Environnement de Programmation Multi Niveau pour Architectures Hétérogènes MPSoC

K. Popovici

▶ **To cite this version:**

K. Popovici. Environnement de Programmation Multi Niveau pour Architectures Hétérogènes MP-SoC. Micro et nanotechnologies/Microélectronique. Institut National Polytechnique de Grenoble - INPG, 2008. Français. NNT : . tel-00271128

HAL Id: tel-00271128
https://theses.hal.science/tel-00271128

Submitted on 8 Apr 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

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

<u>T H E S E</u>

pour obtenir le grade de

DOCTEUR DE L'INP Grenoble

Spécialité : « Micro et nano électronique»

préparée au laboratoire **TIMA**
dans le cadre de **l'Ecole Doctorale *« Electronique, Electrotechnique, Automatique et Traitement du Signal »***

présentée et soutenue publiquement

par

Katalin Maria POPOVICI

le 25 Mars 2008

Environnement de Programmation Multi Niveau pour Architectures Hétérogènes MPSoC

DIRECTEUR DE THESE : Dr. Ahmed Amine JERRAYA

M. Guy Mazaré	, Président
M. Fabrice Kordon	, Rapporteur
M. Dominique Lavenier	, Rapporteur
M. Ahmed Amine Jerraya	, Directeur de thèse
M. Tanguy Risset	, Examinateur
M. Frédéric Rousseau	, Examinateur

A ma famille

Remerciements

Je voudrais remercier profondément à mon directeur de thèse, **M. Ahmed Jerraya** de m'avoir donné l'opportunité de faire une thèse dans le groupe SLS, son soutien et la liberté qu'il m'a laissé dans mon travail. Qu'il trouve toute l'expression de ma sincère reconnaissance.

Je remercie à **M. Guy Mazaré** d'avoir présidé mon jury de thèse, et aussi pour ces appréciations. Je tiens à remercier à **M. Fabrice Kordon** et **M. Dominique Lavenier** d'avoir accepté rapporter sur mon travail de thèse et pour leur remarques qui ont beaucoup contribuée pour améliorer cette thèse. Je remercie à **M. Tanguy Risset** pour sa participation à mon jury de thèse et pour ses commentaires pertinents et à **M. Frédéric Rousseau** pour son soutien et surtout pour ses conseils tout au long de mon séjour ici.

Je tiens à remercier à l'ancien directeur du laboratoire, **M. Bernard Courtois** et à la nouvelle directrice **Mme Dominique Borrione**, pour leur accueil au laboratoire TIMA.

Je remercie particulièrement à **Lorena** et **Nacer** de m'avoir encadré pendant mes stages Socrates au TIMA et de m'avoir accordé beaucoup de leur temps, pour tous les conseils, les discussions fructueuses et les efforts fournis.

Je remercie à tous mes collègues de l'équipe SLS pour leur collaboration et pour leur sympathie. Je tiens à remercier en particulier à **Hao**, **Youssef**, **Youngchul** et **Alexandre (Chagoya)**, mes collègues de bureau, pour les très bons moments qu'on a vécu ensemble dans le bureau 417. Je tiens à remercier à tous qui m'ont aidé à commencer cette thèse : **Iuliana, Cosmin, Gabriela N., Wassim, Sang-Il, Marcio, Arnaud, Ivan, Lorenzo, Frédéric H., Wander** et pour leur gentillesse lorsque je suis arrivée dans le groupe SLS. J'exprime mon gratitude à tous mes amis du groupe SLS (en manque d'espace je vais citer en ordre alphabétique, mais je porte chacun d'entre eux un sentiment d'amitié personnel) : **Abdel, Aimen, Alex (Chureau), Amin, Benaoumeur, Lilia, Lobna, Patrice, Pierre, Quentin, Xavier ;** et à tous ceux qui m'ont donné des conseils précieux et qui m'ont montré leur soutien : **Sonja, Frédéric P., Paul**. Un grand merci a tous les membres du laboratoire TIMA que je n'ai pas cité mais que je n'oublie pas.

Un grand merci à mes amis roumains : **Claudia**, **Marius (Gligor), Luiza.** Je leur souhaite bonne chance et bonne continuation. J'adresse mes remerciements à mes amis brésiliennes : **Lisane Brisolara** et **Edson Moreno.** Bon courage pour la suite. Je remercie également à mes amis de Grenoble : **Manu, Kamel, Nadir**.

Une pensée toute particulière pour mon fiancé **Marius**. Je tiens à lui exprimer tout mon amour et ma gratitude pour son soutien, pour le support qu'il m'a montré dans les moments difficiles et pour avoir partagé les moments heureux. Je tiens aussi à lui remercier pour l'exemple qu'il m'a donné dans sa persévérance au travail, pour sa patience et sa tendresse. Je lui souhaite une carrière brillante, et que tous ses rêves se matérialisent.

Je finis par remercier à **mes parents** pour leur amour, leur constant soutient intellectuel et affectif, pour leur encouragements et leur confiance. Je vous aime. Je remercie également à mon frère, **Zoli** et ma belle sœur **Tunde** pour leur amitié et soutien, ainsi que le cadeau le plus important qu'ils m'ont offert : ma très belle nièce **Boglarka**. Je vous aime tous.

Table de Matières

1. Contexte 9
2. Motivations.. 11
3. Objectifs 13
4. État de l'art.. 14
5. Contribution 16
   5.1. Conception de l'Architecture Système.. 18
   5.2. Conception de l'Architecture Virtuelle 20
   5.3. Conception de l'Architecture Transactionnelle 20
   5.4. Conception du Prototype Virtuel.. 21

1. Contexte

Les progrès technologiques constants en termes d'intégration sur silicium ont permis de concevoir des systèmes sur puces de plus en plus complexes. Cette thèse s'inscrit dans le domaine de la conception de systèmes embarqués multiprocesseurs monopuces, plus communément appelés MPSoC.

La complexité croissante des MPSoC est accentuée par l'émergence de nouvelles applications télécoms (WCDMA, CDMA 2000) et multimédia ((MPEG 2/4, H.263/4, MP3) ou encore d'applications de jeux vidéo avec des contraintes fonctionnelles (puissance de calcul, consommation, embarquabilité, reconfigurabilité) et non fonctionnelles (temps de mise sur le marché, rétrécissement de la durée de vie du produit, coût) de plus en plus sévères. Pour répondre à ces exigences et maîtriser cette complexité, les architectures MPSoC hétérogènes sont essentielles afin d'atteindre les performances de calcul et de communication requises [Mey 06].

Un système MPSoC hétérogène comprend différents types d'unités de calcul spécifiques programmables et/ou non programmables (DSP, microcontrôleur, ASIP, FPGA, ASIC, etc.) et différents réseaux de communication (liens rapides, non standard pour l'organisation et l'accès mémoire, bus hiérarchiques sur puce, réseau sur puce). Ce type de plateformes hétérogènes offre à la fois un parallélisme de calcul et une programmabilité très souple.

Les plateformes hétérogènes typiques utilisées dans l'industrie sont le TI OMAP [TI], le ST Nomadik [Nom], le Philips Nexperia [Nex] et le Diopsis D940 [Dio]. Ils intègrent un processeur DSP et un microcontrôleur. De plus, la communication se fait via une infrastructure efficace mais très sophistiquée. Les architectures hétérogènes de MPSoC peuvent être représentées comme un ensemble de sous-systèmes logiciels et matériels interconnectés par un réseau de communication (figure 1) [Cul 98].

Un sous-système logiciel est un sous-système programmable, à savoir un sous-système contenant une entité programmable par un langage de haut niveau. Celui-ci intègre différents composants matériels comprenant une unité de calcul (CPU), des composants locaux spécifiques tels que des mémoires locales, des registres de données et de contrôle, des accélérateurs de matériel, un contrôleur d'interruptions, un composant DMA pour l'accès

direct aux mémoires, des composants de synchronisation tels que boîtes aux lettres (mailbox) ou sémaphores, des composants spécifiques d'entrée-sortie, etc. (d'autres périphériques).

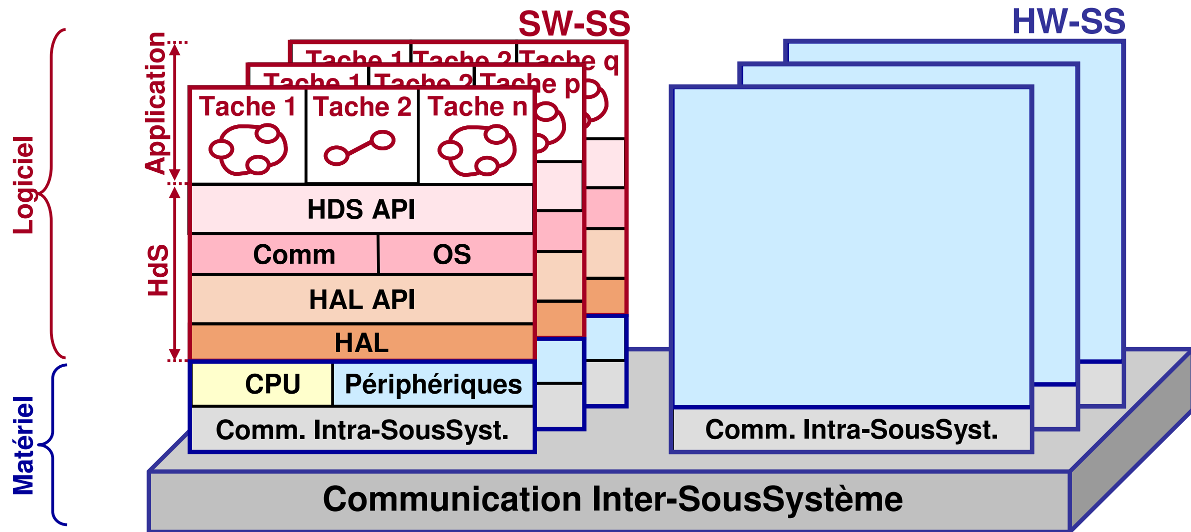


Figure 1. Architecture matérielle – logicielle pour MPSoC

Chaque sous-système logiciel exécute une pile logicielle spécifique. La pile logicielle est organisée en 2 couches : l'application et le logiciel dépendant du matériel (Hardware dependent Software ou HdS). La couche applicative est associée au comportement des fonctions de haut niveau composant les tâches de l'application ciblée. La couche de HdS est associée au comportement du logiciel de bas niveau dépendant du matériel, tel que routines d'interruptions, changement de contexte, contrôle d'entrée-sortie spécifique et ordonnancement des tâches.

En fait, la couche de HdS inclut 3 composants : le système d'exploitation (SE) ou Operating System (OS), la communication spécifique d'entrée-sortie (COMM) et la couche d'abstraction du matériel (HAL). Le système d'exploitation permet d’ordonnancer les tâches, de gérer le matériel, etc. La communication permet d’abstraire la réalisation des ressources de communication. Le HAL comporte les pilotes d’entrées/sorties et les contrôleurs de bas niveau permettant d’accéder directement au matériel. Le code correspondant à cette couche est fortement lié au matériel. Ces différents éléments sont basés sur des primitives ou Interfaces de Programmation d'Application (Application Programming Interface ou APIs) bien définis, afin de passer d'une couche de logiciel à l'autre.

Un sous-système matériel représente un composant matériel qui met en œuvre les fonctionnalités spécifiques à l'application ou un sous-système de mémoires globales accessible par les unités de calcul.

2. Motivations

De part de leur nature, les MPSoC sont capables d'effectuer plusieurs calculs en parallèle [Lav 06]. Dans la pratique, la programmation de telles architectures consiste généralement à écrire des codes séparés pour les différents types de processeurs (DSP, microcontrôleur), sans aucune validation de l'application globale sur la plateforme matérielle. La validation s'effectue seulement quand tous les binaires logiciels sont produits et peuvent être exécutés sur la plateforme matérielle.

Les systèmes embarqués incluent plusieurs processeurs qui exécutent des instructions spécifiques. Par conséquent, la complexité du code logiciel est très grande (environ 100000 lignes de code pour certaines applications) et demande donc un temps de conception très grand. Le logiciel ne peut plus être développé en langage assembleur. Donc, une approche de conception à un niveau d'abstraction plus élevé est requise.

Pour ce genre d'architectures MPSoC, les environnements de programmation classiques ne sont pas adaptés pour les raisons suivantes : (i) la programmation de haut niveau ne gère pas efficacement les entrées/sorties (I/Os) et les systèmes de communication spécifiques à l'architecture, tandis que (ii) la programmation de bas niveau avec la gestion explicite des entrées-sorties et de la communication spécifique est très coûteuse en termes de temps de développement et d'erreurs. De plus, la conception d'un système à un bas niveau nécessite un temps de conception trop long vu que le temps d'exécution est au niveau cycle d'horloge et la communication est au niveau registre.

La prochaine génération de MPSoC semble accentuer cette tendance en architectures matérielles plus complexes, car plusieurs DSP et microcontrôleurs seront intégrés sur une seule puce [Tur 05]. Ainsi, la principale difficulté est de savoir comment programmer de telles architectures de manière efficace, à partir d'un langage de haut niveau. Les différents types de processeurs exécutent différentes piles logicielles. Une difficulté additionnelle est de corriger et de valider les couches inférieures de logiciel requises pour le portage du code d'application de haut niveau sur l'architecture hétérogène ciblée. La validation et la correction du HdS sont le goulot d'étranglement principal dans la conception des MPSoC [Wol 06] car chaque sous-système de processeur nécessite un HdS spécifique afin d'être efficace.

Une programmation efficace exige l'utilisation des caractéristiques de l'architecture. Par exemple, un échange de données entre deux tâches exécutés sur des processeurs différents peut utiliser des systèmes de communication différents si on passe par une mémoire partagée

globale ou par la mémoire locale d'un de ces processeurs. De plus, des méthodes différentes de synchronisation (scrutation, interruptions) peuvent être employées pour synchroniser ces échanges. Chacun de ces systèmes de communication présente des avantages et des inconvénients en termes de performances (latence, débit), de partage des ressources (traitement multitâche, entrées-sorties parallèles) et de coût général de communication (taille de la mémoire, temps d'exécution). Le schéma idéal doit être en mesure de produire un logiciel efficace à partir d'un environnement haut niveau en utilisant des primitives génériques de communication, telles que des *send/recv* fournis par MPI [MPI].

Dans un flot de conception idéal, la génération du logiciel ciblant une architecture spécifique se compose du partitionnement et de la répartition de l'application sur l'architecture, de la génération du code final pour l'application et enfin de la génération du logiciel dépendant du matériel (HdS) (figure 2).

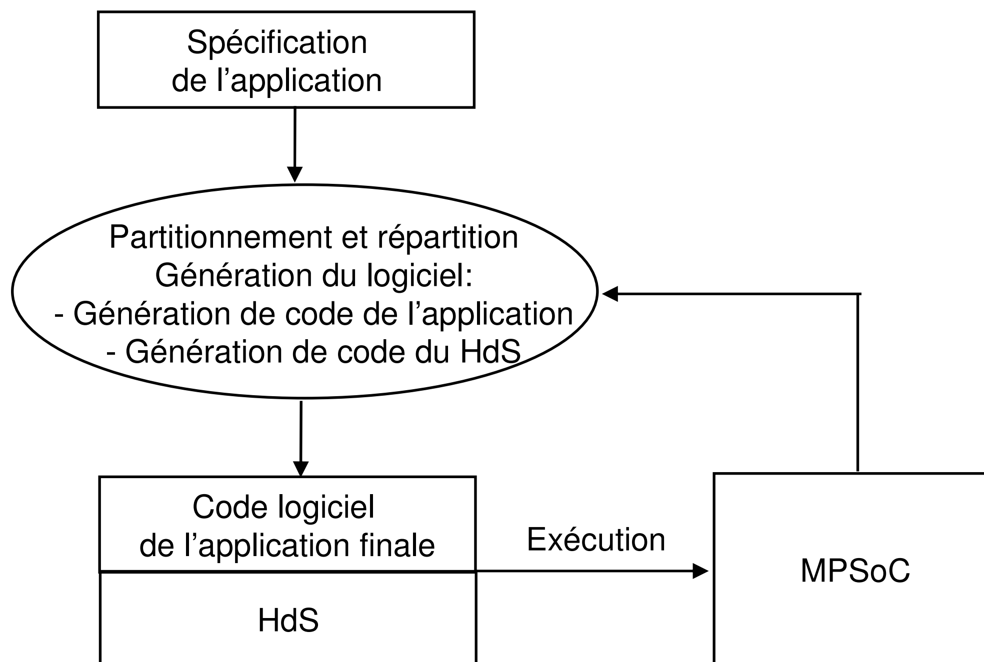


Figure 2. Flot de conception logiciel

Le HdS est constitué de couches de logiciel bas niveau qui peuvent incorporer un système d'exploitation (OS), une gestion de communication et une couche d'abstraction de matériel pour permettre aux fonctions de l'OS d'accéder aux ressources matérielles de la plateforme. Malheureusement, il n'existe pas de flot générique, capable de transposer efficacement des applications de haut niveau sur des architectures hétérogènes MPSoC.

Les approches classiques effectuent la validation du logiciel en employant une plateforme de développement. Comme la montre la figure 3, la plateforme de développement

logiciel est un modèle abstrait de l'architecture sous la forme d'une bibliothèque d'exécution ou de simulation visant à exécuter le logiciel [Vin 01]. La combinaison de la plateforme avec le code logiciel produit un modèle exécutable qui simule l'exécution du système final comprenant l'architecture matérielle et logicielle. Ce modèle exécutable permet la simulation du logiciel et des interactions détaillées entre matériel et logiciel, le débogage du logiciel et éventuellement l'évaluation de la performance. La plateforme et le logiciel peuvent être combinés en utilisant différents schémas. Le débogage du logiciel représente l'un des défis principaux dans la conception d'un MPSoC [Mar 06].

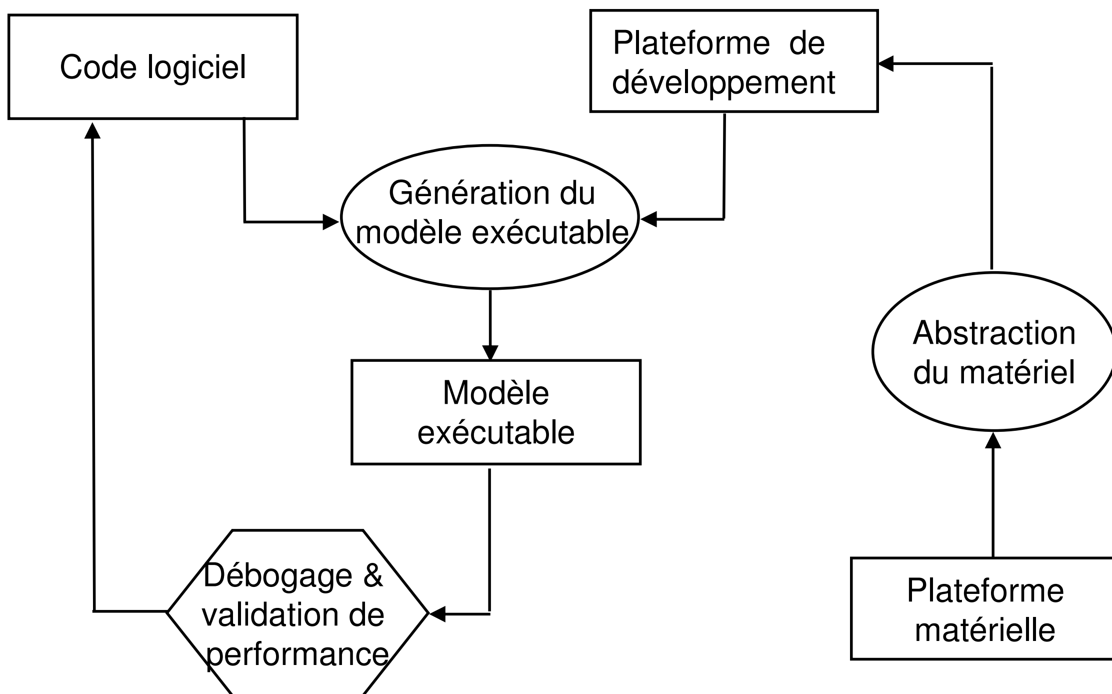


Figure 3. Plateforme de développement logiciel

3. Objectifs

L'objectif de cette thèse est de définir un flot de conception et de validation du logiciel, systématique, basé sur des plateformes de développement logiciel, capable d'employer efficacement les ressources de l'architecture matérielle visée. L'entrée du flot est un haut niveau d'abstraction qui permet la conception des applications sur les plateformes MPSoC hétérogènes existantes. L'objectif d'un tel flot est l'utilisation efficace des ressources de l'architecture et la génération d'un code exécutable pour les applications multimédia.

