the conference on Design, automation and test in Europe*, pages 580–589, Piscataway, NJ, USA, 2001. IEEE Press. 3.2, 3.2.2

[GP09] X. Guerin and F. Petrot. A system framework for the design of embedded software targeting heterogeneous multi-core socs. In *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, pages 153–160, July 2009. 7.2

[GRA] Graphviz homepage, http ://www.graphviz.org. 5.5.2

[GSC+07] P. Gerin, H. Shen, A. Chureau, A. Bouchhima, and A. Jerraya. Flexible and Executable Hardware/Software Interface Modeling for Multiprocessor Soc Design Using Systemc. In *ASP-DAC '07 : Proceedings of the 2007 conference on Asia South Pacific design automation*, pages 390–395, Washington, DC, USA, 2007. IEEE Computer Society. 4.2

[GYG03] Andreas Gerstlauer, Haobo Yu, and Daniel D. Gajski. Rtos modeling for system level design. In *DATE '03 : Proceedings of the conference on Design, Automation and Test in Europe*, page 10130, Washington, DC, USA, 2003. IEEE Computer Society. 3.1.1

[GYNJ01] Patrice Gerin, Sungjoo Yoo, Gabriela Nicolescu, and Ahmed A. Jerraya. Scalable and flexible cosimulation of soc designs with heterogeneous multiprocessor target architectures. In *ASP-DAC '01 : Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 63–68, New York, NY, USA, 2001. ACM. 3.1.1

[GZD+00] Daniel Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC : Specification Language and Methodologie*. Kluwer Academic Publishers, Boston, March 2000. 2.2

[HAG08] Yonghyun Hwang, Samar Abdi, and Daniel Gajski. Cycle-approximate retargetable performance estimation at the transaction level. In *DATE '08 : Proceedings of the conference on Design, automation and test in Europe*, pages 3–8, New York, NY, USA, 2008. ACM. 3.2.3

[HHP+07] Kai Huang, Sang-il Han, Katalin Popovici, Lisane Brisolara, Xavier Guerin, Lei Li, Xiaolang Yan, Soo-lk Chae, Luigi Carro, and Ahmed Amine Jerraya. Simulink-based mpsoc design flow : case study of motion-jpeg and h.264. In *DAC '07 : Proceedings of the 44th annual Design Automation Conference*, pages 39–42, New York, NY, USA, 2007. ACM. 2.3.1

[HR00] A. Hergenhan and W. Rosenstiel. Static timing analysis of embedded software on advanced processor architectures. In *DATE '00 : Proceedings of the conference on Design, automation and test in Europe*, pages 552–559, New York, NY, USA, 2000. ACM. 3.2

[HWTT04] Shinya Honda, Takayuki Wakabayashi, Hiroyuki Tomiyama, and Hiroaki Takada. Rtos-centric hardware/software cosimulator for embedded system design. In *CODES+ISSS '04 : Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 158–163, New York, NY, USA, 2004. ACM. 3.1.1

[ILK+] Tsuyoshi Isshiki, Dongju Li, Hiroaki Kunieda, Toshio Isomura, and Kazuo Satou. Trace-driven workload simulation method for multiprocessor system-on-chip. In *Papier DAC 09*. 3.2, 3.2.2

[ITR07] ITRS. International technology roadmap for semiconductors. In *System Drivers*, 2007. (document), 1, 1.3

[ITR08] ITRS. International technology roadmap for semiconductors. In *Update Overview*, 2008. 1

[JBP06] Ahmed A. Jerraya, Aimen Bouchhima, and Frédéric Pétrot. Programming models and hw-sw interfaces abstraction for multi-processor soc. In *DAC '06 : Proceedings of the 43rd annual conference on Design automation*, pages 280–285, New York, NY, USA, 2006. ACM. 2.1.2.2, 2.3.3

[KAFK+05] Kingshuk Karuri, Mohammad Abdullah Al Faruque, Stefan Kraemer, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Fine-grained application source code profiling for asip design. In *DAC '05 : Proceedings of the 42nd annual Design Automation Conference*, pages 329–334, New York, NY, USA, 2005. ACM. 3.2.3

[KD90] David Ku and Giovanni DeMicheli. HardwareC – a language for hardware design (version 2.0). Technical report, Stanford, CA, USA, 1990. 2.2

[KEBR08] Matthias Krause, Dominik Englert, Oliver Bringmann, and Wolfgang Rosenstiel. Combination of instruction set simulation and abstract rtos model execution for fast and accurate target software evaluation. In *CODES/ISSS '08 : Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 143–148, New York, NY, USA, 2008. ACM. 3.1.3, 3.1.3

[KGW+07] Stefan Kraemer, Lei Gao, Jan Weinstock, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Hysim : a fast simulation framework for embedded software development. In *CODES+ISSS '07 : Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 75–80, New York, NY, USA, 2007. ACM. 3.1.3, 3.1.3

[KKW+06] Torsten Kempf, Kingshuk Karuri, Stefan Wallentowitz, Gerd Ascheid, Rainer Leupers, and Heinrich Meyr. A sw performance estimation framework for early system-level-design using fine-grained instrumentation. In *DATE*, pages 468–473, 2006. 3.1.1, 3.2.3, 6.2.1.2

[LA04] C. Lattner and V. Adve. Llvm : A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2004. 5.5.1

[LBH+00] M. T. Lazarescu, J. R. Bammi, E. Harcourt, L. Lavagno, and M. Lajolo. Compilation-based software performance estimation for system level design. In *HLDVT '00 : Proceedings of the IEEE International High-Level Validation and Test Workshop (HLDVT'00)*, page 167, Washington, DC, USA, 2000. IEEE Computer Society. 3.2.1

[Lev99] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. 4.3.2

[LMPC04] R. Le Moigne, O. Pasquier, and J-P. Calvez. A generic rtos model for real-time systems simulation with systemc. In *DATE '04 : Proceedings of the conference on Design, automation and test in Europe*, page 30082, Washington, DC, USA, 2004. IEEE Computer Society. 3.1.1, 3.2.3

[LP02] Jong-Yeol Lee and In-Cheol Park. Timed compiled-code simulation of embedded software for performance analysis of soc design. In *DAC '02 : Proceedings of the 39th conference on Design automation*, pages 293–298, New York, NY, USA, 2002. ACM. 6.2.1.2, 6.2.1.2

[MML97] Sharad Malik, Margaret Martonosi, and Yau-Tsun Steven Li. Static timing analysis of embedded software. In *DAC '97 : Proceedings of the 34th annual Design Automation Conference*, pages 147–152, New York, NY, USA, 1997. ACM. 3.2

[MRRJ04] Anish Muttreja, Anand Raghunathan, Srivaths Ravi, and Niraj K. Jha. Automated energy/performance macromodeling of embedded software. In *DAC '04 : Proceedings of the 41st annual Design Automation Conference*, pages 99–102, New York, NY, USA, 2004. ACM. 3.1.3

[MRRJ05] Anish Muttreja, Anand Raghunathan, Srivaths Ravi, and Niraj K. Jha. Hybrid simulation for embedded software energy estimation. In *DAC '05 : Proceedings of the 42nd annual conference on Design automation*, pages 23–26, New York, NY, USA, 2005. ACM. 3.1.3, 3.1.3

[MSVSL08] Trevor Meyerowitz, Alberto Sangiovanni-Vincentelli, Mirko Sauermann, and Dominik Langen. Source-level timing annotation and simulation for a heterogeneous multiprocessor. In *DATE '08 : Proceedings of the conference on Design, automation and test in Europe*, pages 276–279, New York, NY, USA, 2008. ACM. 3.2.2

[NEW] Newlib c homepage, http ://sources.redhat.com/newlib/. 7.2

[NST06] Hiroaki Nakamura, Naoto Sato, and Naoshi Tabuchi. An efficient and portable scheduler for rtos simulation and its certified integration to systemc. In *DATE '06 : Proceedings of the conference on Design, automation and test in Europe*, pages 1157–1158, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association. 3.1.1, 3.2.3

[OZW04] Márcio Seiji Oyamada, Felipe Zschornack, and Flávio Rech Wagner. Accurate software performance estimation using domain classification and neural networks. In *SBCCI '04 : Proceedings of the 17th symposium on Integrated circuits and system design*, pages 175–180, New York, NY, USA, 2004. ACM. 3.2

[PAS+06] Hector Posadas, Jesús Ádamez, Pablo Sánchez, Eugenio Villar, and Francisco Blasco. Posix modeling in systemc. In *ASP-DAC '06 : Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 485–490, Piscataway, NJ, USA, 2006. IEEE Press. 3.1.1

[PH98] David A. Patterson and John L. Hennessy. *Computer organization and design (2nd ed.) : the hardware/software interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. 4.2

[PHS+04] H. Posadas, F. Herrera, P. Sánchez, E. Villar, and F. Blasco. System-level performance analysis in systemc. In *Proceedings of the conference on Design, automation and test in Europe*, 2004. 3.2.2, 6.2.1.2

[PJ07] Katalin Popovici and Ahmed Amine Jerraya. Simulink based hardware-software codesign flow for heterogeneous mpsoc. In *SCSC : Proceedings of the 2007 summer computer simulation conference*, pages 497–504, San Diego, CA, USA, 2007. Society for Computer Simulation International. 2.3.1, 3.1.2