L'architecture et l'application sont décrites dans un modèle unique en Simulink. La répartition du logiciel sur les différents processeurs et les mécanismes de communication sont

explicites dans ce modèle de haut niveau. Les piles logicielles exécutables sont générées pour chaque unité de calcul à partir de la description initiale en Simulink. Chaque pile logicielle est structurée en couches pour permettre la flexibilité en matière de réutilisation de composants logiciels (OS, protocole de communication) et de portabilité sur d'autres plateformes (HAL). Ces différents composants logiciels, ainsi qu'une architecture matérielle permettant de valider ces composants, sont générés de façon systématique. Les composants logiciels correspondent à quatre niveaux d'abstraction différents de l'architecture matérielle/logicielle : niveau système (system architecture), niveau architecture virtuelle (virtual architecture), niveau architecture transactionnelle (transaction accurate architecture) et niveau prototype virtuel (virtual prototype). Pour permettre la validation du logiciel, différentes plateformes de développement sont également générées à chaque niveau d'abstraction.

Ansi, ce flot est basé sur quatre niveaux d'abstraction matériel/logiciel qui permettent une génération incrémentale du code logiciel ainsi qu'une validation de la pile logicielle à chaque niveau.

4. État de l'art

Les travaux précédents liés à la génération et la validation du logiciel à partir d'un environnement de haut niveau peuvent être classés en trois catégories : la conception orientée logiciel, la conception orientée matériel ou la conception orientée électronique au niveau système (Electronic System Level ou ESL).

Les approches orientées logiciel se servent d'un modèle logiciel sous forme de bibliothèques d'exécution pour modéliser l'interaction avec le matériel [Des 02] [Mag 05]. L'application peut être décrite dans un langage de haut niveau ou générée à partir d'une description UML ou toute autre spécification basée sur un modèle (model based design) [Bal 06] [Chen 05] [Gil 04][Kan 06] [Mod] [Vand 06]. La construction d'une pile logicielle se compose de la compilation de ce code et de son lien avec les bibliothèques d'exécution. La bibliothèque est définie séparément pour chaque processeur et peut être très sophistiquée. De telles approches ont déjà été appliquées pour les architectures SoC (par exemple YAPI [Koc 00]) comme la plateforme de Trimedia comprenant un DSP et un MIPS. Les bibliothèques sont difficiles à porter sur d'autres processeurs, ce qui rend cette approche inutilisable pour des architectures hétérogènes MPSoC qui ont besoin d'un temps rapide de mise sur le marché et d'une exploration d'architecture pour répondre aux exigences de performance. En fait, le

portage de la bibliothèque est fastidieuse et implique un long temps de conception et une grande complexité dans le débogage du logiciel.

L'approche orientée matériel exécute le logiciel final sur une plateforme virtuelle et correspond aux modèles classiques de cosimulation matériel/logiciel utilisant des simulateurs de jeux d'instructions (Instruction Set Simulator ou ISS) [Row 09] [Sem 00]. Ces techniques exigent que tous les logiciels et les matériels soient entièrement spécifiés. Ainsi, la validation du logiciel se produit trop tard et le processus de débogage peut être trop coûteux et fastidieux.

Les approches axées sur l'ESL utilisent des APIs haut niveau pour abstraire l'interface matériel-logiciel, par exemple DSOC [Pau 06] ou TTL [Van 04]. Cette approche permet la génération et validation automatique d'un prototype virtuel d'un modèle de niveau système, mais la génération de la couche HdS est effectuée en une seule étape, ce qui implique généralement l'utilisation de systèmes de communication prédéfinis. D'ailleurs, l'écart entre le modèle niveau système et le code produit rend le débogage des piles logicielles plus difficile car l'identification des différentes sources d'erreurs n'est pas évidente.

Le flot de conception de logiciel proposé combine tous les avantages des trois premières méthodes décrites précédemment. Il commence par un modèle haut niveau de l'application en Simulink permettant la simulation fonctionnelle rapide du modèle d'application. Il utilise des plateformes spécifiques qui intègrent les particularités de l'architecture matérielle finale permettant l'estimation de performances. Il abstrait l'interface matérielle- logicielle à l'aide d'APIs haut niveau qui cachent beaucoup de détails d'architecture lors de la description de l'application. Même si le partitionnement et la répartition sont explicites dans notre modèle, le flot fournit toujours un niveau d'abstraction suffisant pour produire un gain significatif de productivité. En outre, l'un des principaux avantages de l'approche présentée dans ce document est le débogage graduel des composants de la pile logicielle.

Notre approche est dérivée d'une méthode de conception orientée plateforme [Vin 04]. Cette méthode met l'accent sur la création de couches d'abstraction dans le flot de conception et étudie les propriétés sémantiques à travers ces différentes couches. Il se base sur une méthodologie structurée pour développer des flots de conception de logiciel économiquement viables.

Dans cette thèse, l'accent repose sur le débogage systématique du logiciel, qui est à réaliser en structurant la pile logicielle en couches bien définies et en générant et utilisant des

plateformes de développement logiciel et des modèles exécutables à différents niveaux d'abstraction afin de permettre le débogage séparé des différentes couches logicielles.

5. Contribution

La contribution principale de cette thèse est la définition et le développement d'un flot de conception et de validation de logiciel pour les MPSoC. L'approche proposée commence par un modèle de haut niveau de l'application décrit en Simulink, permettant la simulation fonctionnelle rapide du modèle d'application. Ensuite, le flot génère le code logiciel et la plateforme de développement logiciel correspondant. Les spécificités de la plateforme sont prises en compte dans les modèles abstraits de l'architecture et permettent l'estimation précise des performances de calcul et communication. Le flot proposé abstrait les interfaces de matériel/logiciel en employant des APIs haut niveau, qui cachent les détails liés à l'architecture.

En outre, l'un des principaux avantages de l'approche présentée dans ce document est la génération et la validation progressive des différents composants de la pile logicielle. Ceci rend le débogage de l'application et du HdS plus facile, ouvrant de plus de nouvelles possibilités comme l'estimation de la performance très tôt dans le flot de conception et l'exploration de différents schémas de communication.

La programmation des MPSoC signifie une production efficace du logiciel fonctionnant sur le MPSoC en utilisant les ressources disponibles de l'architecture pour la communication et la synchronisation. Ceci touche deux aspects : la génération et la validation des piles logicielles pour les MPSoC et la génération et la validation de la communication pour les MPSoC.

Comme le montre la figure 4, le flot de conception de logiciel commence par des spécifications de l'application et de l'architecture. L'application est composée d'un ensemble de fonctions. La spécification de l'architecture représente la vue globale de l'architecture, composée de plusieurs sous-systèmes matériels et logiciels.

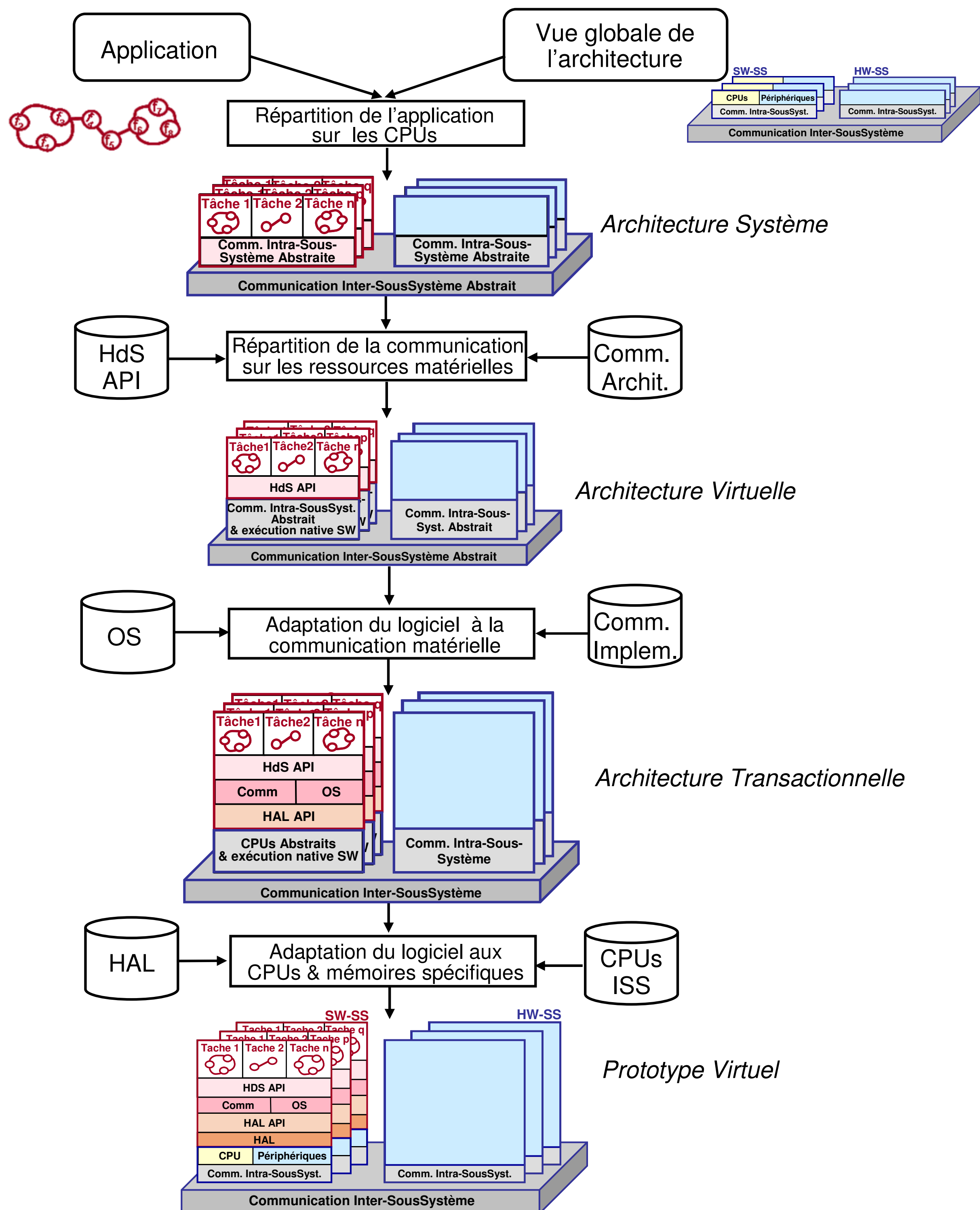


Figure 4. Flot de conception logiciel pour MPSoC

Les étapes principales de la programmation des MPSoC sont :

- Partitionnement de l'application et répartition de l'application sur l'architecture ciblée, ou Conception d'Architecture Système

- Répartition de la communication sur les ressources matérielles disponibles de l'architecture, ou Conception d'Architecture Virtuelle
- Adaptation du logiciel à la communication matérielle spécifique, ou Conception d'Architecture Transactionnelle
- Adaptation du logiciel aux processeurs et mémoires spécifiques, ou Conception du Prototype Virtuel

Le résultat de chacune de ces quatre phases représente une étape dans le processus de raffinement du logiciel et de la communication. Le raffinement est un processus progressif. À chaque étape, d'autres composants logiciels et détails de communication sont intégrés aux composants précédemment produits et validés. Le passage progressif du haut niveau au bas niveau doit être validé à chaque étape de conception. La validation est habituellement effectuée par analyse formelle ou simulation. Dans cette thèse, on utilise la validation basée sur la simulation pour assurer que le comportement du système respecte les spécifications initiales.

La validation et le débogage du logiciel sont effectués par l'exécution du code logiciel sur la plateforme de développement correspondante. Le débogage est effectuée à l'aide des programmes standard tels que le débogueur GNU ou en traçant les signaux SystemC pendant la simulation. Le débogage est un processus itératif car les différents composants logiciels ont besoin de différents niveaux de détail. Par exemple, le débogage du code des tâches de l'application n'a pas besoin de l'implémentation du protocole de synchronisation entre les processeurs, tandis que le débogage de l'intégration du code de tâches avec le système d'exploitation exige ce genre de détail. L'interaction matériel/logiciel détaillée permet le débogage de ce code logiciel de bas niveau spécifique à l'architecture.

5.1. Conception de l'Architecture Système

La première étape du flot de conception logiciel représente le partitionnement et la répartition de l'application sur l'architecture ciblée. Pendant cette étape, les interactions entre l'application et l'architecture sont définis : le nombre de tâches de l'application qui peuvent être exécutées en parallèle, la granularité de ces tâches et l'association entre les tâches et les processeurs qui les exécutent. Le résultat de cette étape est généralement la décomposition de l'application en tâches et la correspondance entre les tâches et processeurs. Cette étape

s'appelle également *conception d'architecture système*, et le modèle résultant est le *modèle d'architecture système*.

Le modèle d'architecture système représente une description fonctionnelle des spécifications de l'application, combinée à l'information de partitionnement et de répartition. Les aspects liés au modèle d'architecture (par exemple les unités de calcul disponibles dans la plateforme matérielle ciblée) sont combinés dans le modèle de l'application (par exemple plusieurs tâches exécutées sur les différents processeurs). Ainsi, le modèle d'architecture système exprime le parallélisme dans l'application regroupant les fonctions dans des tâches et les tâches dans des sous-systèmes. Il rend également explicite les unités de communication pour abstraire les protocoles de communication entres les tâches à l'intérieur d'un sous-système et les protocoles de communication entres les différents sous-systèmes.

Les langages permettant de réaliser des modèles à ce niveau sont des langages de haut niveau comme Simulink. La conception de l'application en Simulink nous fournit les différentes tâches qui seront transposées sur la plateforme ainsi que la répartition de ces tâches et sous-systèmes sur l'architecture, et décrit les divers chemins de communication.

Le chapitre 2 est consacré à cette étape de conception. Des différentes architectures système sont construites utilisant Simulink pour différentes applications multimédia, (telles que le décodeur Motion JPEG et l'encodeur vidéo H.264), partitionnées sur différentes architectures MPSoC. Une application de traitement de signal basée sur Token Ring est utilisée pour illustrer les différents concepts de base. La simulation du modèle de l'architecture système a permis de valider la fonctionnalité de ces applications.

Le modèle Simulink a été annoté avec des paramètres spécifiques à l'architecture matérielle et logicielle. Les paramètres spécifiques à l'architecture matérielle sont utilisés pour identifier les différentes ressources matérielles fournies par l'architecture (type de processeur, type de mémoire, type d'accès à la mémoire, type de communication entre les différents processeurs, etc.). Les paramètres spécifiques à l'architecture logicielle sont utilisés pour identifier les différents composants logiciels qui seront intégrées dans la pile logicielle (système d'exploitation, primitives de communication utilisées pour l'échange des données entre les tâches qui s'exécutent sur le même processeur, etc.). Ces paramètres permettent la génération et la conception automatique de l'architecture virtuelle, de l'architecture transactionnelle et du prototype virtuel.

5.2. Conception de l'Architecture Virtuelle

La deuxième étape du flot proposé représente la répartition des communications sur les ressources de la plateforme matérielle. Lors de cette phase, basée sur le mécanisme de communication, les différentes FIFOs utilisées pour la communication entre les différentes tâches sont reparties sur les ressources matérielles disponibles dans l'architecture pour implémenter le protocole indiqué. Par exemple, une unité de communication FIFO peut être implémentée dans une FIFO matérielle ou une mémoire partagée. Le code des tâches est adapté au mécanisme de communication grâce aux APIs de l'HdS. Cette étape s'appelle également *conception d'architecture virtuelle* et le modèle résultant s'appelle *modèle d'architecture virtuelle*.

Dans ce modèle, la partie matérielle représente les sous-systèmes logiciels, les composants mémoire qui interviennent dans les communications et les supports de communication entre les sous-systèmes. La partie logicielle représente le code applicatif et une interface logicielle qui permet à l'application d'accéder aux ressources de la plateforme, principalement pour la mise en œuvre des communications.

Le chapitre 3 est consacré à la conception d'architecture virtuelle. La conception de l'architecture virtuelle a été effectuée dans SystemC pour les applications suivantes : le Token Ring exécuté sur l'architecture 1AX, le décodeur Motion JPEG partitionné sur l'architecture Diopsis RDT avec bus AMBA et l'encodeur H.264 exécuté sur l'architecture Diopsis R2DT avec réseau sur puce (NoC ou Network on Chip). La conception de l'architecture virtuelle utilise les paramètres matériels et logiciels annotés dans l'architecture système. La simulation du modèle de l'architecture virtuelle a permis la validation du partitionnement de l'application, la validation du code des tâches et l'estimation des performances de communication.

5.3. Conception de l'Architecture Transactionnelle

L'étape suivante du flot proposé se compose de l'adaptation du logiciel à la communication matérielle spécifique. Pendant cette étape, les aspects liés au protocole de communication sont détaillés, par exemple le mécanisme de synchronisation entre les différents processeurs fonctionnant en parallèle devient explicite. Le code logiciel doit être adapté à la méthode de synchronisation, en utilisant par exemple des événements ou des sémaphores. Ceci peut être fait en employant les services de l'OS et des composants de communication de la pile logicielle. Cette phase d'intégration de l'OS et de la communication

est également appelée *conception d'architecture transactionnelle*. Le modèle résultant s'appelle *modèle d'architecture transactionnelle*.

Dans ce modèle, la partie logicielle est enrichie d'un système d'exploitation et d'un code logiciel gérant les entrées-sorties spécifiques. L'architecture matérielle de chaque sous-système logiciel devient plus explicite.

Le chapitre 4 est consacré à la conception d'architecture transactionnelle. La conception de l'architecture transactionnelle a été effectuée dans SystemC pour les applications suivantes : le Token Ring exécuté sur l'architecture 1AX, le décodeur Motion JPEG partitionné sur l'architecture Diopsis RDT avec bus AMBA et l'encodeur H.264 exécute sur l'architecture Diopsis R2DT avec réseau sur puce. La conception de l'architecture transactionnelle utilise les paramètres matériels et logiciels annotés dans l'architecture système.

La simulation du modèle de l'architecture transactionnelle a permis la validation du code des tâches avec le système d'exploitation et du composant de communication de la pile logicielle. L'architecture transactionnelle a permis aussi l'estimation plus précise de la performance de communication.

5.4. Conception du Prototype Virtuel

La dernière étape correspond à l'adaptation du logiciel aux processeurs spécifiques Cela inclut l'intégration du logiciel dépendant de matériel (HAL) dans la pile logicielle pour permettre l'accès bas niveau aux ressources matérielles. L'étape est également connue sous le nom de *conception de prototype virtuel*. Le modèle résultant s'appelle *modèle de prototype virtuel*. Dans ce modèle, les piles logicielles sont complétées par l'implémentation de l'API HAL et exécutées sur un simulateur d'instructions de CPU (*Instruction Set Simulator* ou *ISS*).

Ces différentes étapes du flot global correspondent à la génération et la validation des différents composants logiciels à différents niveaux d'abstraction.

Le chapitre 5 est consacré à la conception de prototype virtuel. La conception du Prototype Virtuel a été effectuée dans SystemC pour les applications suivantes: le Token Ring exécuté sur l'architecture 1AX, le décodeur Motion JPEG exécuté sur un seul processeur (ARM, DSP) et l'encodeur H.264 exécuté sur un seul processeur (ARM). La conception du prototype virtuel utilise les paramètres matériels et logiciels annotés dans l'architecture système.

La simulation du modèle du prototype virtuel a permis la validation de la pile logicielle finale et des mesures précises pour les performances.

Dans cette thèse, le flot proposé a été appliqué avec succès pour la génération et la validation du logiciel pour plusieurs architectures MPSoC complexes qui exécutent plusieurs applications multimédia, comme l'encodeur vidéo H.264, le décodeur d'images Motion JPEG, le décodeur audio MP3 et l'encodeur audio Vocodeur. Les plateformes de développement en SystemC à différents niveaux d'abstraction sont automatiquement générées à partir de la description initiale en Simulink. Les architecture MPSoC considérées contiennent plusieurs processeurs différents interconnectés par un bus ou un réseau sur puce.