[RBL06] Thomas Reps, Gogul Balakrishnan, and Junghee Lim. Intermediate-representation recovery from low-level code. In *PEPM '06 : Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 100–111, New York, NY, USA, 2006. ACM. 5.4

[RGC+09] K. Rahmouni, P. Gerin, S. Chabanet, P. Pianu, and F. Petrot. Modelling and architecture exploration of a medium voltage protection device. In *Industrial Embedded Systems, 2009. SIES '09. IEEE International Symposium on*, pages 46–49, July 2009. 2.2

[Ric69] Martin Richards. Bcpl : a tool for compiler writing and system programming. In *AFIPS '69 (Spring) : Proceedings of the May 14-16, 1969, spring joint computer conference*, pages 557–566, New York, NY, USA, 1969. ACM. 2

[RS09] Christine Rochange and Pascal Sainrat. A context-parameterized model for static analysis of execution times. pages 222–241, Berlin, Heidelberg, 2009. Springer-Verlag. 6.2.1.1

[SD08] Gunar Schirner and Rainer Dömer. Introducing preemptive scheduling in abstract rtos models using result oriented modeling. In *DATE '08 : Proceedings of the conference on Design, automation and test in Europe*, pages 122–127, New York, NY, USA, 2008. ACM. 3.1.1

[SOC] Soclib project. http ://www.soclib.fr. 7.1

[SYSa] Open systemc initiative homepage, http ://www.systemc.org. 2.2, 2.2.1

[SYSb] Systemcass homepage, https ://www-asim.lip6.fr/trac/systemcass/wiki. 7.3.6

[Tan84] A. Tanenbaum. *Structured computer organization ; (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984. 2.1.2.2

[TRKA07] Walter Tibboel, Victor Reyes, Martin Klompstra, and Dennis Alders. System-level design flow based on a functional reference for hw and sw. In *Proc. of the Design Automation Conference*, pages 23–28, 2007. 3.1.2

[VKS00] Diederik Verkest, Joachim Kunkel, and Frank Schirrmeister. System level design using c++. In *DATE '00 : Proceedings of the conference on Design, automation and test in Europe*, pages 74–83, New York, NY, USA, 2000. ACM. 2.2, 2.3.1

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. volume 7, pages 1–53, New York, NY, USA, 2008. ACM. 3.2

[WH09] Zhonglei Wang and Andreas Herkersdorf. An efficient approach for system-level timing simulation of compiler-optimized embedded software. In *Proc. of the Design Automation Conference*, 2009. (document), 3.2, 3.2.3, 3.5, 6.2.1.2

[Wol02] Fabian Wolf. *Behavioral Intervals in Embedded Software : Timing and Power Analysis of Embedded Real-Time Software Processes*. Kluwer Academic Publishers, Norwell, MA, USA, 2002. 6.2.4

[WOT+] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs : characterization and methodological considerations. In *ISCA '95*. 8.2

[WSH08] Zhonglei Wang, Antonio Sanchez, and Andreas Herkersdorf. Scisim : a software performance estimation framework using source code instrumentation. In *WOSP '08 : Proceedings of the 7th international workshop on Software and performance*, pages 33–42, New York, NY, USA, 2008. ACM. 3.1.1, 3.1.1, 3.2, 3.2.2, 3.2.3

[YBB+03] Sungjoo Yoo, Iuliana Bacivarov, Aimen Bouchhima, Yanick Paviot, and Ahmed A. Jerraya. Building fast and accurate sw simulation models based on hardware abstraction layer and simulation environment abstraction layer. In *DATE '03 : Proceedings of the conference on Design, Automation and Test in Europe*, page 10550, Washington, DC, USA, 2003. IEEE Computer Society. 3.1.2

[YJ03] Sungjoo Yoo and Ahmed A. Jerraya. Introduction to hardware abstraction layers for soc. In *DATE '03 : Proceedings of the conference on Design, Automation and Test in Europe*, page 10336, Washington, DC, USA, 2003. IEEE Computer Society. 2.1.2.2, 2.1

[ZM96] V. Zivojnovic and H. Meyr. Compiled hw/sw co-simulation. In *Design Automation Conference Proceedings 1996, 33rd*, pages 690–695, Jun, 1996. 3.2.1

Publications

Conférences Internationnales

[1] **P. Gerin**, S. Yoo, G. Nicolescu, and A. Jerraya. Scalable and Flexible Cosimulation of SoC Designs with Heterogeneous Multi-Processor Target Architectures. In *ASP-DAC '01 : Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 63–68, New York, NY, USA, 2001. ACM.

[2] A. Bouchhima, L. Kriaa, W. Youssef, **P. Gerin**, F. Pétrot, A. A. Jerraya. A Unified HW/SW Interface Refinement Approach for MPSoC Design In *Proceedings of The IEEE-NEWCAS Conference 2006*, Gatineau, Canada, June 18-21, 2006.