Multilevel Programming Environment for Heterogeneous MPSoC Architectures

Table of Contents

1. INTRODUCTION 37

1.1. Context 39

1.2. MPSoC Programming Steps 40

1.3. Hardware/Software Abstraction Levels 43

1.3.1. The Concept of Hardware/Software Interface 44

1.3.2. Software Execution Models with Abstract Hardware/Software Interfaces 45

1.4. The MPSoC Architecture 49

1.5. Software Stack for MPSoC 52

1.5.1. Definition of the Software Stack 53

1.5.2. Software Stack Organization 53

1.5.2.1. Application Layer 54

1.5.2.2. HdS Layer 54

1.5.2.2.1. Operating System 55

1.5.2.2.2. Communication Software Component 56

1.5.2.2.3. Hardware Abstraction Layer 57

1.6. The Concept of Mixed Architecture/Application Model 57

1.6.1. Definition of the Mixed Architecture/Application Model 57

1.6.2. Execution Model for Mixed Architecture/Application Model 58

1.6.2.1. Execution model described in SystemC 58

1.6.2.2.. Execution model described in Simulink 61

1.7. Examples of Heterogeneous MPSoC Architectures 63

1.7.1. 1AX with AMBA Bus 64

1.7.2. Diopsis RDT with AMBA Bus 66

1.7.3. Diopsis R2DT with NoC 69

1.8. Examples of Multimedia Applications 71

1.8.1. Token Ring Functional Specification 72

1.8.2. Motion JPEG Decoder Functional Specification 73

1.8.3. H.264 Encoder Functional Specification 75

1.9. Conclusions 77

2. SYSTEM ARCHITECTURE DESIGN 79

2.1. Introduction 81
2.1.1. Mapping Application on Architecture 81
2.1.1.1 The Mapping 81
2.1.1.2 The Design Space Exploration 83
2.1.2. Definition of the System Architecture 84
2.1.3. Global Organization of the System Architecture 86
2.2. Basic Components of the System Architecture Model 88
2.2.1. Functions 89
2.2.2. Communication 90
2.3. Modeling System Architecture in Simulink 90
2.3.1. Writing Style, Design Rules and Constraints in Simulink 90
2.3.1.1. Constraints on the Simulink standard blocks 90
2.3.1.2. Constraints on the S-Functions 91
2.3.1.3. Constraints on the communication 92
2.3.2. Software at System Architecture Level 93
2.3.3. Hardware at System Architecture Level 93
2.3.4. Hardware-Software Interface at System Architecture Level 93
2.4. Execution Model of the System Architecture 94
2.5. Design Space Exploration of System Architecture 95
2.5.1. Goal of Performance Evaluation 95
2.5.2. Architecture/Application Parameters 95
2.5.3. Performance Measurements 99
2.5.4. Design Space Exploration 99
2.6. Application Examples at the System Architecture Level 100
2.6.1. Motion JPEG Application on Diopsis RDT 100
2.6.2. H.264 Application on Diopsis R2DT 104
2.7. State of the Art and Research Perspectives 108
2.7.1. State of the Art 108
2.7.2. Research Perspectives 110
2.8. Conclusions 112
3. VIRTUAL ARCHITECTURE DESIGN 113
3.1. Introduction 115
3.1.1. Definition of the Virtual Architecture 115
3.1.2. Global Organization of the Virtual Architecture 116

3.2. Basic Components of the Virtual Architecture Model 118
3.2.1. Software Components 118
3.2.2. Hardware Components 119
3.3. Modeling Virtual Architecture in SystemC 119
3.3.1. Software at Virtual Architecture Level 119
3.3.2. Hardware at Virtual Architecture Level 122
3.3.3. Hardware-Software Interface at Virtual Architecture Level 126
3.4. Execution Model of the Virtual Architecture 128
3.5. Design Space Exploration of Virtual Architecture 128
3.5.1. Goal of Performance Evaluation 128
3.5.2. Architecture/Application Parameters 129
3.5.3. Performance Measurements 130
3.5.4. Design Space Exploration 132
3.6. Application examples at the Virtual Architecture Level 133
3.6.1. Motion JPEG Application on Diopsis RDT 133
3.6.2. H.264 Application on Diopsis R2DT 137
3.7. State of the Art and Research Perspectives 142
3.7.1. State of the Art 142
3.7.2. Research Perspectives 143
3.8. Conclusions 144
4. TRANSACTION ACCURATE ARCHITECTURE DESIGN 145
4.1. Introduction 147
4.1.1. Definition of the Transaction Accurate Architecture 147
4.1.2. Global Organization of the Transaction Accurate Architecture 148
4.2. Basic Components of the Transaction Accurate Architecture Model 151
4.2.1. Software Components 151
4.2.2. Hardware Components 152
4.3. Modeling Transaction Accurate Architecture in SystemC 152
4.3.1. Software at Transaction Accurate Architecture Level 153
4.3.2. Hardware at Transaction Accurate Architecture Level 157
4.3.3. Hardware-Software Interface at Transaction Accurate Architecture Level 160
4.4. Execution Model of the Transaction Accurate Architecture 161
4.5. Design Space Exploration of Transaction Accurate Architecture 162
4.5.1. Goal of Performance Evaluation 162

4.5.2. Architecture/Application Parameters 163
4.5.3. Performance Measurements 163
4.5.4. Design Space Exploration 164
4.6. Application Examples at the Transaction Accurate Architecture Level 165
4.6.1. Motion JPEG Application on Diopsis RDT 165
4.6.2. H.264 Application on Diopsis R2DT 169
4.7. State of the Art and Research Perspectives 177
4.7.1. State of the Art 177
4.7.2. Research Perspectives 178
4.8. Conclusions 179
5. VIRTUAL PROTOTYPE DESIGN 181
5.1. Introduction 183
5.1.1. Definition of the Virtual Prototype 183
5.1.2. Global Organization of the Virtual Prototype 184
5.2. Basic Components of the Virtual Prototype Model 185
5.2.1. Software Components 185
5.2.2. Hardware Components 186
5.3. Modeling Virtual Prototype in SystemC 186
5.3.1. Software at Virtual Prototype Level 187
5.3.2. Hardware at Virtual Prototype Level 192
5.3.3. Hardware-Software Interface at Virtual Prototype Level 194
5.4. Execution Model of the Virtual Prototype 194
5.5. Design Space Exploration of Virtual Prototype 195
5.5.1. Goal of Performance Evaluation 195
5.5.2. Architecture/Application Parameters 196
5.5.3. Performance Measurements 196
5.5.4. Design Space Exploration 197
5.6. Application Examples at the Virtual Prototype Level 198
5.6.1. Motion JPEG Application on Diopsis RDT 198
5.6.2. H.264 Application on Diopsis R2DT 200
5.7. State of the Art and Research Perspectives 202
5.7.1. State of the Art 202
5.7.2. Research Perspectives 203
5.8. Conclusions 204

6. CONCLUSIONS AND FUTURE PERSPECTIVES 207
6.1. Conclusions 209
6.2. Future Perspectives 211
References 215
Publications 225

List of Figures

Figure 1. MPSoC Hardware-Software Architecture 39
Figure 2. MPSoC Programming Steps 41
Figure 3. Software Execution Models at Different Abstraction Levels 46
Figure 4. MPSoC Architecture 49
Figure 5. Software Stack Organization 53
Figure 6. SystemC Simulation Steps 60
Figure 7. Simulink Simulation Steps 62
Figure 8. 1AX MPSoC Architecture 64
Figure 9. Diopsis RDT Heterogeneous Architecture 67
Figure 10. Target Diopsis based Architecture 68
Figure 11. Diopsis R2DT with Hermes NoC 69
Figure 12. Token Ring Functional Specification 72
Figure 13. Motion JPEG Decoder 73
Figure 14. H.264 Encoder Algorithm Main Profile 76
Figure 15. Mapping Token Ring on the 1AX architecture 83
Figure 16. Design Space Exploration 84
Figure 17. Global View of the System Architecture 85
Figure 18. System Architecture Model of Token Ring 88
Figure 19. User-defined C-Function 91
Figure 20. DFT Function of the Token Ring 92
Figure 21. Mapping Motion JPEG on Diopsis RDT 101
Figure 22. System Architecture Example: MJPEG Mapped on Diopsis 102
Figure 23. Mapping H.264 on Diopsis R2DT 105
Figure 24. H.264 Encoder System Architecture Model in Simulink 106
Figure 25. Global View of the Virtual Architecture 116
Figure 26. Task T2 Code 121
Figure 27. SystemC Code for the Top Module 123
Figure 28. SystemC Code for the ARM-SS Module 124
Figure 29. Example of Implementation of Communication Channels 125
Figure 30. SystemC Main Function 126

Figure 31. Example of Hardware/Software Interface 127
Figure 32. Waveforms Traced during the Token Ring Simulation 132
Figure 33. Global View of Diopsis RDT running MJPEG 134
Figure 34. Virtual Architecture Simulation for Motion JPEG 137
Figure 35. Global View of Diopsis R2DT running H.264 139
Figure 36. Abstract Hermes NoC at Virtual Architecture Level... 140
Figure 37. Words transferred through the Hermes NoC 141
Figure 38. Global View of the Transaction Accurate Architecture 148
Figure 39. Initialization of the Tasks running on ARM7 154
Figure 40. Implementation of *recv_data(…)* API 155
Figure 41. Example of Task Header File 156
Figure 42. Data Structure of Tasks'Ports 156
Figure 43. Implementation of the *__schedule()* Service of OS 157
Figure 44. SystemC Code for the Top Module 158
Figure 45. SystemC Code for the ARM7-SS Module 159
Figure 46. SystemC Clock 160
Figure 47. Implementation of the *__ctx_switch* HAL API 160
Figure 48. Execution Model of the Software Stacks running on the ARM7 and XTENSA Processors 162
Figure 49. Transaction Accurate Architecture Model of the Diopsis RDT Architecture running Motion JPEG Decoder Application 166
Figure 50. MJPEG Simulation Screenshot 167
Figure 51. Global View of the Transaction Accurate Architecture for Diopsis R2DT with Hermes NoC running H.264 Encoder Application 169
Figure 52. Hermes NoC in Mesh Topology at Transaction Accurate Level 171
Figure 53. Total KBytes Transmitted through the Mesh 172
Figure 54. Hermes NoC in Torus Topology at Transaction Accurate Level 173
Figure 55. Simulation Screenshot of H.264 Encoder Application running on Diopsis R2DT with Torus NoC 175
Figure 56. IP Cores Mapping Schemes A and B over the NoC 175
Figure 57. Global View of the Virtual Prototype 184
Figure 58. HAL Implementation for Context Switch on ARM7 Processor 188
Figure 59. Enabling and Disabling ARM Interrupts 188
Figure 60. Example of Compilation Makefile for ARM7 Processor 189

Figure 61. Load and Execution Memory View 190
Figure 62. Example of Scatter Loading Description File for the ARM Processor 191
Figure 63. Example of Initialization Sequence for the ARM Processor 192
Figure 64. SystemC Code of the ARM7-SS Module 193
Figure 65. Execution Model of the Virtual Prototype 195
Figure 66. Global View of the Virtual Prototype for Diopsis RDT with AMBA Bus running Motion JPEG Decoder Application 199
Figure 67. Execution Clock Cycles of Motion JPEG Decoder QVGA 199
Figure 68. Global View of the Virtual Prototype for Diopsis R2DT with Hermes NoC running H.264 Encoder Application 201
Figure 69. Execution Clock Cycles of H.264 Encoder, Main Profile, QCIF Video Format . 201
Figure 70. Program and Memory Size 202

List of Tables

Table 1. Task code generation for Motion JPEG ... 133

Table 2. Messages through the AMBA bus ... 136

Table 3. Task code generation for H.264 Encoder ... 138

Table 4. Results captured in Hermes NoC using DXM as communication scheme ... 141

Table 5. Memory accesses ... 168

Table 6. Mesh Noc Routing Requests ... 171

Table 7. Torus Noc Routing Requests ... 174

Table 8. Torus Noc Amount of Transmitted Data [Bytes] ... 174

Table 9. Execution and Simulation Times of the H.264 Encoder for Different Interconnect, Communication and IP Mappings ... 176

Table 10. ARM7 and ARM9 processors family ... 196

Chapter 1

INTRODUCTION

This chapter introduces the definitions of the basic concepts used in the document. The chapter details the software and hardware organization for the heterogeneous MPSoC architectures ant summarizes the main steps in programming MPSoC. The software design represents an incremental process performed at four MPSoC abstraction levels (system architecture, virtual architecture, transaction accurate architecture and virtual prototype). At each design step, different software components are generated and validated using hardware simulation models. The overall design flow is given in this chapter. Examples of target architectures and applications, which will be used in the remaining part of this document, are described.

1.1. Context

Current multimedia and telecom applications such as MPEG 2/4, H.263/4, CDMA 2000, WCDMA, and MP3 require heterogeneous multiprocessor system on chip (MPSoC) architectures in order to achieve computation and communication performances [Mey 06]. Heterogeneous MPSoC includes different kinds of processing units (DSP, microcontroller, ASIP, etc) and different communication schemes (fast links, non standard memory organization and access). This kind of heterogeneous architectures provides highly concurrent computation and flexible programmability.

Typical heterogeneous platforms used in industry are TI OMAP [TI], ST Nomadik [Nom], Philips Nexperia [Nex] and Diopsis D940 [Dio]. They incorporate a DSP processor and a microcontroller, communicating via efficient, but sophisticated infrastructure.

Heterogeneous MPSoC architectures may be represented as a set of software and hardware processing subsystems which interact via a communication network (figure 1) [Cul 98].

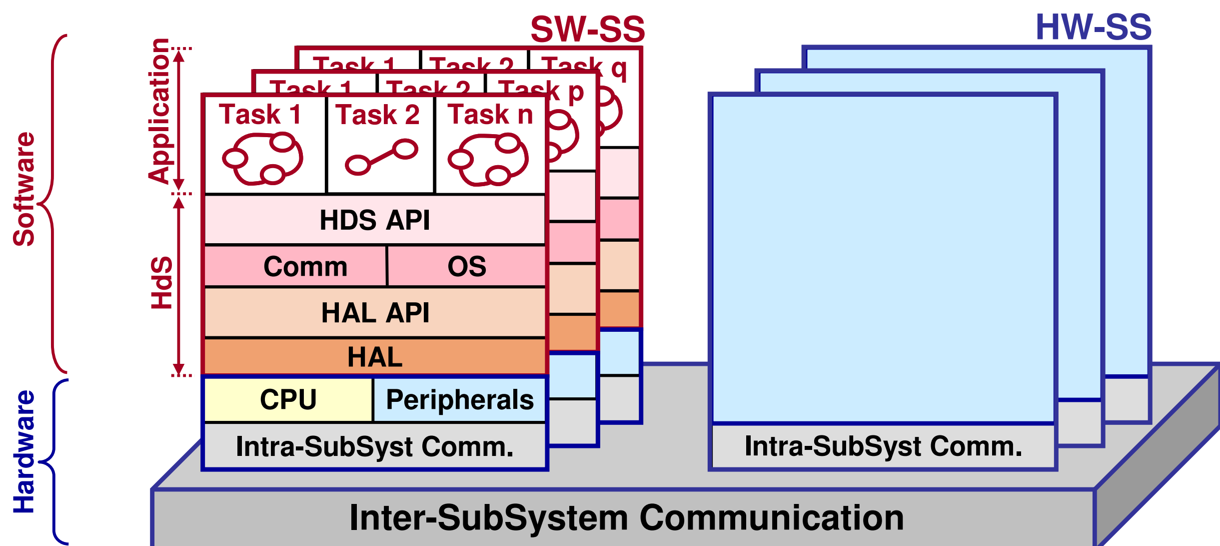


Figure 1. MPSoC Hardware-Software Architecture

A software subsystem is a programmable subsystem, namely a processor subsystem. This integrates different hardware components including a processing unit for computation (CPU), specific local components such as local memory, data and control registers, hardware accelerators, interrupt controller, DMA engine, synchronization components such as mailbox or semaphores and specific I/O components or other peripherals.

Each processor subsystem executes a specific software stack organized in 2 layers: the application and the Hardware dependent Software (HdS) layers. The application layer is associated with the high level behavior of the heterogeneous functions composing the target application. The HdS layer is associated with the hardware dependent low level software behavior, such as interrupts routine services, context switch, specific I/O control and tasks scheduling. In fact, the HdS layer includes 3 components: Operating System (OS), specific I/O communication (Comm) and the Hardware Abstraction Layer (HAL). These different components are based on well defined primitives or Application Programming Interfaces (APIs) in order to pass from one software layer to another.

A hardware subsystem represents specific hardware component that implements specific functionalities of the application or a global memory subsystem accessible by the processing units.

The rest of this document is organized as it follows: Chapter 1 introduces the context of MPSoC design, the difficulties of programming these complex architectures, the design and validation flow of the multiple software stacks running on the different processor subsystems, the adopted different abstraction levels and the definition of some concepts later used in this document. Chapter 2, 3, 4 and 5 details the software design and validation for MPSoC at four abstraction levels, namely the system architecture, virtual architecture, transaction accurate architecture, respectively the virtual prototype design. Chapter 6 draws conclusions and proposes future research perspectives.

1.2. MPSoC Programming Steps

Programming an MPSoC means to generate software running on the MPSoC efficiently by using the available resources of the architecture for communication and synchronization. This concerns two aspects: software stack generation and validation for the MPSoC and communication mapping on the available hardware communication resources and validation for MPSoC.

As shown in figure 2, the software design flow starts with an application and an abstract architecture specification. The application is made of a set of functions. The architecture specification represents the global view of the architecture, composed of several hardware and software subsystems.

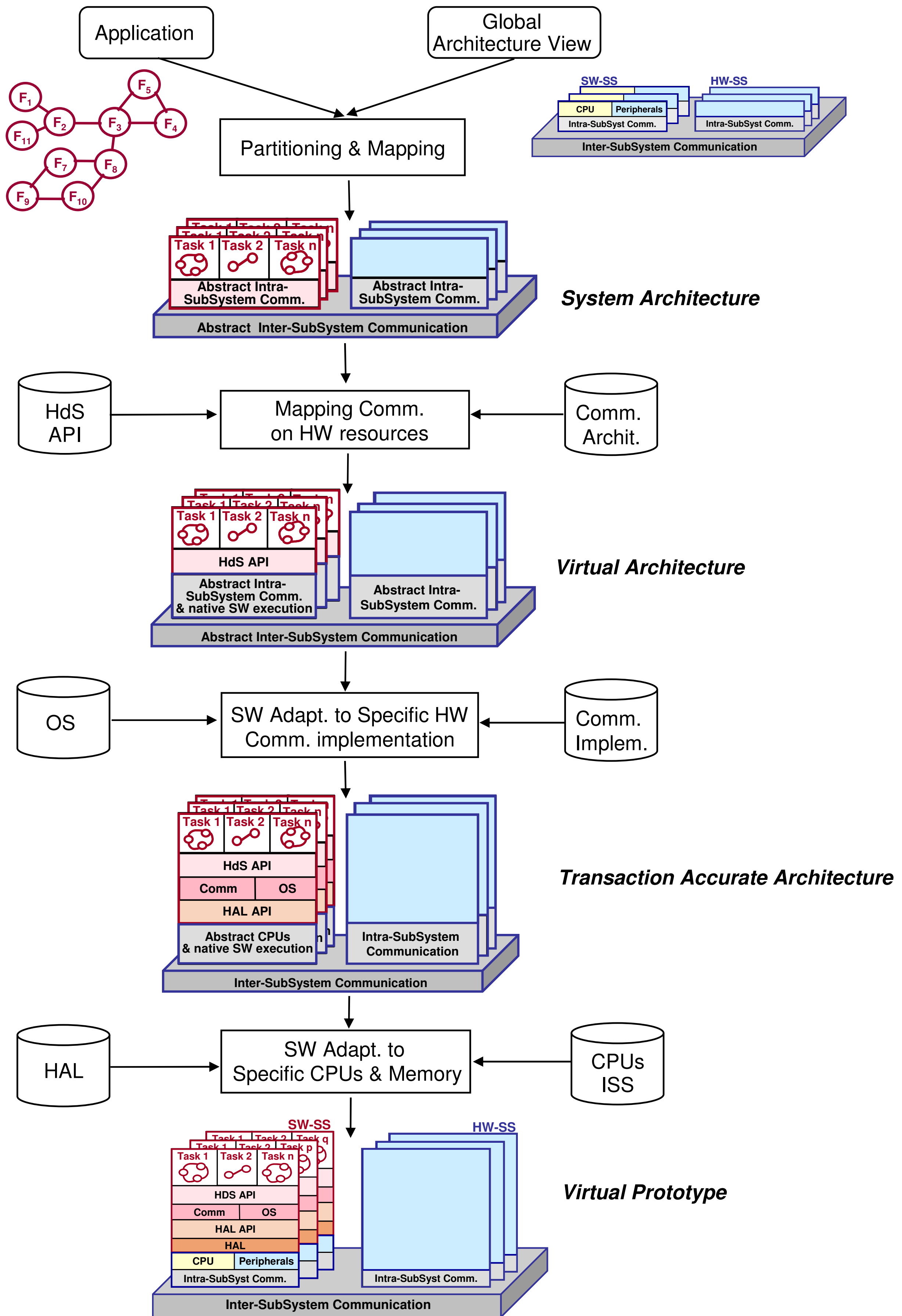


Figure 2. MPSoC Programming Steps

The main steps in programming the MPSoC architecture are:

- Partitioning and mapping the application onto the target architecture subsystems
- Mapping application communication on the available hardware communication resources of the architecture
- Software adaptation to specific hardware communication protocol implementation
- Software adaptation to detailed architecture implementation (specific processors and memory architecture).

The result of each of these four phases represents a step in the software and communication refinement process. The refinement is an incremental process. At each stage, additional software component and communication architecture details are integrated with the previously generated and validated components. This conducts to a gradual transformation of a high level representation with abstract components into a concrete low level executable software code. The transformation has to be validated at each design step. The validation is performed by formal analysis, simulation or combining simulation with formal analysis [Kun 06]. In the following, we will use simulation-based validation to ensure that the system behavior respects the initial specification.

During the *partitioning and mapping* of the application on the target architecture, the relationship between application and architecture is defined. This refers to the number of application tasks that can be executed in parallel, the granularity of these tasks (coarse grain or fine grain) and the association between tasks and the processors that will execute them. The result of this step is the decomposition of the application into tasks and the correspondence tasks-processors [Thi 07]. This step is also called *System Architecture Design*, and the resulting model is the System Architecture model.

The system architecture model represents a functional description of the application specification, combined with the partitioning and mapping information. Aspects related to the architecture model (e.g. processing units available in the target hardware platform) are combined into the application model (i.e. multiple tasks executed on the processing units). Thus, the system architecture model expresses parallelism in the target application through capturing the mapping of the functions into tasks and the tasks into subsystems. It also makes explicit the communication units to abstract the intra-subsystem communication protocols

(the communication between the tasks inside a subsystem) and the inter-subsystem communication protocols (the communication between different subsystems).

The second step implements the *mapping of communication* onto the hardware platform resources. At this phase, the different links used for the communication between the different tasks are mapped on the hardware resources available in the architecture to implement the specified protocol. For example, a FIFO communication unit can be mapped to a hardware queue, a shared memory or some kind of bus-based device. The task code is adapted to the communication mechanism through the use of adequate HdS communication primitives. This step is also entitled *Virtual Architecture Design* and the resulting model is named Virtual Architecture model.

The next step of the proposed flow consists of *software adaptation to specific communication protocol implementation*. During this stage, aspects related to the communication protocol are detailed, for example the synchronization mechanism between the different processors running in parallel becomes explicit. The software code has to be adapted to the synchronization method, such as events or semaphores. This can be done by using the services of OS and communication components of the software stack. This phase of integrating the OS and communication is also named *Transaction Accurate Architecture Design* and the resulting model is the Transaction Accurate Architecture model.

The last step corresponds to *specific adaptation of the software to the target processors and specific memory map*. This includes the integration of the processor dependent software code into the software stack (HAL) to allow low level access to the hardware resources and the final memory mapping. This step is also known as *Virtual Prototype Design* and the resulting model is called Virtual Prototype model.

These different steps of the global flow correspond to different software components generation and validation at different abstraction levels, as it will be described in the following paragraphs.

1.3. Hardware/Software Abstraction Levels

The structured model of the software stack representation allows generation and validation of the different software components separately [Jer 06]. The different components and layers of the software stack correspond to different abstraction levels. The debug of this

software stack made of several components is one of the MPSoC current design challenges [Mar 06].

In order to validate the software, an execution model is required at each abstraction level to allow debugging the specific software component. The **execution model** represents an abstract architecture model [Roa 07] which allows simulating and validating the software component at each abstraction level. The *execution model* is often called *software development platform* and it is the result of abstracting different components of the target hardware architecture. This abstract architecture model hides details of the underlying implementation of the hardware platform, but ensures a sufficient level of control that the software code can be validated in terms of performance, efficiency and reliable functionality.

The software validation and debug is performed by execution of the software code on a corresponding execution model. The debug is an iterative process because the different software components need different detail levels in order to be validated. For example, the debug of the application tasks code does not need explicit implementation of the synchronization protocol between the processors using mailboxes in the development platform, while the debug of the integration of the tasks code with the OS requires this kind of detail. The detailed hardware-software interaction allows debugging this low level architecture specific software code. All these requirements are considered during the abstraction of the architecture at each design step to build the executable model.

The debug of the software is performed by simulation at the different abstraction levels. Thus, the *system architecture model* simulation is used to debug the application algorithm. The *virtual architecture model* simulation serves to debug the application tasks code. The *transaction accurate architecture model simulation* is used to debug the glue between the application tasks code and OS and communication libraries. The *virtual prototype model* uses Instruction Set Simulators to execute and debug the full software stack.

At all these abstraction levels, the debug process uses standard debugging tools and environments, such as GNU debuggers, or trace waveforms during the simulation, such as SystemC waveforms.

1.3.1. The Concept of Hardware/Software Interface

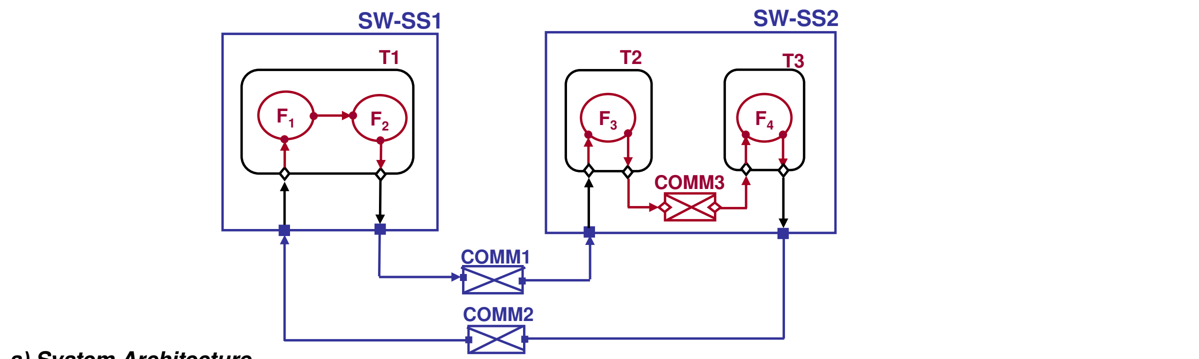
The software generation and validation using an executable model needs abstract hardware/software interfaces including both software and hardware components. The

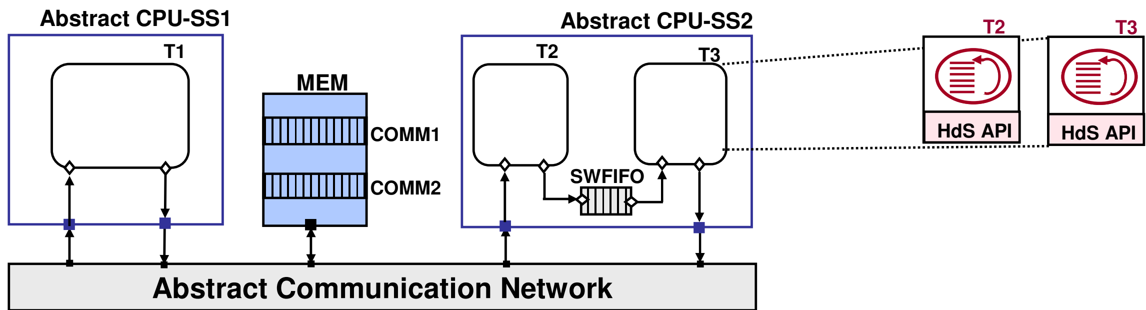
hardware/software interface links the software part with the hardware part of the system. The hardware/software interface needs to handle two different interfaces: one on the software side using APIs and one on the hardware side using wires [Bou 05]. This heterogeneity makes the hardware/software interface design very difficult and time-consuming because the design requires both hardware and software knowledge and their interaction [Jer 05]. The hardware/software interface requires handling many software and hardware architecture parameters. To allow the gradual validation of the software stack, the hardware-software interface needs to be described at the different abstraction levels.

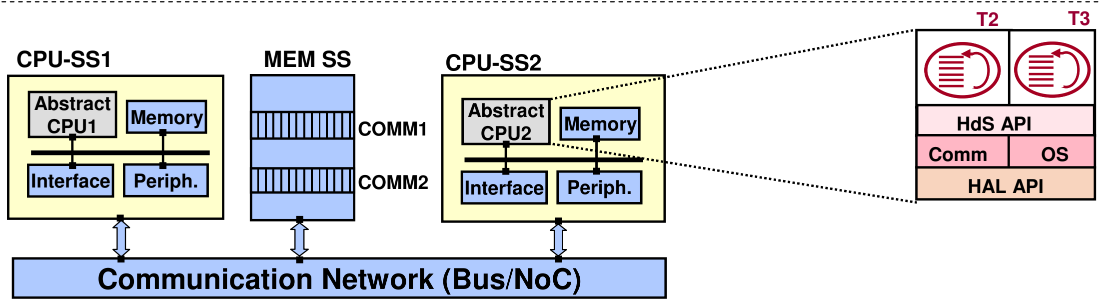
1.3.2. Software Execution Models with Abstract Hardware/Software Interfaces

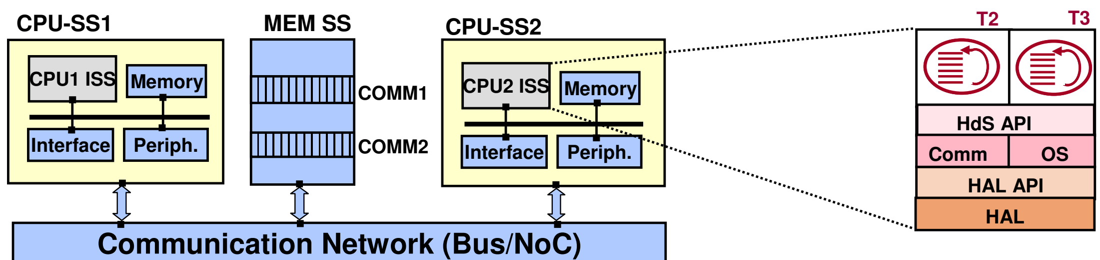
Figure 3 illustrates the software execution models at different abstraction levels for a simplified application made of 3 tasks (T1, T2 and T3), that need to be mapped on an architecture made of 2 processing units and several memory hardware subsystems. For each level, figure 3 shows the software organization, the hardware-software interface and the execution model that will be used to validate the software component at the corresponding abstraction level. The key differentiation between these levels is the way of specifying the hardware-software interfaces and the communication mechanism implementation.

The highest level is the **system architecture** level (figure 3.a). In this case, the software is made of a set of functions grouped into tasks. The function is an abstract view of the behavior of an aspect of the application. Several tasks may be mapped on the same software subsystem. The communication between functions, tasks and subsystems make use of abstract communication links, e.g. standard Simulink links or explicit communication units that correspond to specific communication paths of the target platform. The links and units are annotated with communication mapping information. The corresponding execution model consists of the set of the abstract subsystems. The simulation at this level allows validation of the application's functionality. This model captures both the application and the architecture in addition to the computation and communication mapping. Figure 3.a shows the system architecture model with the following symbols: circles for the functions, rounded rectangular to represent the task, rectangular for the subsystem, crossed rectangular for the communication units between the tasks, filled circles for the ports of the functions, diamonds for the logic ports of the tasks and filled rectangular for group of hardware ports. The dataflow is illustrated by unidirectional arrows.









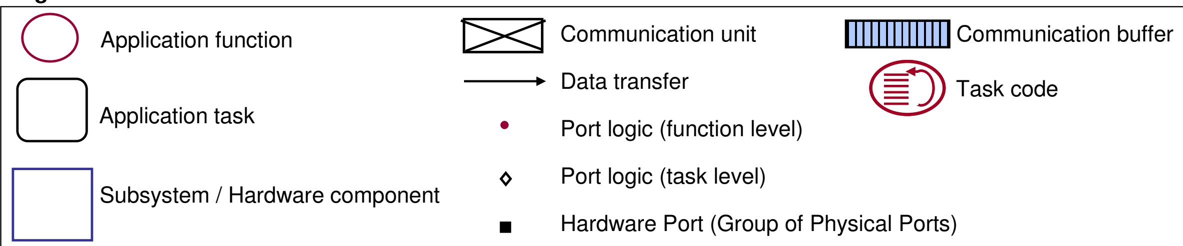


Figure 3. Software Execution Models at Different Abstraction Levels

In this case, the system is made of 2 abstract software subsystems (SW-SS1, SW-SS2) and 2 inter-subsystem communication units (COMM1, COMM2). The SW-SS1 software subsystem encapsulates task T1, while the subsystem SW-SS2 groups together tasks T2 and T3. The intra-subsystem communication between the tasks T2 and T3 inside SW-SS1 is performed through the communication unit COMM3.

The next abstract level is called **virtual architecture** level (figure 3.b). The hardware-software interfaces are abstracted using HdS API that hides the OS and the communication layers. The application code is refined into tasks that interact with the environment using explicit primitives of the HdS API. Each task represents a sequential C code using a static scheduling of the initial application functions. This code is the final application code that will constitute the top layer of the software stacks. The communication primitives of the HdS API access explicit communication components. Each data transfer specifies an end-to-end communication path. For example, the functional primitives *send_mem(ch,src,size)/ recv_mem(ch,dst,size)* may be used to transfer data between the 2 processors using a global memory connected to the system bus, where *ch* represents the communication channel used for the data transfer, *src/dst* the source/destination buffer and *size* the number of words to be exchanged. The communication buffers are mapped on explicit hardware resources.

At the virtual architecture level, the software is executed using an abstract model of the hardware architecture that provides an emulation of the HdS API. The software execution model is composed of these abstract subsystems, explicit interconnection component and storage resources. During the simulation at the virtual architecture level, the software tasks are scheduled by the hardware platform since the final OS is not yet defined. The simulation at this level allows validation of the final code of tasks and may give useful statistics about the communication requirements. The virtual architecture is message accurate in terms of data exchange between the different tasks. Thanks to the HdS APIs, the tasks code remains unchanged for the following levels. In this document, the virtual architecture platform is considered as a SystemC model where the software tasks are executed as SystemC threads.

In the example illustrated in figure 3.b, the system is made of two abstract processor subsystems (CPU1-SS, CPU2-SS) and a global memory (MEM) interconnected through an abstract communication network. The communication units *comm1* and *comm2* are mapped on the global memory and the communication unit *comm3* becomes a software fifo (swfifo).

The next level is called the **transaction accurate architecture** level (figure 3.c). At this level, the hardware-software interfaces are abstracted using a HAL API that hides the

processor's architecture. The code of the software task is linked with an explicit OS and specific I/O software implementation to access the communication units. The resulting software makes use of hardware abstraction layer primitives (HAL_API) to access the hardware resources. This will constitute the final code of the two top layers of the resulting software stack. The data transfers use explicit addresses, e.g. *read_mem(addr, dst, size)/ write_mem(addr, src, size),* where *addr* represents the source, respectively the destination address, *src/dst* represents the local address and *size* the size of the data.

The software is executed using a more detailed development platform to emulate the network component, the explicit peripherals used by the HAL API and an abstract computation model of the processor. During the simulation at this level, the software tasks are scheduled by the final OS, while the communication between tasks mapped on the same processor is also implemented by the OS. The simulation at this level allows validating the integration of the application with the OS and the communication layer. It may also provide precise information about the communication performances. The accuracy of the performance estimation is transaction accurate level. In this document, the transaction accurate architecture is generated as a SystemC model where the software stacks are executed as external processes communicating with the SystemC simulator through the IPC layer of the Linux OS running on the host machine.

In the example illustrated in figure 3.c, the system is made of the 2 processor subsystems (CPU1-SS, CPU2-SS) and the global memory subsystem (MEM-SS) interconnected through an explicit communication network (bus or NoC). Each processor subsystem includes an abstract execution model of the processor core (CPU1, respectively CPU2), local memory, interface and other peripherals. Each processor subsystem executes a software stack made of the application tasks code, communication and OS layers.

Finally, the HAL API and processor are implemented through the use of a HAL software layer and the corresponding processor part for each software subsystem. This represents the **virtual prototype** level (figure 3.d). At the virtual prototype level the communication consists of physical I/Os, e.g. *load/store*. The platform includes all the hardware components such as cache memories or scratchpads. The scheduling of the communication and computation activities for the processors becomes explicit. The simulation at this level allows cycle accurate performance validation and it corresponds to classical hardware/software cosimulation models with Instruction Set Simulators [Row 94]

[Sem 00] [Nic 02] for the processors and RTL components or cycle accurate TLM components for the hardware resources.

In the example illustrated in figure 3.d, the 2 processor subsystems (CPU1-SS, CPU2-SS) include ISS for the execution of the software stack corresponding to CPU1, respectively CPU2. Each processor subsystem executes a software stack made of the application tasks code, communication, OS and HAL layers.

In order to validate the software during the different design steps, different execution models are used adapted to each software abstraction level. In the rest of the document, we use Simulink for the initial simulation at system architecture level, while for the all others we use SystemC.

1.4. The MPSoC Architecture

In the following paragraphs, the definition of the MPSoC architecture will be given.

System on chip (SoC) represents the integration of different computing elements and/or other electronic subsystems into a single integrated circuit (chip). It may contain digital, analog, mixed-signal, and often radio-frequency functions – all on one chip.

Multi-Processor System on Chip (MPSoC) are SoC that may contain one or more types of computing subsystems, memories, input/output devices (I/O) and other peripherals. These systems range from portable devices such as MP3 players, videogame consoles, digital cameras or mobile phones to large stationary installations like traffic lights, factory controllers, engine controllers for automobiles or digital set-top boxes.

The MPSoC architecture is made of three types of components: software subsystems, hardware subsystems and inter-subsystem communication, as illustrated in figure 4.

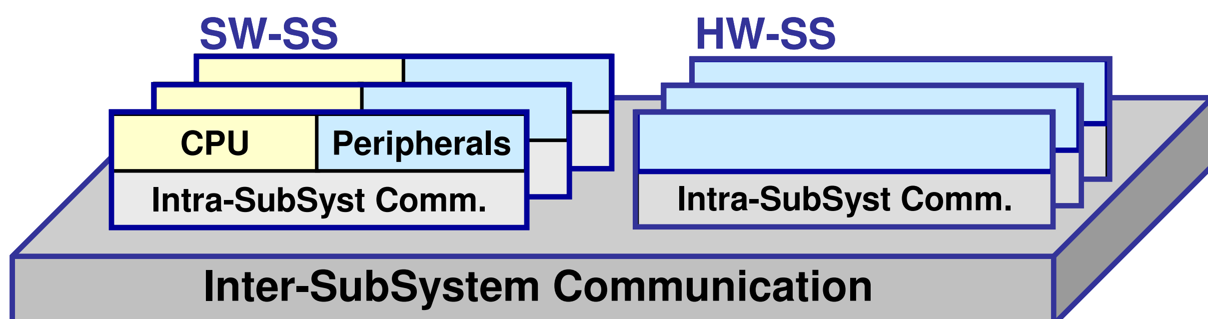


Figure 4. MPSoC Architecture

The **hardware subsystems (HW-SS)** represent custom hardware subsystems that implement specific functionality of an application or global memory subsystems. The HW-SS contain two types of components: intra-subsystem communication and specific hardware components. The hardware components implement specific functions of the target application or represent global memories accessible by the computing subsystems. The intra-subsystem communication represents the communication inside the HW-SS between the different hardware components. This can be in form of a small bus (collection of parallel wires for transmitting address, data and control signals) or point-to-point communication links.

The **software subsystems (SW-SS)** represent programmable subsystems, also called processor nodes of the architecture. The SW-SS include computing resources, intra-subsystem communication and other hardware components, such local memories, I/O components or hardware accelerators. The computing resources represent the processing units or CPUs. The **CPU (Central Processing Unit)** also known as processor core, processing element or shortly processor, executes programs stored in the memory by fetching their instructions, examining them, and then executing them one after another [Tan 99]. There are two types of SW-SS: single core and multi-core. The single-core SW-SS includes a single processor, while the multi-core SW-SS can integrate several processor cores in the same subsystem. The intra-subsystem communication represents the communication inside the SW-SS, e.g. local bus, hardware fifo, point-to-point communication links or other local interconnection network used to interconnect the different hardware components inside the SW-SS.

The **inter-subsystem communication** represents the communication architecture between the different software and hardware subsystems. This can be a hardware FIFO connecting multiple subsystems or a scalable global interconnection network, such as bus or Network on Chip (NoC). Despite most of the buses, the NoC allows simultaneous data transfers, being composed of several links and switches that provide a means to route the information from the source node to the destination node [Cul 98].

Homogeneous MPSoC architectures are made of identical software subsystems incorporating the same type of processors. In the **heterogeneous MPSoC** architectures, different types of processors are integrated on the same chip, resulting different types of software subsystems. These can be GPP (General Purpose Processor) subsystems for control operations of the application; DSP (Digital Signal Processor) subsystems special tailored for data intensive applications such as digital signal applications, or ASIP (Application Specific

Instruction Set Processor) subsystems with a configurable instruction set to fit specific functions of the application.

The different subsystems working in parallel on different parts of the same application must communicate each other to exchange information. There are two distinct MPSoC designs that have been proposed and implemented for the communication models between the subsystems: shared-memory and message passing [Cul 98].

The **shared memory** communication model characterizes the homogeneous MPSoC architecture. The key property of this class is that communication occurs implicitly. The communication between the different CPUs is made through a global shared memory. Any CPU can read or write a word of memory by just executing LOAD and STORE instructions. Besides the common memory, each SW-SS may have some local memory which can be used for program code and those items that need not be shared. In this case, the MPSoC architecture executes a multithreaded application organized as a single software stack.

The **message passing** organization assumes a heterogeneous MPSoC architecture with multiple software stacks running on the non identical software subsystems. The communication between different subsystems is generally made through message passing. The key property of this class is that the communication between the different processors is explicit through I/O operations. The CPUs communicate by sending each other message by using primitives such as *send* and *recv*. There are three types of message passing: synchronous (if the sender executes a *send* operation and the receiver has not yet executed a *receive*, the sender is blocked until the receiver executes the *receive*), buffered or asynchronous blocking (when a message is sent before the receiver is ready, the message is buffered somewhere, for example in a mailbox, until the receiver takes it out; thus the sender can continue after a *send* operation, if the receiver is busy with something else) and asynchronous non-blocking (the sender may continue immediately after making the communication call) [Tan 99].

Heterogeneous MPSoC generally combines both models to integrate a massive number of processors on a single chip [Pau 06]. Future heterogeneous MPSoC will be made of few heterogeneous subsystems where each may include a massive number of the same processor to run a specific software stack [Jer 06].