[3] **P. Gerin**, H. Shen, A. Chureau, A. Bouchhima, and A. Jerraya. Flexible and Executable Hardware/Software Interface Modeling for Multiprocessor Soc Design Using Systemc. In *ASP-DAC '07 : Proceedings of the 2007 conference on Asia South Pacific design automation*, pages 390–395, Washington, DC, USA, 2007. IEEE Computer Society.

[4] **P. Gerin**, X. Guérin and F. Pétrot. Efficient Implementation of Native Software Simulation for MPSoC In *DATE '08 : Proceedings of the conference on Design, automation and test in Europe*, pages 676–681, New York, NY, USA, 2008. ACM.

[5] H. Shen and **P. Gerin** and F. Pétrot. Configurable Heterogeneous MPSoC Architecture Exploration Using Abstraction Levels. In RSP '08 : Proceedings of the 2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping, pages 51–57, Washington, DC, USA, IEEE Computer Society.

[6] K. Rahmouni and **P. Gerin** and S. Chabanet and P. Pianu and F. Pétrot. Modelling and Architecture Exploration of a Medium Voltage Protection Device. In Industrial Embedded Systems, 2009. SIES '09. IEEE International Symposium on, pages 46–49, 2009.

[7] **P. Gerin**, M. M. Hamayun, and F. Pétrot. Native MPSoC Co-Simulation Environment for Software Performance Estimation In *CODES+ISSS '09 : Proceedings of the conference on Hardware/software codesign and system synthesis*, à paraître.

Chapitres de livres

[8] Frédéric Pétrot and **Patrice Gerin**. Simulation at cycle accurate and transaction accurate levels. In *System level design with .NET technology*, El-Mostapha Aboulamid and Frédéric Rousseau, editors. Chapter 6, pp. 159-180, 2009, Taylor and Francis, CRC Press.

[9] Frédéric Pétrot, **Patrice Gerin** and Mian-Muhammad Hamayun. On Software Simulation for MPSoC : A Modeling Approach for Functional Validation and Performance Estimation. In *Heterogeneous Embedded Systems - Design Theory and Practice*, Ian O'Connor and Gabriela Nicolescu, editors. Planned for publication in 2010, CRC Press.

RÉSUMÉ

Les systèmes sur puces actuelles (SoC) mettent à profit des architectures multiprocesseurs hétérogènes (MPSoC) afin de répondre dans des délais de conception très courts aux exigences du marché en terme de performances et de consommation. L'utilisation de plusieurs processeurs à certes un coût par rapport à des solutions "plus matérielles" mais qui est compensé par la flexibilité offerte par le logiciel. Cette dominance du logiciel nous contraint à commencer la validation et l'intégration avec le matériel dès les premières étapes des flots de conception.

La simulation est le seul moyen de valider un circuit ne pouvant pas être décrit analytiquement ou dont la complexité ne permet pas aux concepteurs de le faire, soit en pratique, tous les systèmes intégrés. Les principales contributions de cette thèse sont (1) la proposition d'une méthodologie de conception de plateformes de simulation basée sur l'exécution native du logiciel, (2) une technique d'instrumentation permettant l'annotation du logiciel s'exécutant sur cette plateforme de simulation.

Les plateformes de simulation ainsi développées permettent, malgré le niveau d'abstraction utilisé, l'exécution de la quasi totalité du logiciel final (système d'exploitation, pilotes de périphériques, application) sur des modèles réalistes de l'architecture matérielle du système. Associées à la technique d'instrumentation, ces plateformes permettent de prendre en compte de manière précise des grandeurs physiques telles que le temps ou la consommation liées à l'exécution du logiciel.

MOTS-CLÉS : systèmes sur puces, simulation native, architectures multiprocesseurs, instrumentation automatique du logiciel

ABSTRACT

Current Multi-Processor System-On-Chips (MPSoCs) architectures benefit from the heterogeneous multiprocessor platforms to fit consumption and performance requirements while keeping software flexibility. The dominance of software in MPSoCs compel designers to start the software validation and integration with hardware in the earlier steps of the design flow in order to achieve a short time-to-market.

The contributions of this thesis are (1) the proposition of a methodology to model simulation platforms based on the native execution of the software, (2) an instrumentation techniques that allow the annotation of the embedded software.

These simulation platforms allow the execution of almost all parts of the final software (including the operating system) on top of realistic hardware architecture models of the targeted system. Combined with the instrumentation technique, these platforms allow to precisly consider physical quantities such as the execution time or the electrical consumption related to the software execution.

KEY-WORDS : systems on chip, native simulation, multiprocessors architectures, automated software instrumentation

ISBN : 978-2-84813-142-9