Besides the hardware architecture previously presented, the MPSoC means also software running on hardware. The major challenge for technical success of MPSoC is to make sure that the software executes efficiently on the hardware [Bert 02].

The software design makes use of a programming model. The **programming model** abstracts the hardware for the software design. It is made of a set of functions (implicit and/or explicit primitives) that can be used by the software to interact with the hardware. Additionally, the programming model needs to cover the four abstraction levels required for the software refinement previously presented (system architecture, virtual architecture, transaction accurate architecture and virtual prototype).

Examples of programming models can be considered: the OpenMP [Cha 00] for the shared memory architectures and MPI [MPI], TTL [Van 04] or YAPI [Koc 00] for the message passing architectures. The Multiflex [Pau 06] supports both shared memory model and a remote procedure call based programming approach called DSOC (Distributed System Object Component) for message passing architectures. Its shared memory functionality is close to the one provided by POSIX [But 97], e.g. thread creation, mutexes, condition variables, etc. The DSOC uses a broker to spawn the remote methods which aligns with CORBA. The programming model is usually embodied in a **programming environment** [Cul 98].

This document considers heterogeneous MPSoC architectures organized as it was illustrated previously in figure 4 with the support of message passing communication model.

1.5. Software Stack for MPSoC

The software running on the MPSoC architecture is called **embedded software**. The software costs are often a large part of the total cost of an embedded system and are characterized by different performance requirements [Hen 03].

Often, the performance requirement in an embedded application is a **real-time** requirement. A real-time performance requirement is one where a segment of the application has an absolute maximum execution time that is allowed. For example, in a digital set-top box the time to process each video frame is limited, since the processor must accept and process the next frame shortly. In some applications, a more sophisticated requirement exists: the average time for a particular task is constrained as well as the numbers of instances when some maximum time is exceeded. Such approaches (sometimes called **soft real-time**) arise when it is possible to occasionally miss the time constraints on an event, as long as not too many are missed. Real-time performances tends to be highly application dependent.

Two other key characteristics exist in many embedded applications: the need to **minimize the memory** and the need to **minimize the power**. Sometimes the application is expected to fit totally in the memory of the processor on chip; other times the application needs to fit totally in a small off-chip memory. In any event, the importance of memory size translates to an emphasis on code size, since data size is dictated by the application. Large memories also mean more power [Hen 03].

1.5.1. Definition of the Software Stack

In this document, the software running on the software subsystems is called **software stack**. In heterogeneous MPSoC architectures, each software subsystem executes a software stack. The software stack is made of two components: the application tasks code and the hardware dependent software (HdS). The HdS layer is made of three components: the Operating System (OS), specific I/O communication software and the Hardware Abstraction Layer (HAL). The HdS is responsible to provide application and architecture specific services, i.e. scheduling the application tasks, communication between the different tasks, external communication with other subsystems, hardware resources management and control. The following paragraphs detail the software stack organization, including all these different components.

1.5.2. Software Stack Organization

The software stack is structured in different software layers that provide specific services.

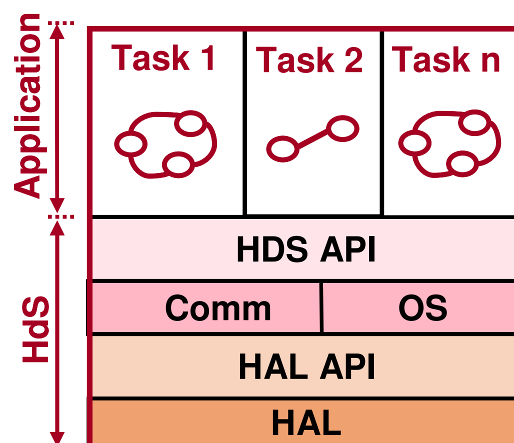


Figure 5. Software Stack Organization

Figure 5 illustrates the software stack organization in two layers: application layer and HdS (Hardware dependent Software) layer. In the first section, the application layer will be presented, and then the HdS will be defined.

1.5.2.1. Application Layer

The application layer may be a multi-tasking description or a single task function of the application targeted to be executed on the software (processor) subsystem. A **task or thread** is a lightweight process that runs sequentially and has its own program counter, register set and stack to keep track of where it is. In this document, the terms task and thread are used as interchangeable terms. Multiple tasks can be executed in parallel by a single CPU (single-core) or by multiple CPUs of the same type grouped in the software subsystem (multi-core). The tasks may share the same resources of the architecture, such as processors, I/O components and memories. On a single processor core node, the multithreading generally occurs by time slicing, wherein a single processor switches between different threads. In this case, the processing is not literally simultaneous, as the single processor is doing only one thing at a time. On a multi-core processor subsystem, threading can be achieved via multiprocessing, wherein different threads can run literally simultaneously on different processors inside the software node [Tan 95].

The application layer consists of a set of tasks that makes use of programming model or Application Programming Interface (API) to abstract the underlying HdS software layer. These APIs corresponds to the HdS APIs.

1.5.2.2. HdS Layer

The HdS layer represents the software layer which is directly in contact with, or significantly affected by, the hardware that it executes on, or can directly influence the behavior of that hardware [Pos 03]. The HdS integrates all the software that is directly depending on the underlying hardware, such as hardware drivers or boot strategy. It also provides services for resources management and sharing, such as scheduling the application tasks on top of the available processing elements, inter-task communication, external communication, and all other kinds of resources management and control. The federative HdS term underlines the fact that, in an embedded context, we are concerned with application

specific implementations of these functionalities that strongly depend on the target hardware architecture [Jer 06].

Current research studies proved that the HdS debug represents 78% of the global system total debugging time of an MPSoC design cycle [You 04]. This may due to incorrect configuration or access to the hardware architecture, e.g. a wrong configuration of the memory mapping for the interrupt control registers. In order to reduce its complexity, the HdS is structured into three software components: operating system (OS), communication management (Comm) and hardware abstraction layer (HAL).

1.5.2.2.1. Operating System

The operating system (OS) is the software component that manages the sharing of the resources of the architecture. It is responsible for the initialization and management of the application tasks and communication between them. It provides services such as tasks scheduling, context switch, synchronization and interrupt management.

Finding the optimal algorithm for the tasks scheduling represents a NP-complete problem [Ven 05]. There are different categories of scheduling algorithms. The classic criteria are hard real-time versus soft real-time or non real-time; preemptive versus cooperative; dynamic versus static, centralized versus distributed [Tan 95].

Contrary to non real-time, the real-time scheduler must guarantee the execution of a task in a certain period of time. Hard real-time must guarantee that all deadlines are met.

Preemptive scheduling allows a task to be suspended temporally by the OS, for example when a higher-priority task arrives, resuming later when no higher-priority tasks are available to run. This is associated with time-sharing between the tasks. Examples of preemptive scheduling algorithms are: round robin, shortest-remaining-time or rate-monotonic schedulers. The cooperative or non-preemptive scheduling algorithm runs each task to its completion. In this case, the OS waits for a task to surrender control. This is usually associated with event-driven operating systems. Examples of non-preemptive algorithm are the shortest-job-next or highest-response-ratio-next.

With static algorithms, the scheduling decisions (preemptive or non-preemptive) are made before execution. Contrary to static algorithms, the dynamic schedulers make their scheduling decisions during the execution.

The implementation of the scheduler may be centralized, which controls all the task execution ordering and communication transactions or distributed which distributes the control decision to local schedulers [Cho 05].

When a task is ready for execution and it is selected by the scheduler of OS according to the scheduler algorithm, the OS is also responsible to perform the context switch between the currently running task and the new task. The context switch represents the process of storing and loading the state of the CPU in order to share the available hardware resources between different tasks. The state of the current task, including registers, is saved, so that in case the scheduler gets back for execution the first task, it can restore its state and continue normally.

In order to ensure a correct runtime and communication order between the different tasks running on parallel, synchronization is required. The tasks can synchronize by using semaphores or by sending/receiving synchronization signals (events) each other. The mutex is a binary semaphore which ensures mutual exclusion on a shared resource, such as a buffer shared by two threads, by locking and unlocking it whenever the resource is accessed by a task [Tan 97] [Tan 99].

The interrupt handler is another OS service used for interrupts management. There are two types of processor interrupts: hardware and software. A hardware interrupt causes the processor to save its state of execution via a context switch, and begins the execution of an interrupt handler. Software interrupts are usually implemented as instructions in the instruction set of the processor, which cause a context switch to an interrupt handler similar to a hardware interrupt. The interrupts represent a way to avoid wasting the processor's execution time in polling loops waiting for external events. Polling means when the processor waits and monitors a device until the device is ready for an I/O operation.

1.5.2.2.2. Communication Software Component

The second software component of the HdS layer constitutes the communication component, which is responsible to manage the I/O operations and more generally the interaction with the hardware components and the other subsystems. The communication component implements the different communication primitives used inside a task to exchange data between the tasks running on the same processor or between the tasks running on different processors. It may include different communication protocols, such as fifo (first-in-first-out) implemented in software, or communication using dedicated hardware components. If the

communication requires access to the hardware resources, the communication component invokes primitives that implement this kind of low level access. These function calls are done in form of HAL APIs.

The HAL APIs allow for the OS and Communication components to access the third component of the software stack, that is the HAL layer.

1.5.2.2.3. Hardware Abstraction Layer

Low level details about how to access the resources are specified in the Hardware Abstraction Layer (HAL) [Yoo 03]. The HAL is a thin software layer which totally depends on the type of processor that will execute the software stack, but also depends on the hardware resources interacting with the processor. The HAL includes the device drivers to implement the interface for the communication with the device. This includes the implementation of drivers for the I/O operations or other peripherals. The HAL is responsible also with processor specific implementations, such as loading the main function executed by an OS, more precisely the boot code, or implementation of the *load* and *restore* CPU registers during a context switch between two tasks, but also codes for configuration and access to the hardware resources, e.g. MMU (Memory Management Unit), timer, interrupt enabling/disabling etc.

The structured representation of the software stack in several layers (application tasks, OS, communication and HAL), as previously described, has two main advantages: flexibility in terms of software components reuse by changing the OS or the communication software components, and portability to other processor subsystems by changing the HAL software layer.

1.6. The Concept of Mixed Architecture/Application Model

The following paragraphs give the definition of the mixed architecture/application model and describe the execution scheme that allows simulating this model.

1.6.1. Definition of the Mixed Architecture/Application Model

The architecture and application specifications can be combined in a mixed hardware/software model where the software tasks are mapped on the processor subsystems. This mixed hardware-software representation can be modeled by abstracting the processor

subsystems and communication topology. The processor subsystems are substituted by abstract subsystem models, while the communication is described using an abstract communication platform. The result is a mixed application/architecture model, named also mixed hardware/software model. The mixed architecture/application concept allows modeling heterogeneous MPSoC at different abstraction levels, independent from the description language used by the designer. The mixed hardware software model is called also Combined Algorithm/Architecture model [Bon 06].

1.6.2. Execution Model for Mixed Architecture/Application Model

The execution of the mixed hardware/software model is performed through a simulation which allows validation and debug of the system functionality at different stages of the design process. The execution model allows capturing the behavior of the application together with the architecture with a detailed hardware-software interaction. The execution model helps to create early performance models of the MPSoC and validate the system performances. By using different test benches, the execution model allows to test different functionality scenarios, even before the final implementation.

The execution model can be described using different simulation environments, such as SystemC or Simulink. In the following sections, the execution models described in SystemC, respectively Simulink will be presented.

1.6.2.1. Execution model described in SystemC

SystemC is a standard system level design language based on a C++ class library [OSCI]. SystemC is convenient for mixed hardware/software modeling. It provides the abstraction and constructs needed for high-level hardware modeling and verification. Such abstraction, primarily at the transaction-level, allows much faster simulations and analysis, and enables design issues to be detected early in the process. At the same time, the software can be described as C or C++ modules.

The hardware is described in SystemC using the concept of modules, ports and channels or signals provided by a C++ extension library. The software is described using the concept of concurrent threads. The threads are encapsulated into the modules and may access external channels through the ports of the modules. In this model, the hardware-software interaction is modeled using the classical concepts of channel or signals.

The execution model in SystemC allows the execution of the threads independently using their own execution stacks [OSCI]. These threads or processes can be sensitive to events on input or output ports. The sensitivity list of a process may be defined statically or can change dynamically during the simulation. There are three types of SystemC processes: SC_THREAD, SC_CTHREAD and SC_METHOD. The threads (SC_THREAD) can suspend and resume execution only when they call the *wait()* function. The clocked threads (SC_CTHREAD) are special threads which are sensitive only to clock signals. The methods (SC_METHOD) behave like a non-preemptable standard procedure written in a high level programming language. A SC_METHOD may suspend and resume execution when it gives the control to the SystemC simulation kernel.

The simulation in SystemC involves execution of the SystemC scheduler which may execute processes of the application. The SystemC simulation kernel does not preempt the execution of a thread. The SystemC processes are executed until completion or until they yield control to the simulation engine. Hence, the SystemC scheduler is co-operative multitasking, as the processes run without interruption up to the point where it either returns or calls the function *wait()*. The thread code between two *wait()* calls is executed in one simulation clock cycle. Simulation time can advance only when a *wait()* statement has been called.

The SystemC processes scheduler is event-driven, meaning that the processes are executed to the occurrence of events. Events occur or are notified at precise points in the simulation time. The scheduler can execute a process (a SystemC method or a SystemC thread) in the following cases:

- In response to the process instance having been made runnable during the initialization phase of the simulation
- In response to a call function *sc_spawn* to create processes dynamically during the simulation
- In response to the occurrence of an event to which the process instance is sensitive. Each process can have static or dynamic sensitivity list. The static sensitivity list represents the list of events that may cause the process to be resumed or triggered that are fixed before the simulation. The dynamic sensitivity list may change during the simulation.

- In response to a time-out having occurred. A time-out occurs when a given time interval has elapsed.

The simulation can start only after instantiation and proper connection of all modules and signals. The simulation starts with calling *sc_start()* from the top level that contains *sc_main()*. The *sc_start()* accepts as argument the total number of default time units of the simulation period. If the argument is a negative number, the system is simulated infinitely.

Similar to VHDL or Verilog, SystemC threads scheduler supports delta cycles. A delta-cycle is comprised of evaluate and update phases, and multiple delta cycles may occur at a particular simulated time. As illustrated in figure 6, the SystemC simulation has the following steps: initialization, evaluation, and update.

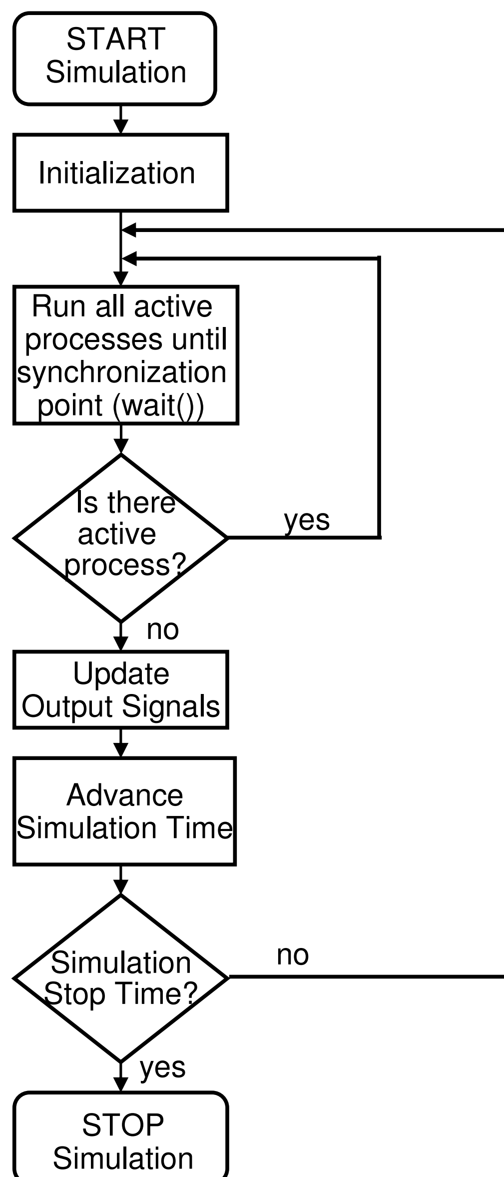


Figure 6. SystemC Simulation Steps

In the initialization step, the SystemC scheduler establishes the initial value for all signals and makes all the processes active. During the evaluation step, the scheduler executes all the processes ready to run in an unspecified order. The order of the thread execution is non-deterministic within a certain simulation phase. This may cause events notification to occur which make other processes ready. If new processes become active, the evaluation will continue until the list of ready processes is empty. The last phase is to update the values of the output signals and to advance the simulation time to the earliest pending time notification.

1.6.2.2. Execution model described in Simulink

Simulink provides the capability to model and simulate the mixed architecture/application representation like a synchronous dataflow model. The hardware is described in Simulink using the concept of subsystems, ports and signals provided by the standard Simulink library. The software is described as a set of functions using the standard Simulink blocks or user defined functions. The functions are encapsulated into the subsystems and may access external signals through the ports of the subsystems. In this model, the hardware-software interaction is modeled using the concepts of signals connecting different ports.

The execution model in Simulink supports various simulation options, such as the simulation's start and stop time and the type of solver used to solve the model at each simulation time step. Specifying simulation options is called configuring the model. Simulink enables to create multiple model configurations, called configuration sets, modify existing configuration sets or switch configuration sets.

Once the configuration model that meets the application requirements is defined or selected, the mixed architecture/application model can be executed. Simulink runs the simulation from the specified start time to the specified stop time. While the simulation is running, the system designer can interact with the simulation in various ways, stop or pause the simulation and launch simulations of other models. If an error occurs during the simulation, Simulink halts the simulation and pops up a diagnostic viewer that helps the user to determine the cause of error.

Figure 7 shows the main steps of the simulation engine in Simulink.

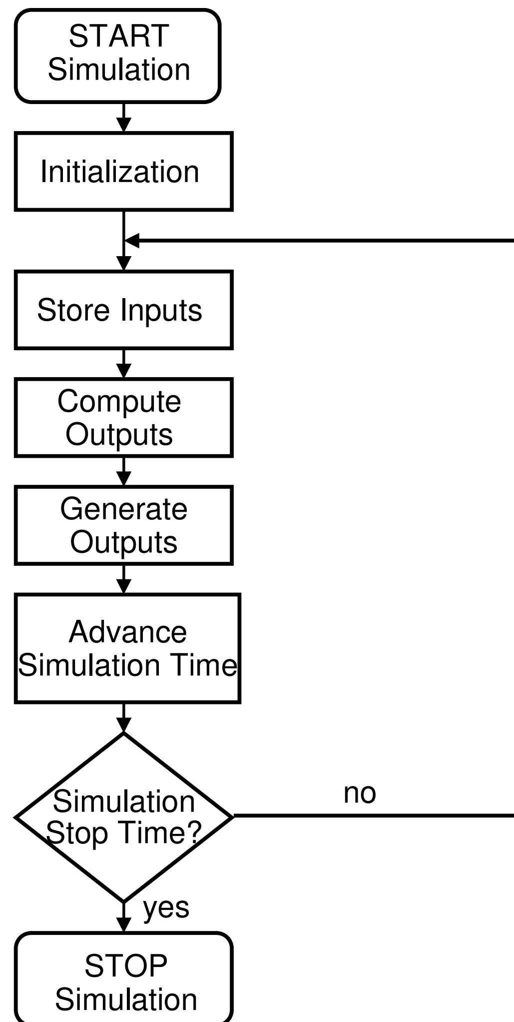


Figure 7. Simulink Simulation Steps

The first step is the initialization. This includes the compilation and link phases. First, the Simulink engine invokes the model compiler. The model compiler converts the model to an executable form. In particular, the compiler evaluates the values of the all block parameters. Then, it determines the signal attributes for the links not explicitly specified, e.g. name, data type, numeric type, and dimensionality of the signal. The model compiler checks that each block can accept the signals connected to its inputs. Simulink uses a process called attribute propagation to determine unspecified attributes. This process entails propagating the attributes of a source signal to the inputs of the blocks that it drives. Then, the model compiler performs block reduction optimizations and flattens the model hierarchy by replacing the hierarchical subsystems with the blocks that they contain. It also determines the sorted order of the blocks, which represent the invocation order of the blocks during the simulation. Finally, the model compiler determines by propagation the sample times of all blocks in the model whose sample times are not explicitly specified by the designer. After the compilation, the Simulink Engine

allocates memory needed for signals, states, and run-time parameters. It also allocates and initializes memory for data structures that store the run-time information for each block. This corresponds to the Link phase. After the memory space allocation, initial values are assigned to the states and outputs of the model to be simulated. The initialization phase occurs once at the start of the simulation loop.

After the initialization, during the simulation loop, the Simulink engine successively stores the inputs, computes and generates the outputs and states of the system at intervals from the simulation start time to the finish time. The successive time points at which the states and outputs are computed are called time steps. The length of time between steps is called step size. The step size depends on the type of the solver. The next simulation step is the sum of the current simulation time and the step size. When the simulation stop time is reached, the simulation stops.

A solver is a Simulink software component that determines the next time step that a simulation needs to take to meet target accuracy requirements that the user specified. Simulink provides an extensive set of solvers, each adept at choosing the next time step for specific types of applications. There are two types of solvers: fixed-step and variable-step. With a fixed-step solver, the step size remains constant throughout the simulation. By contrast, with a variable-step solver, the step size can vary from step to step, depending on the model's dynamics. In particular, a variable-step solver reduces the step size when a model's states are changing rapidly to maintain accuracy and increases the step size when the system's states are changing slowly in order to avoid taking unnecessary steps.

In Simulink, the simulation of the application model starts by default at 0.0 seconds and ends at 10.0 seconds. The Solver configuration panel allows specifying other start and stop times for the currently selected simulation configuration. The simulation time and the actual clock time are not the same. For example, running a simulation for duration defined in the Solver configuration pane of 10 seconds usually does not take 10 seconds. The amount of time it takes to run a simulation depends on many factors, including the model's complexity, the solver's step sizes, and the host computer's speed.

1.7. Examples of Heterogeneous MPSoC Architectures

In the following paragraphs, examples of heterogeneous MPSoC architectures are given. These examples will be used as case studies in the next chapters of the document.

The target hardware architecture considered is represented by a heterogeneous MPSoC architecture. The heterogeneous architecture contains different processor subsystems and memory or hardware subsystems. The different subsystems are interconnected via a global communication network such as bus or network on chip (NoC). A processor subsystem, which executes several tasks, includes one or more processors, local memories, peripherals, and local buses connecting them. The hardware subsystem has the similar structure with the processor subsystem. It includes one or more hardware IPs, local memories, local buses, and communication I/Os such as bus bridge or network interface. A memory subsystem includes a set of memories such as embedded SDRAMs, flash memories and external memories and it is connected to a communication network via communication I/Os.

In this document, the programming environment and the different software generation and validation steps are illustrated for 3 examples of heterogeneous MPSoC architectures, namely the 1AX, the Diopsis RDT architecture with AMBA bus and the Diopsis R2DT architecture with NoC. The main differentiation between these architectures, as it will be described in the following paragraphs, consists of: the type and number of processors incorporated in the architecture, the type of the adopted network component (bus or NoC in different topologies) and the different communication and synchronization schemes provided by the architecture.

1.7.1. 1AX with AMBA Bus

The first example of heterogeneous MPSoC represents the 1AX architecture. This architecture is illustrated in figure 8.

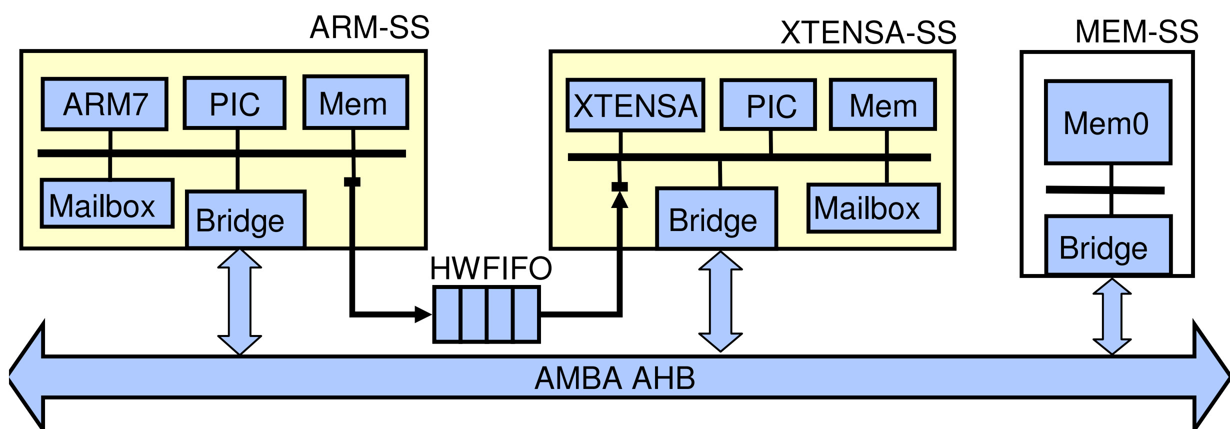


Figure 8. 1AX MPSoC Architecture

The 1AX architecture is composed of two processor subsystems (ARM-SS and XTENSA-SS) and one global memory subsystem (MEM-SS). The ARM-SS includes an ARM7 processor [Arm] used to execute the control functions of the application, while the XTENSA-SS contains a configurable Xtensa processor [Ten] for processing data-intensive algorithms. The Xtensa processor can be customized to the target application functions with an automatic instruction set generator called XPRES (Xtensa Processor Extension Synthesis). The different subsystems are interconnected using an AMBA bus [Arm]. Each processor subsystem integrates the processor core (ARM7, respectively XTENSA), the local memory (Mem), the programmable interrupt controller (PIC), mailbox for the processors synchronization, local bus and the bridge to interface with the AMBA bus.

The interrupt controller handles external interrupts according to priority to cope with external events (from mailbox) or data arriving from the other components (hardware fifo).

The local memories store program code and data. They also serve to store the buffer used for the communication between the tasks running on the same processor. The MEM-SS includes a global memory accessible by both processing units and the bridge for the connection with the AMBA bus. The 1AX architecture contains also a hardware FIFO (HWFIFO) directly connected to the local buses of the two processor subsystems. The HWFIFO contains synchronization.

Each processor and hardware subsystem have a memory address space of 4 MB (megabytes), while the memory subsystem has a memory address space equal with its memory size. The 1AX architecture contains a global memory of 256 MB. The processor subsystems have the first 4 MB address space reserved for the local bus transactions (0x00000000-0x003FFFFF). The memory address space of a processor subsystem is divided into two parts: 3 MB for local memory and 1 MB for peripheral memories. Bus transactions with addresses lower than 4 MB (0x00400000) are treated as accesses to local components, while those with addresses higher than 4MB are forwarded to the global AMBA bus via the bridge component of the processor subsystem. The bus bridge receives the forwarded transactions within the address space assigned to its subsystem.

This architecture allows two types of communication schemes between the processors: using the global memory and using the hardware FIFO. In the first communication scheme, one processor can deliver data to other processor though a global shared memory and send a synchronization event via a mailbox between different processors. For example, a data transfer from the ARM processor to the XTENSA processor using the global memory has the

following steps: first, the ARM processor checks a bit in its mailbox. If the bit is set to one, which means that a space is available in the global memory, the ARM processor clears the bit to zero, writes data to the global memory, and sets a bit of the mailbox in the Xtensa processor subsystem to one, which means that data is available in the global memory. After checking the bit in the mailbox, the Xtensa processor clears the bit of its mailbox to zero, reads the data from the global memory, and sets the bit of the mailbox in the ARM processor subsystem to notify the completion of the read operation. For this type of communication, the bandwidth of the global interconnect could become bottleneck of the inter-processor communication. It also may cause long latency to access the data because of the limitation of the shared bus.

The second possible communication scheme between the two processors is based on the hardware FIFO. The HWFIFO is a point-to-point communication between two processor subsystems. Besides the data transfer, the HWFIFO also implements the synchronization mechanism of the processors. For instance, a data transfer initiated by the ARM processor using the HWFIFO has the following steps: the ARM processor copies data from its local memory to the hardware FIFO directly. When the data number in the FIFO reaches a certain threshold, the Xtensa processor checks it through interrupt or polling methods. Then, the Xtensa processor copies the data from the hardware FIFO to its local memory. When the hardware FIFO reaches empty, the ARM processor checks it through interrupt or polling methods and copies the data again. The HWFIFO provides a new path to transfer data instead of using the shared memory and global network. Thus, it can decrease the required bandwidth of the global memory and network and speed up the communication. But compared to the global memory, the HWFIFO increases hardware area because it needs extra shared memory. It also relies on the processor to transfer data.

1.7.2. Diopsis RDT with AMBA Bus

The second target architecture example is the Shapes MPSoC architecture [Pao 06], which is a multi-tile architecture based on a Diopsis tile. Figure 9 illustrates an elementary tile, namely the RDT (RISC + DSP Tile).

The Diopsis tile is a triple core system integrating an ATMEL mAgicV VLIW DSP [Atm], an ARM 9 RISC microcontroller [Arm] and a distributed network processor (DNP). The Diopsis tile is called also D940. The system combines the flexibility of the ARM9

controller with the high performance of the DSP and the on-chip and off-chip networking capability of the DNP.

The local memories of the DSP and RISC can be accessed by both processing units. Additionally, a distributed external memory (DXM) can be used to share data between all the processors. The data transfer between these processors can follow different schemes based on an AMBA bus, e.g. the DSP can read/write data to the local memory of the ARM by using or not a DMA transfer.

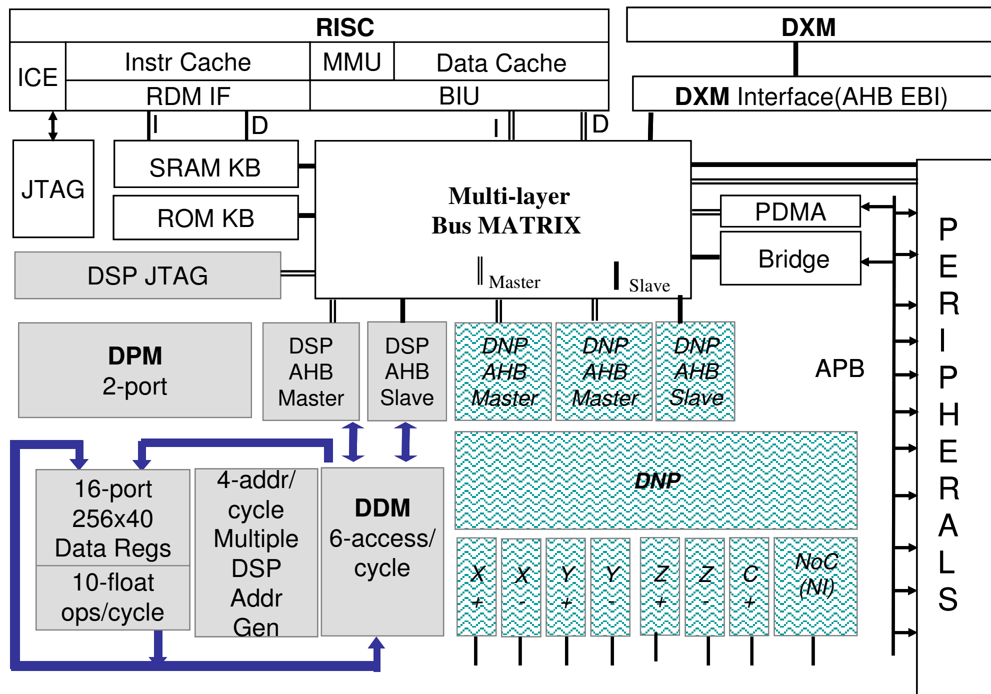


Figure 9. Diopsis RDT Heterogeneous Architecture

This document considers as example of MPSoC architecture a simplified version of the initial Diopsis tile. The selection of the components from the original architecture still captures all the possible communication schemes and specific I/O components. The subset is shown in figure 10.

The reduced Diopsis tile contains 2 software subsystems: the ARM and the DSP software subsystems. The ARM subsystem includes the processor core and local memories: SRAM for data and ROM for program code. The DSP subsystem includes the DSP core, data memory (DMEM), program memory (PMEM), control and data registers (REG), direct memory access engine (DMA), programmable interrupt controller (PIC) and the mailbox as synchronization component for the communication between the two processors. The interrupt controller handles the external interrupts according to their priorities. The timer has the

highest priority of all the interrupt sources for both processors. All the communication devices are assumed to generate an interrupt when new data becomes available.

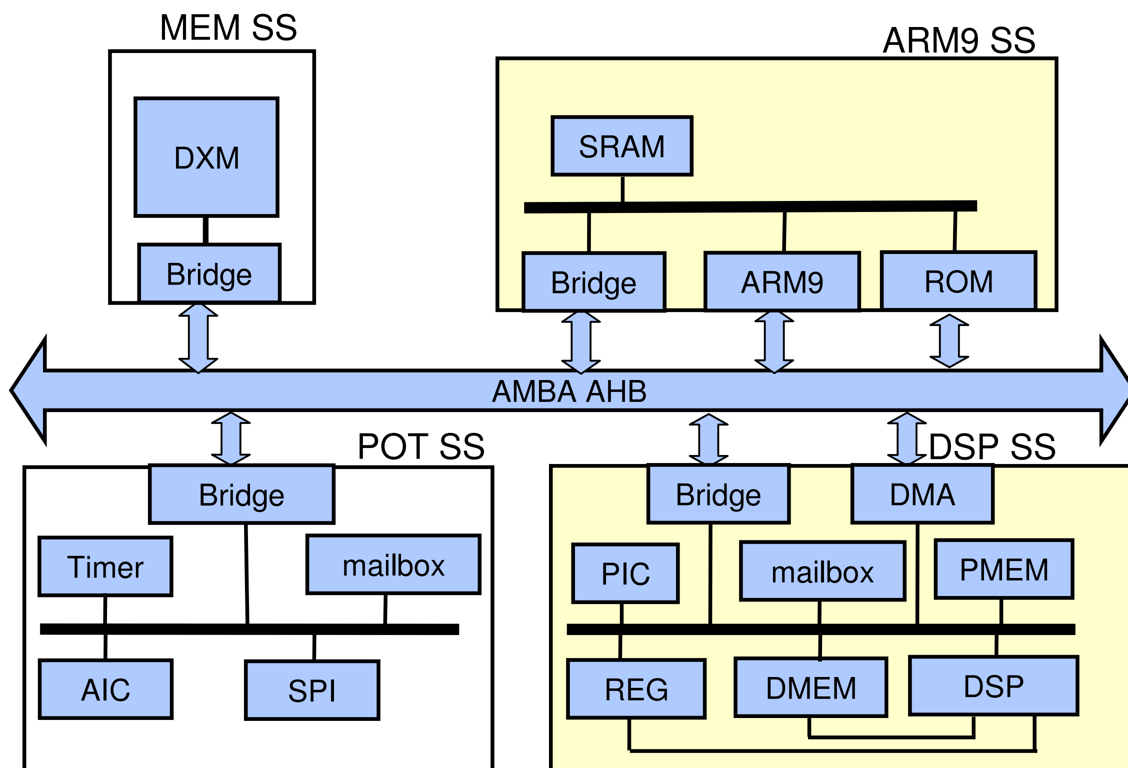


Figure 10. Target Diopsis based Architecture

Apart from the software subsystems, the architecture contains 2 hardware subsystems as well. The hardware nodes consist of distributed external memory subsystem (DXM) and peripherals on tile (POT) subsystem. The distributed external memory subsystem includes a global memory shared by the processors. The POT includes the system peripherals of the RISC processor, e.g. timer, advanced interrupt controller (AIC), but also the I/O components of the tile such as the serial peripheral interface (SPI).

The interconnection between these software and hardware subsystems is made via the AMBA bus. Hence, all the subsystems contain a bridge component to interface with the AMBA bus and a local bus for the local components interconnection. The AMBA bus supports burst mode transmissions in order to allow continuously data transfer though the bus after its initialization.

For performance reasons, the ARM processor can access directly the data memory and control/status registers of the DSP processor via the AMBA slave interface of the DSP subsystem. In the same way, the DSP core can read/write directly on the local memory of the

RISC processor by initiating a DMA transfer. Moreover, the processors can store and load data to/from DXM connected to the AMBA bus. Therefore, this architecture allows different kinds of communication mapping schemes between the processors characterized by different performances.

1.7.3. Diopsis R2DT with NoC

The third target architecture considered in this document represents the Diopsis R2DT (RISC + 2 DSP) tile. This heterogeneous architecture is an extension of the previously presented RDT tile. Figure 11 shows the Diopsis R2DT tile.

It contains three software subsystems: one ARM9 RISC processor subsystem and two ATMEL magicV VLIW DSP processing subsystems. Similarly with the RDT tile, the hardware nodes represent the global external memory (DXM) and POT (Peripherals on Tile) subsystem. The POT subsystem contains the peripherals of the ARM9 processor and the I/O peripherals of the tile. All the three processors may access the local memories of the other processors and the distributed external memory (DXM).

In this architecture, the different subsystems are interconnected using the Hermes Network on Chip (NoC) [Mor 04]. The bridges required for the data transfer through the AMBA bus of the RDT architecture are replaced with Network Interface (NI) components. In the same manner, the DMA engines of the DSP subsystems provide interfaces to the NoC instead of the AMBA AHB interface.

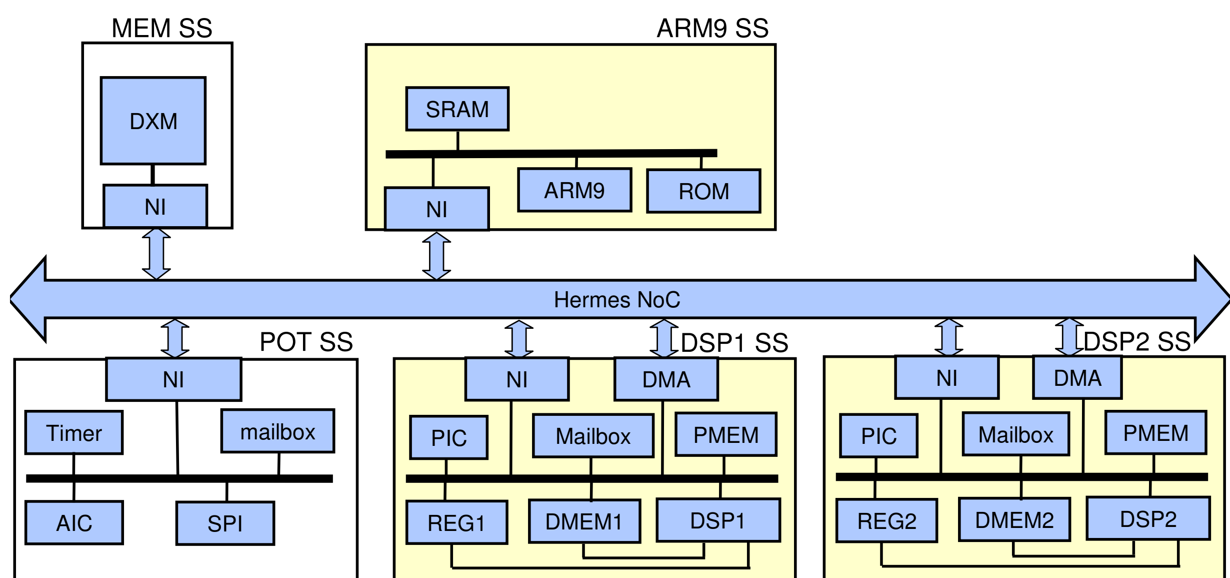


Figure 11. Diopsis R2DT with Hermes NoC

The NoC represents an on-chip packet-switched micro-network of interconnects. It is constructed from multiple point-to-point data links interconnected by switches (routers), such that data messages can be relayed from any source module to any destination module over several links, by making routing decisions at the switches. As the NoC can operate simultaneously on different data packets, it allows several data transfers in parallel through the network. Therefore, it overcomes the limitations offered by a bus in terms of bandwidth and scalability [Ben 02].

The basic components of a NoC are the switches, network interfaces and the links between them. The data delivered through the NoC from the source module to the destination module is divided into packets. A packet represents the basic unit for routing and sequencing. The packets may be divided into flits. A flit (flow control digit) is the basic unit of bandwidth and storage allocation. Flits do not have any routing or sequence information and have to follow the router for the whole packet.

There are several factors that may influence the performances of a NoC [Pul 07], such as:

- *Topology*. The topology represents the static arrangement of the routers and the channels between them. A good topology allows fulfilling the requirements of the traffic at reasonable cost. Examples of topologies are the ring, butterfly, tree, torus and mesh topologies.
- *Routing techniques*. The routing algorithm performs the selection of a path through the network. For instance, the XY routing algorithm supposes to route the flit firstly on the horizontal direction (X) and then, when it reaches the column where the destination module is located, it is routed in a vertical direction (Y). The XY routing algorithm is minimal path routing algorithm and is free of deadlock [Asc 05]. The YX routing algorithm is similar with the XY, but reverses the order of vertical and horizontal routing. Another type of routing technique is the west-first algorithm.
- *Switching strategy*. The switching strategy specifies how the flits are forwarded by the routers during the packet transmission. For instance, in the wormhole transmission scheme, the router can start forwarding the first flit of a packet without waiting for the tail [Moh 98]. Another type of switching strategy represents the small frame switching.

- *Flow control*. The flow control means how are the network resources allocated, if packets traverse the network.
- *Router architecture*. This defines the properties of the switches and the buffers of the switches, such as buffer size, buffer dimension, number of buffers, etc.
- *Traffic pattern*. The traffic pattern defines the data flows between every pair of modules connected to NoC.

The Hermes NoC supports two types of topologies: Mesh and Torus. In the Mesh topology, the NoC employs a 2D arrangement with 9 routers (3x3). The routers may have from three to five ports, depending on the router position relative to the limits of the mesh. The Mesh NoC uses a pure XY routing algorithm shared by all the ports, a round robin scheduler to arbitrate the simultaneous packet transmission requests and wormhole packet switching strategy.

In the Torus NoC model, every router has five bidirectional ports to implement a 3x3 2D Torus topology with wraparound links at the edges of the network. The routing algorithm is a deadlock free version of the well known non-minimal west-first algorithm proposed in [Gla 94].

1.8. Examples of Multimedia Applications

In the following paragraphs, three examples of applications are given. These examples represent the target applications that will run on the architecture examples previously described, considered as case studies in the remaining part of this document.

The target application domain represents the multimedia applications domain. This kind of application can be found in many areas, such as entertainment, engineering, advertisements, medicine, scientific research, spatial temporal applications (visual thinking, visual/spatial learning) etc. Multimedia applications are based on information processing, e.g. text, audio, graphics, animation, video, interactivity. Examples of multimedia applications are: MPEG 2/4, H.263/4, JPEG 2000, MJPEG or the MPEG 1 Audio (layer 3) encoder/decoder (shortly the MP3).

As presented in the following paragraphs, the considered target applications are:

- the Token Ring application, a simpler example used to illustrate the concepts and methodology and targeted to be executed on the 1AX MPSoC architecture
- the Motion JPEG Decoder for image processing that will be mapped and executed on the Diopsis RDT architecture
- the H.264 Encoder application for video processing, which will be running on the Diopsis R2DT architecture with different NoC topologies.

Additionally, the programming environment was applied for an audio processing application as well, the MP3 Decoder, running on the Diopsis RDT architecture [Pop 07].

The following paragraphs describe the considered 3 application examples (Token Ring, Motion JPEG and H.264).

1.8.1. Token Ring Functional Specification

The first target application is the Token Ring application. The application is composed of three nodes that exchange a token. The nodes are connected in form of a ring. When a node receives the token, it checks if the node is the destination of the token by comparing the node's identifier with the token's value. In this case, the node performs some computations. Otherwise, it forwards the token to the next node. The functional specification of the Token Ring application is illustrated in figure 12.

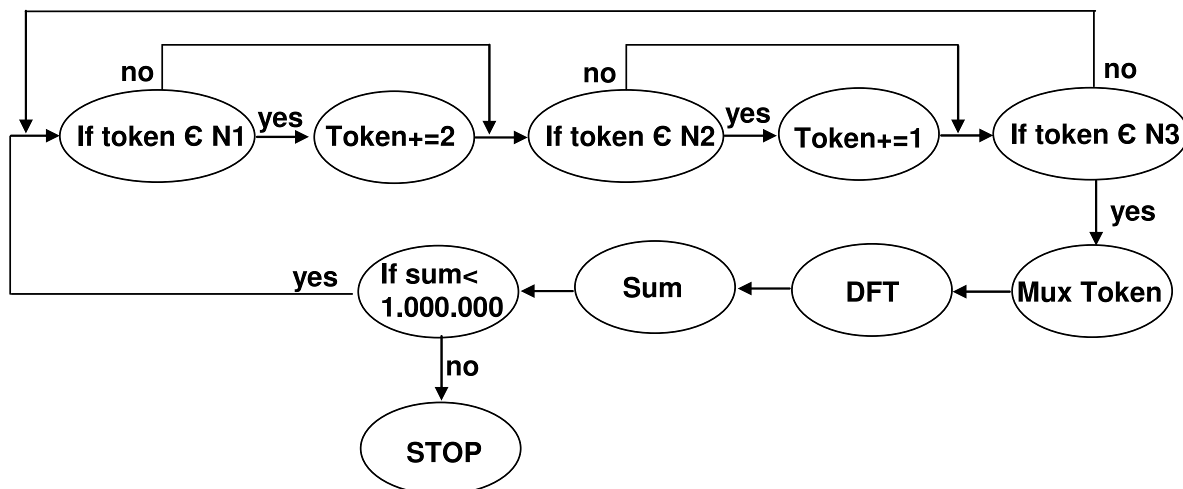


Figure 12. Token Ring Functional Specification

If the token is designed to the first node, the node increments the token value with 2 units. The second node increments the token value with 1 unit. Finally, the third token

multiplexes the value of the token and computes a DFT (Discrete Fourier Transform) function. The multiplexed value of the token represents the input for the DFT computation.

The DFT, occasionally called the finite Fourier transform, is a transform for Fourier analysis of finite-domain discrete-time signals. It expresses an input function in terms of a sum of sinusoidal components by determining the amplitude and phase of each component. However, the DFT is distinguished by the fact that its input function is *discrete* and *finite*: the input to the DFT is a finite sequence of real numbers, which makes the DFT ideal for processing information stored in computers. In particular, the DFT is widely employed in signal processing and related fields to analyze the frequencies contained in a sampled signal, to solve partial differential equations, and to perform other operations such as convolutions. The DFT can be computed efficiently in practice using a fast Fourier transform (FFT) algorithm [Coo 69].

After the DFT computation, the generated coefficients are summed and assigned as new value to the token. The iteration process will stop when the resulted sum is bigger than 1000000. Otherwise, the new value of the token is transmitted to the first node forming a loop.

1.8.2. Motion JPEG Decoder Functional Specification

The Motion JPEG Decoder application represents an image processing multimedia application. In this document, the baseline Motion-JPEG decoder is used as target application example, which represents the basic JPEG decoding process supported by all the JPEG decoders [Wal 91]. JPEG is named from its designer, the Joint Photographic Expert Group.

The JPEG decoder performs the decompression of an encoded JPEG bitstream (01011…) and renders the decoded images on a screen. The JPEG compression algorithm splits the input image on blocks of 8x8 pixels. The decoder performs the exact opposite process of the encoder. The main functions of the decoder are illustrated in figure 13.

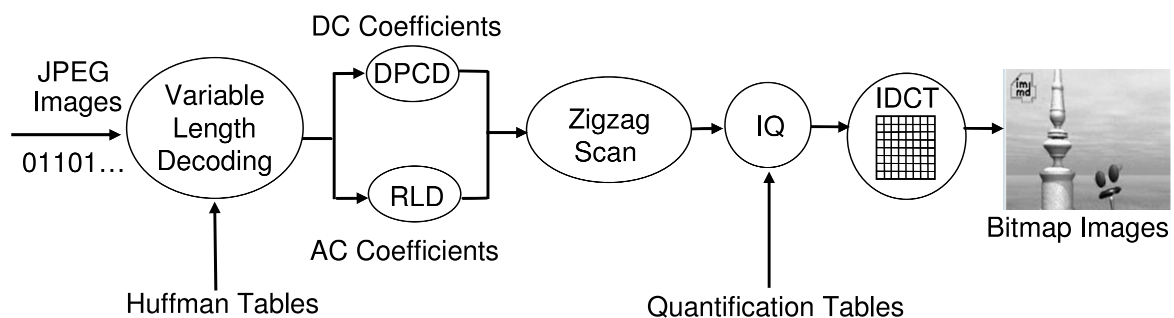


Figure 13. Motion JPEG Decoder

The main functions of the Motion JPEG decoder algorithm are described as follows:

- *VLD (Variable Length Decoder).* The VLD represents the Huffman entropy decoder. The input binary sequence of the compressed image is converted to a symbol sequence using Huffman tables. This represents the opposite step of the VLC (Variable Length Coder) step of the JPEG Encoder, when variable length codes are assigned to the symbols created from the DCT coefficients. The decoder can store only two sets of Huffman tables: one AC table and one DC table per set.
- *DPCD (Differential Pulse Code Demodulation).* The DPCD represents the opposite process of the DPCM (Differential Pulse Code Modulation) part of the JPEG Encoder, which is applied on the DC coefficient. The DC coefficient represents the coefficient with 0 frequencies in both dimensions of the DCT coefficients matrix, located at the left-top corner [0, 0]. The DPCD is in charge with the reconstruction of the DC coefficient.
- *RLD (Run Length Decoding).* The RLD is the opposite step of the RLC (Run Length Coding) of the compression algorithm, which is applied on the AC coefficients (the 63 DCT coefficients which are different from the DC coefficient). The AC coefficients are treated separately from the DC coefficient. The RLD supposes to reconstruct the sequence of the AC coefficients from the sequence of symbols by inserting the zero-valued AC coefficients before the non-zero valued AC coefficients in the coefficients sequence.
- *Zigzag Scan.* This step puts back the 64 DCT coefficients in form of a matrix with 8x8 dimensions. The input of this step is an array of 64 elements in zigzag order, and its output is an 8x8 matrix in original order.
- *IQ (Inverse Quantization).* The IQ is applied upon the 64 DCT coefficients using quantification tables. This step consists of the multiplication of each of the 64 coefficients by its corresponding quantizer step size. The quantizer steps are stored in the quantification tables.
- *IDCT (Inverse Discrete Cosine Transform).* This step transforms the 64 DCT coefficients (the 8x8 block) from frequency domain to spatial domain and reconstructs the 64-point output image signal by summing the basis signals.

1.8.3. H.264 Encoder Functional Specification

The H.264 Encoder application represents the third application example used as case study. This application is a video processing multimedia application. It represents a standard for video compression also known as MPEG-4 Part 10 or AVC (Advanced Video Coding). The H.264 supports coding and decoding of 4:2:0 YUV video formats.

The input image frame (F_n) of a video sequence is processed in units of a macroblock, each consisting of 16 pixels. A pixel consists of three color components: R (red), G (green) and B (blue). Usually, pixel data is converted from RGB to YUV color space, where Y represents the luma, and U and V the chroma samples. A macroblock contains 16x16=256 Y luma samples and 8x8=64 U and 8x8=64 V chroma components. Each of these components is processed separately. There are three types of macroblocks: I, P and B. The macroblocks are numbered in raster scan order within a frame. A set of macroblocks is called slice.

To encode a macroblock, there are three main steps: prediction, transformation with quantization, and entropy encoding. These main functions of the standard H.264/AVC (Advanced Video Coding) are illustrated in figure 14 [Ric].

The prediction step tries to find a reference macroblock which is similar with the current macroblock to be encoded. Depending on where the reference macroblock comes from, there are two types of prediction: Intra and Inter mode. In Intra mode or I-mode, the macroblocks are predicted using the previously encoded, reconstructed and unfiltered macroblocks. In this case, the reference macroblock is usually calculated with mathematical functions of neighboring pixels of the current macroblock. In the Inter (P or B) mode the macroblocks are predicted from a reference picture (F'_{n-1}) by motion estimation (ME) and motion compensation (MC). This involves finding a 16x16 sample region in the reference frame that closely matches the current macroblock. The reference picture maybe chosen from a selection of past or future (display order) pictures, that have been already encoded, reconstructed and filtered. The ME involves finding a 16x16 sample region in the reference frame that closely matches the current macroblock. A popular matching criteria is to measure the sum of absolute difference (SAD) between the current block and the candidate block and to find its minimal value [ChenJ 06].

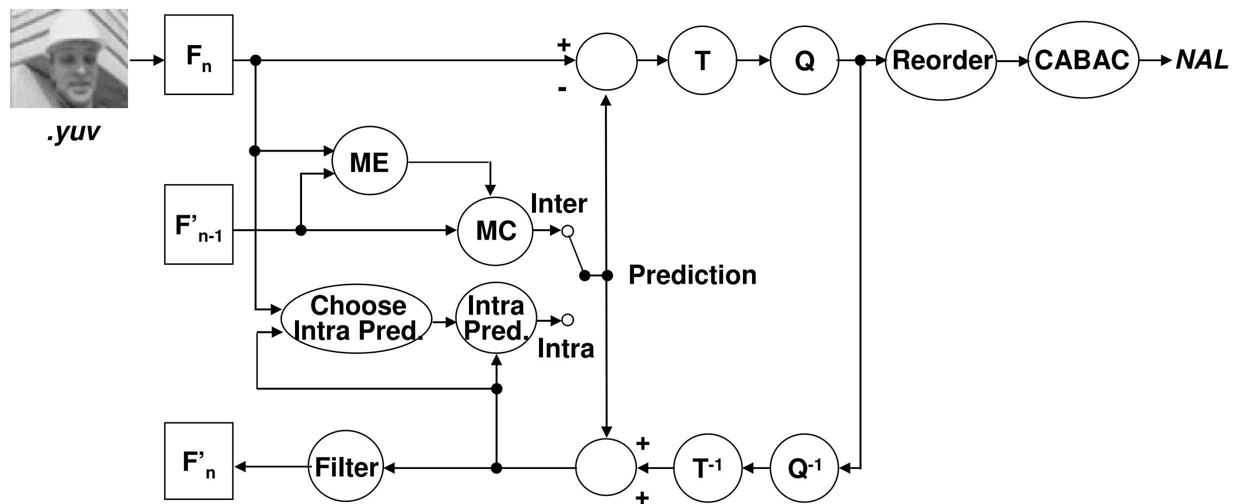


Figure 14. H.264 Encoder Algorithm Main Profile

After the prediction, the resulted macroblock is subtracted from the initial block to produce a residual (difference) block. Then, the residual block is transformed (T) and quantized (Q) to give a set of quantized transform coefficients, which are reordered (Reorder) and entropy encoded (CABAC in figure 14). The H.264 uses two types of transforms (T): Hadamard and DCT transforms, depending on the residual data that is to be encoded. A Quantizer step is used for the division of the DCT coefficients. The Quantizer steps size could be different for the luma and chroma components, but in range 0-51. During the Reordering, each 4x4 block of quantized transform coefficients are mapped to a 16-element array in a zigzag order, which will be entropy encoded. There are two types of entropy encoder: CAVLC (Context Adaptive Variable Length Coder) and CABAC (Context Adaptive Binary Arithmetic Coding).

The entropy encoded coefficients together with the information required to decode each macroblock (prediction modes, quantizer parameters, motion vector information, etc) form the compressed bitstream. This compressed bitstream is passed to a Network Abstraction Layer (NAL) for transmission or storage of the encoded image.

During the encoding process, the H.264 algorithm decodes (reconstructs) the macroblock to provide a reference for further predictions. The quantized transform coefficients are scaled (Q^{-1}) and inverse transformed (T^{-1}) to produce a difference block. This difference block is added to the predicted macroblock to create the reconstructed block.

Then, a filter is applied to reduce the effects of blocking distortion. The filter smoothes block edges, improving the appearance of the decoded frame. After that, the reconstructed reference picture is created from a series of blocks.

The H.264 standard supports seven sets of capabilities, which are referred as profiles, targeting specific class of applications. The most used profiles for MPSoC implementation are the Baseline, Main Profile and Extended Profile. In this document, the Main Profile will be used as application case study. The Main Profile includes support for interlaced video, which means that not the entire image is compressed, but only every second line, i.e. the odd lines of the 1st image and even lines of the 2nd image. The Main Profile also supports entropy coding using context-based adaptive binary arithmetic coding (CABAC).

1.9. Conclusions

This chapter defined the basic concepts used in programming heterogeneous MPSoC architectures, such as MPSoC, software stack, hardware-software interface and execution model.

The software stack was organized into several components (application tasks code, OS, communication and HAL). This layered organization of the software stack allows a gradual design performed in several steps corresponding to different abstraction levels (system architecture, virtual architecture, transaction accurate architecture and virtual prototype). The software validation is performed by simulation using an abstract architecture model.

The specification of three multimedia applications (Token Ring, Motion JPEG Decoder and H.264 Encoder) and three examples of MPSoC architectures (1AX, Diopsis RDT, Diopsis R2DT), which will execute these applications, were given.

In the following chapters, the software design and validation at each of these different abstraction levels will be detailed.

Chapter 2

SYSTEM ARCHITECTURE DESIGN

This chapter presents the system architecture design. The system architecture design consists of partitioning and mapping the application onto the target architecture, and mapping the communication onto the available hardware resources. The key contribution in this chapter represents the definition, organization and design of the system architecture using Simulink, for the Token Ring application running on the 1AX architecture, the Motion JPEG application targeting the Diopsis RDT architecture and the H.264 Encoder running on the Diopsis R2DT architecture. The functional simulation of the system architecture models allows validation of the applications' algorithm.

2.1. Introduction

The system architecture design consists of partitioning the application into several parallel tasks and mapping the application tasks onto the target architecture. The result of the system architecture design represents the system architecture model. In this chapter, the mapping process will be defined, and then the system architecture model will be presented.

The objectives of the system architecture design are:

- Functional validation of the target application algorithm
- Specification of the application partitioning and mapping onto the hardware architecture.

2.1.1. Mapping Application on Architecture

2.1.1.1 The Mapping

MPSoC design flow starts with two separate models: architecture and application [Lie 01] similar with the Y-chart [Kie 02]. Usually, the description of the application functionality and hardware topology are independent of one other. The architecture is specified as a set of processor and hardware subsystems that interact via communication network.

The application is generally specified as a functional model made of a set of multiple functions. Then, the functions are grouped into tasks in order to identify the parts of the application which can be done in parallel. This corresponds to the partitioning step. A parallel software is composed of multiple cooperating tasks, each of which performs a subset of functions of the application. In case that the initial model of the application is sequential, e.g. sequential C code, a parallelization step is required.

The parallelization process determines how the computation, data access or input/output operations and data can be distributed among different processing elements [Cul 98]. It also determines which parts of the application will be implemented in software and which parts in hardware. The parallelization of the application consists in dividing the computation in several pieces that can be executed in parallel. These different pieces group several functions of the application and are named tasks or processes. The parallelization mechanism is called partitioning. A parallel application decreases the total execution time of

the application compared to its sequential execution [Pau 06]. The partitioning step is a quite difficult to be optimized in a general case [Ver 07] and it will not be considered in this document.

The partitioned model of the application will be mapped on the target architecture. The different tasks running in parallel may be executed by different processors. The number of tasks does not have to be the same with the number of processors available in architecture. If the number of processors available on the target architecture is less than the number of tasks, more than 1 task may be executed on the same processor. The assignment of tasks to a target processor that will execute them is called mapping.

The mapping represents the association between the tasks and the processing elements on which they are executing and the association between the buffers used for the communication between the tasks and the hardware communication resources of the architecture [Fla 07]. The mapping should ensure a balanced distribution of the computation over the processors in order to meet the design constraints, e.g. the overhead of the communication, synchronization and parallelism management [Cul 98].

The output of the mapping represents the assignment of the application functions to the architectural units [Mic 02]. This corresponds to the system architecture model.

Example 1. Mapping Token Ring Application on 1AX

This paragraph illustrates the partitioning, mapping and binding process for the Token Ring application onto the 1AX architecture. Figure 15 shows the correlation between the application functions and the architecture elements.

In this example, the functions of the application are grouped into three tasks. Each task corresponds to a token node. The first two tasks (T1 and T2) are mapped on the ARM7 processor, while the third task T3 is mapped onto the XTENSA processor.

Figure 15 shows also the allocation of the communication buffers onto the hardware communication resources. Thus, the communication buffer used for the data exchange between the two tasks mapped on the ARM7 processor T1 and T2 is mapped onto the local memory SRAM of the ARM sub-system. The communication buffers used between the task T1 and T3, respectively task T2 and T3, which correspond to the communication between the two processors, are mapped on the global memory.

```
      The   result   of   the   mapping   process   is   the   system
architecture model of the Token Ring application mapped on the
1AX architecture.
```

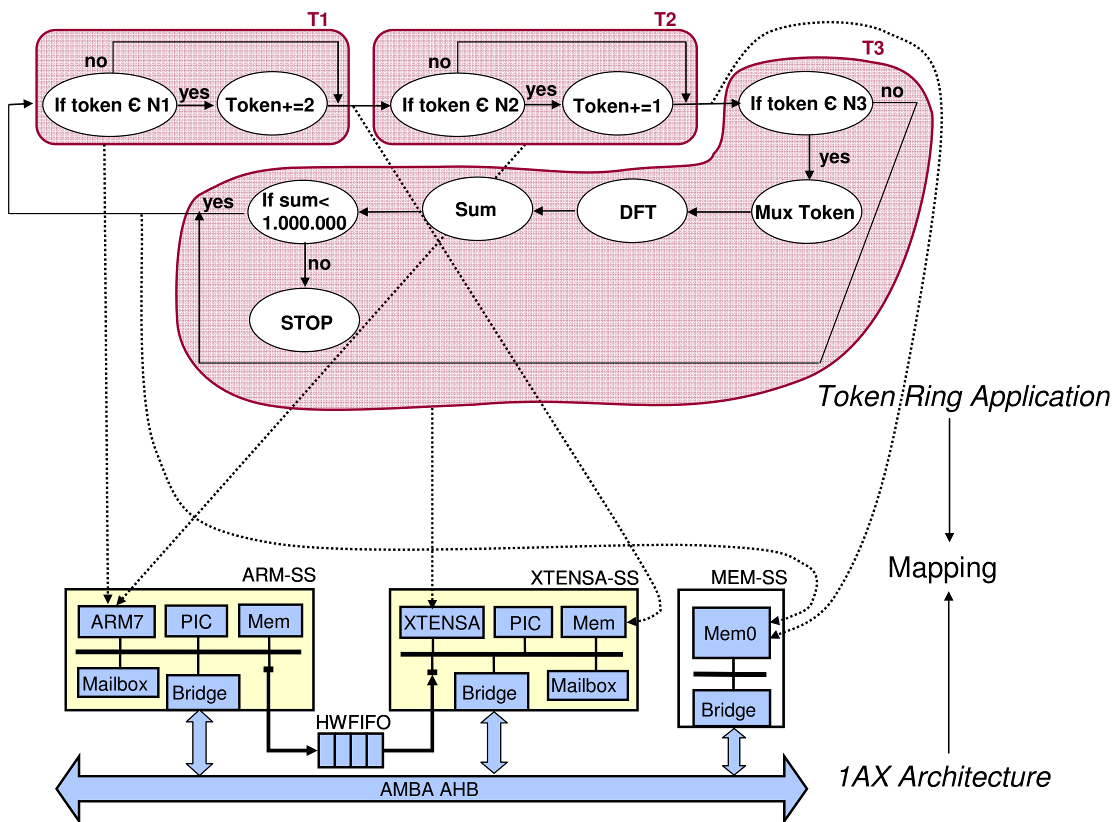


Figure 15. Mapping Token Ring on the 1AX architecture

2.1.1.2 The Design Space Exploration

Generally, there are many ways to map an application onto a given hardware architecture. The design space exploration represents the different combinations of mapping parallel software to parallel hardware. In order to be affordable in terms of design cost, the mapping should not require any change of the application code but only of the hardware dependent code. It also represents the different ways of interaction between the hardware, software and their configuration and extension.

Usually, the mapping of an application to an MPSoC platform starts from a complex system specification and goes through a vast design space exploration. The application software needs to be adapted to the parallel capabilities of the multiprocessor architectures. Furthermore, to enable fast and flexible exploration of the possible application-to-architecture

mappings, it is necessary to automate the hardware-software partitioning of the application [Bel 06].

There are two ways for design space exploration: spatial and temporal. The spatial exploration refers to binding application to architecture. It defines the possibilities of mapping tasks on processors and the communication channels between the tasks to communication paths available in the MPSoC architecture. The temporal exploration refers to computation and communication ordering. It defines the scheduling policy on each resource and the according parameters, e.g. time-division multiple access scheme and the associated slot length, fixed priority scheduling and the associated priorities, or static scheduling and the associated ordering [Thi 07].

The goal of design space exploration is to find a best matching between application and architecture based on well defined criteria or objectives, including the design constraints. The design space exploration is generally represented as an iterative loop with two main phases: performance evaluation and optimization in terms of cost and performances (figure 16). The evaluation of the performances for MPSoC may be done using simulation or analysis-based methods [Chak 03].

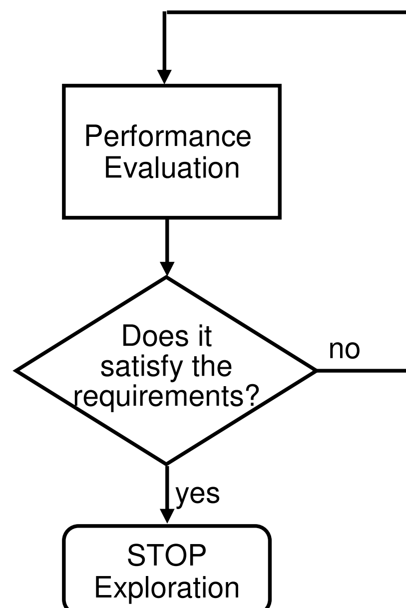


Figure 16. Design Space Exploration

2.1.2. Definition of the System Architecture

The output of the mapping process represents a model at the highest abstraction level, called system architecture model (SA) (figure 17). The definition of the system architecture

given by the Carnegie Mellon University's Software Engineering Institute in its glossary is: "*representation of a system in which there is a mapping of functionality onto hardware and software components, a mapping of the software architecture onto the hardware architecture, and human interaction with these components*" [Car].

The system architecture represents a high level application model combined with partitioning and mapping information. Aspects related to the architecture model (e.g. processing units available in the target hardware platform) are combined into the application model (i.e. multiple tasks executed on the processing units), resulting a combined architecture/application model. Thus, the system architecture model expresses parallelism in the target application through capturing the mapping of the application functions into tasks and the tasks into subsystems. It makes also explicit the communication units between the tasks to abstract the implementation of the communication protocol used for the data exchange between them.

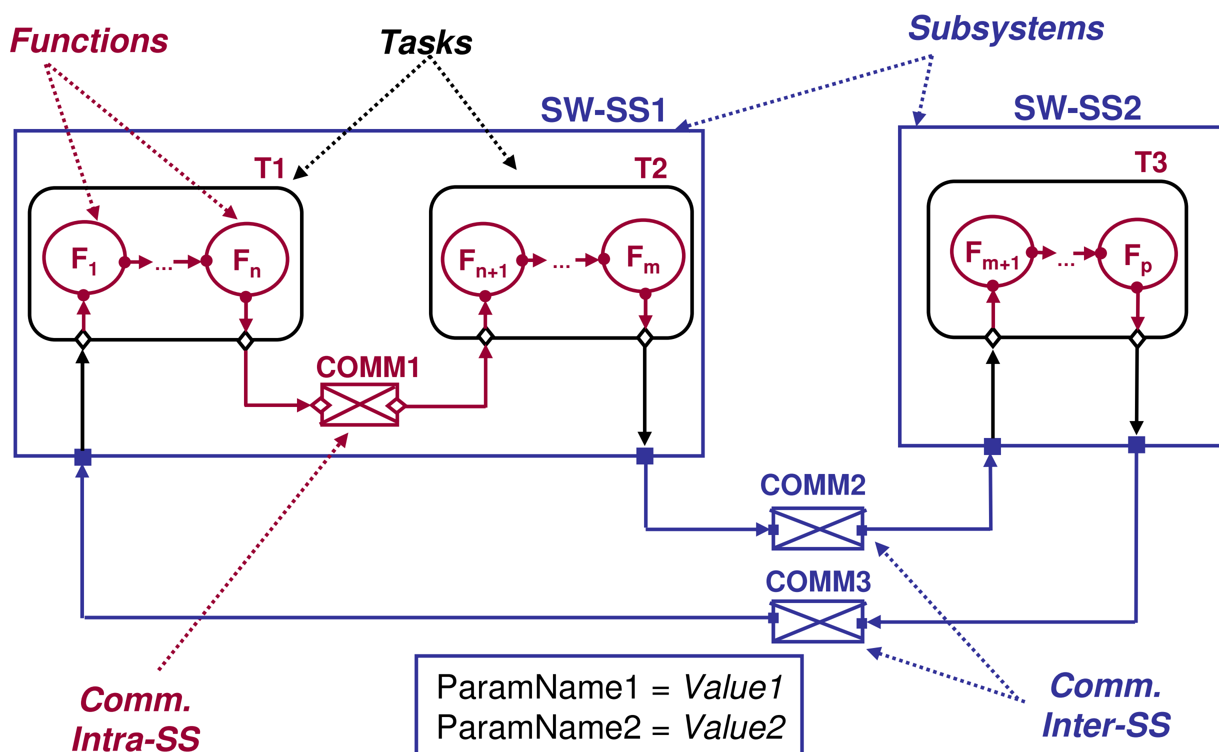


Figure 17. Global View of the System Architecture

At the system architecture level, the software is made of a set of functions grouped into tasks. The function is an abstract view of the behavior of an aspect of the application. Several tasks may be mapped on the same software subsystem. The communication between functions, tasks and subsystems make use of abstract communication links, e.g. standard

Simulink signals or explicit communication units that correspond to specific communication paths of the target platform. The corresponding hardware platform consists of the set of the abstract subsystems.

2.1.3. Global Organization of the System Architecture

The system architecture model is a hierarchical model composed of several components layers. This approach provides insight into how a model is organized and how its parts interact.

The system architecture model may be represented using the hierarchy of concepts depicted in figure 17. Thus, figure 17 shows a conceptual representation of the system architecture defining the following concepts: subsystems, tasks, functions, inter-subsystem communication and intra-subsystem communication.

A subsystem represents a set of tasks that are aimed to be mapped on the same subsystem. Examples of subsystems are SW-SS1 and SW-SS2 in figure 17. This corresponds to the mapping process of the application tasks on the different computation resources of the target architecture. A task groups a set of application functions. Examples of tasks are T1, T2 and T3. This corresponds to the result of the partitioning process of the application functions into tasks.

The basic element of the system architecture model represents the function. This can be an elementary application function either pre-defined or user defined function. Example of predefined functions are the functions representing mathematical operations (+, -, /, *), constants, conditional structures (if-else) or repetitive structures (do-while). The user defined functions represent specific functions implemented in diverse programming languages (e.g. C, C++, Matlab). The user defined functions are also part of the system architecture model.

In order to specify the communication protocol used for the data exchange between the different tasks, communication units are inserted between them in the system architecture model. Later in the software design flow, during the next design steps, the communication units will be replaced by behaviorally equivalent channel implementations with the annotated protocol and device drivers from a real-time operating system targeting to run on the processor.

There are 2 types of communication units: Inter-subsystem and Intra-subsystem. The Inter-subsystem communication shows the communication between different subsystems, e.g.

the communication units COMM2 and COMM3 between the subsystems SW-SS1 and SW-SS2. The Intra-subsystem communication specifies the communication between the tasks mapped on the same subsystem, e.g. COMM1 communication unit between tasks T1 and T2 mapped on SW-SS1. The number of the communication units depends on the application partitioning and mapping on the target hardware architecture. The communication between the functions inside the same task is implicit in the system architecture model and it will be translated to communication via local variables inside the task during the next design step.

The system architecture model is annotated with software and hardware architecture parameters to allow the generation and validation of the software stack and hardware simulation platforms, and design space exploration.

The system architecture model may be represented using different environments such as Simulink or SystemC. In the following paragraphs, the system architecture will be detailed using as case study the Simulink environment.

Example 2. System Architecture Model of the Token Ring Application mapped on the 1AX architecture in Simulink

Figure 18 illustrates a screenshot of the system architecture modeled in Simulink for the Token Ring application mapped on the 1AX architecture. The top layer of the model's hierarchy represent the two software subsystems available in the 1AX architecture (ARM7-SS and XTENSA-SS) and the 2 inter-subsystem communication units between them (COMM1 and COMM2). The 2 inter-subsystem communication units allow the data exchange between the 2 processor subsystems.

The application functions of the Token Ring application are grouped into three tasks (T1, T2 and T3). Figure 18 illustrates that for the Token Ring application 2 tasks (T1, T2) are mapped on the ARM7 processor and the third task (T3) is mapped on the XTENSA processors.

The third task (T3) running on the XTENSA processor computes the DFT function. The DFT function is implemented in an application library developed in C programming language, representing an example of user defined function for the Token Ring application.

The data exchange between the tasks T1 and T2 mapped on the ARM7 processor requires the communication unit COMM.

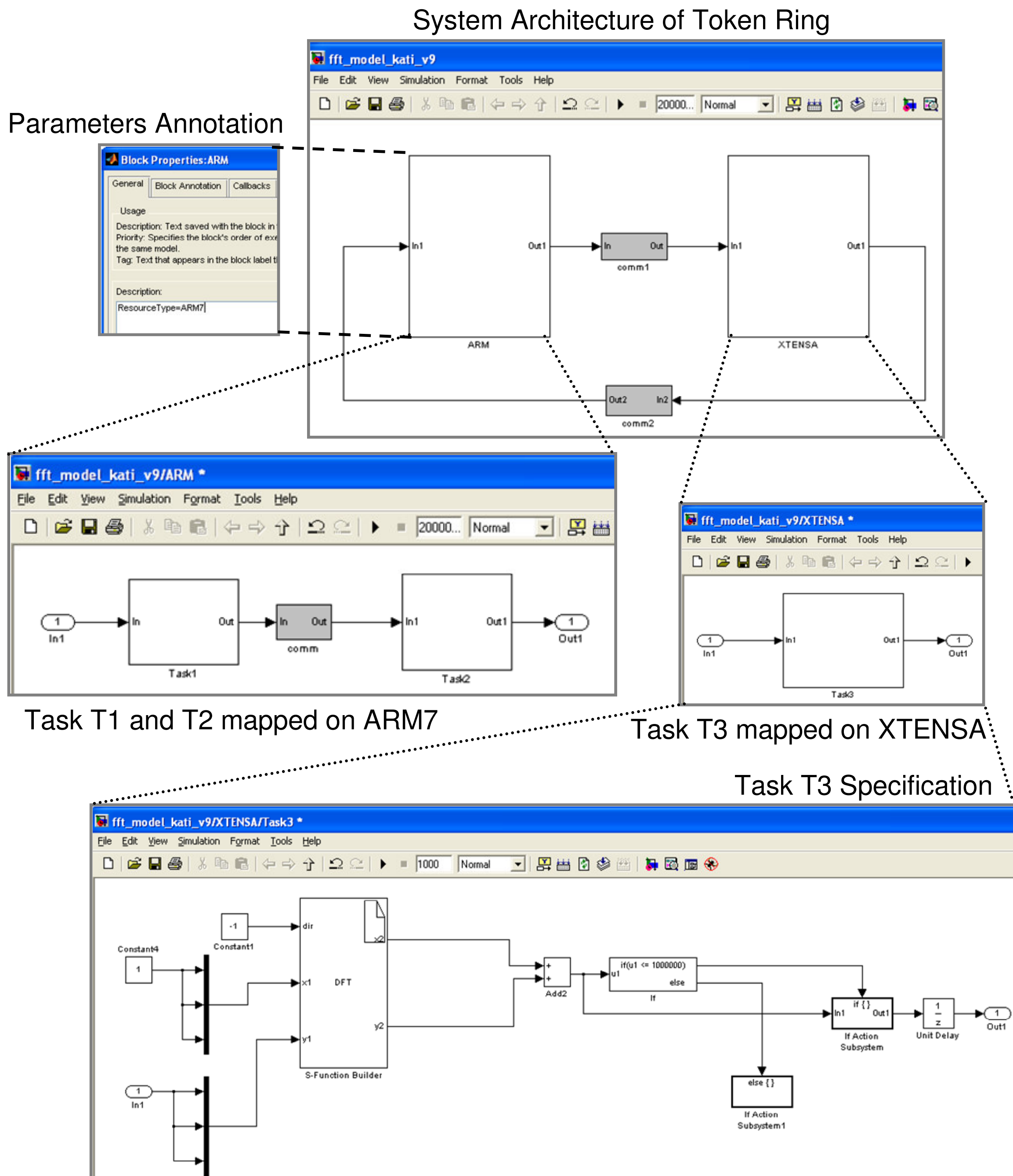


Figure 18. System Architecture Model of Token Ring

2.2. Basic Components of the System Architecture Model

The basic components of the system architecture Model are the computation and communication components. The computation components consist of the application

functions, while the communication makes use of generic I/Os, such as Simulink I/Os or SystemC signals. The detailed description of these components will be illustrated in the following paragraphs using the Simulink environment as representation medium of the system architecture design.

2.2.1. Functions

The application functions can be modeled in Simulink by using Simulink blocks. There are two types of blocks: standard Simulink blocks and user defined blocks. Blocks are the elements from which Simulink models are built. Every Simulink block has a set of attributes, called parameters, which govern its appearance and its behavior during the simulation. Some types of parameters are common to all blocks (e.g. Block Name), while other attributes are specific to a particular type of block. Simulink allows users to specify values for many of a block’s parameters, thus enabling to customize the behavior to fit the requirements of a particular application. For the standard blocks, Simulink provides predefined continuous and discrete function blocks and a graphical user interface (GUI) to relieve the application model building.

For the user defined blocks, Simulink provides the capability to integrate in the model user defined blocks developed in other programming languages such as C, C++ or Matlab, by using S-functions. To integrate S-functions in Simulink, there are two methods. The first one is to write the S-function block manually. But this method requires also a manual development of a wrapper function, which calls the actual function code. Additionally, the S-function has to be manually compiled using the *mex* utility, in order to generate the MEX file accepted by the Simulink simulation engine.

The second method to use an S-function consists of an automatic generation and compilation of the S-function by using the S-Function Builder tool integrated in the Simulink environment. The resulted S-function has a fixed type signature. But the designer has only to set up the configuration panel by specifying the source files of the hand-written function code, the format of the function call and the input/output arguments passed to the subroutine. The input arguments represent the constant parameters required for the subroutine execution. The output arguments represent the return value of the subroutine or the non constant parameters whose values can be changed by the function. Then, based on the configuration, the S-Function Builder will automatically create and compile the corresponding S-function.

2.2.2. Communication

The communication between the different application functions is made using the Simulink signals. These links or signals connect the different Simulink blocks. They may carry data from one block to one or more other blocks. The data transmission from the source block and its arrival at the destination block happens simultaneously in a rendezvous fashion. The signals may carry different types of data, such as integer, floating point or boolean. The dimension of the data may vary also from scalar to vector or matrices, but it has to be constant during the execution of the model.

2.3. Modeling System Architecture in Simulink

The system architecture may be described in Simulink or SystemC. This chapter will present the modeling style in case of using the Simulink environment.

2.3.1. Writing Style, Design Rules and Constraints in Simulink

The system architecture design in Simulink imposes some limitations and constraints. These design rules include:

- Constraints on the selection and configuration of the blocks used for the application modeling;
- Constraints on the integration of application functions implemented in other programming languages such as C/C++;
- Constraints regarding the construction of the system architecture model.

2.3.1.1. Constraints on the Simulink standard blocks

The system architecture model may use only discrete Simulink blocks in order to allow a discrete event simulation.

In case of algebraic loops or feedback path, unit delay blocks have to inserted in the Simulink loop. These unit delay blocks have to be configured by a sample rate equal with 1 in order to delay their input signals inside the loop. The sample rate represents the number of samples par second. The other blocks are characterized by an inherited sample-rate. In case of

inherited sample-rate blocks, Simulink assigns an inherited sample rate to a block based on the sample rates of the blocks connected to its inputs.

The supported predefined blocks are restricted to a subset of the standard Simulink library. This subset includes:

- Mathematical operations, such as: sum, multiplication, division, modulo, absolute value, etc;
- Logic operations: AND, OR, XOR, unary minus, shift arithmetic;
- Discrete blocks: delay, mux, demux, merge, selector, etc;
- Conditional structures: if-then-else and switch-case;
- Repetitive structures: for-loop and while-condition-loop;
- Sources: constants, extern files, input ports;
- Sinks: display block, scope block, extern files and output ports.

2.3.1.2. Constraints on the S-Functions

The system architecture model may include user defined functions written only in C programming language. The S-Functions used to integrate the customized code need to be built by using the S-Function Builder tool, which is the fastest and easiest way for the S-Function generation compared with the manual implementation.

The arguments of the user defined function have to respect a well-defined order. Basically, the function call accepts as parameters the input arguments followed by the output arguments, as illustrated in figure 19. The order of the parameters definition has to be identical with the order of the input/output ports declaration in the configuration panel of the S-Function Builder tool. Moreover, the user defined C function has to return a *void* value.

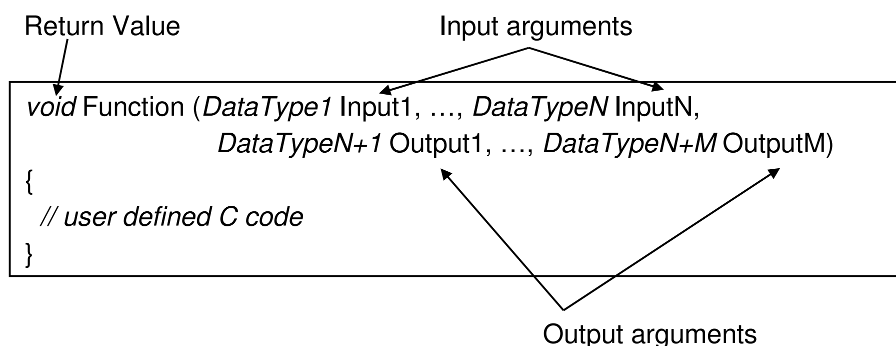


Figure 19. User-defined C-Function

Example 3. S-Function for the FFT computation in Token Ring

For example, the user defined C function used for the DFT computation in the Token Ring application is declared and implemented as shown in figure 20, where *dir*, *x1* and *y1* represent the input arguments and *x2* and *y2* represent the output arguments.

```
void DFT(int* dir, double* x1, double* y1, double* x2, double* y2)
{
 int i,k;
 double arg;
 double cosarg,sinarg;
 double x22[3], y22[3];

 for (i=0; i<3; i++)
    {
       x22[i]=0;
       y22[i]=0;
       arg=-(*dir) * 2 *3.141592654 * ((double)i)/3;
       for (k=0; k<3; k++)
          {
            cosarg = cos(k*arg);
            sinarg = sin(k*arg);
            x22[i] = x22[i]+(x1[k]*cosarg - y1[k]*sinarg);
            y22[i] = y22[i]+(x1[k]*sinarg + y1[k]*cosarg);
          }
       *x2+= x22[i];
       *y2+= y22[i]/2;
    }
}
```

Figure 20. DFT Function of the Token Ring

The different S-Functions are not allowed to share global variables between them, their access being limited to the local data variables. The entire data passing between the different S-Functions has to be modeled explicitly, with a dedicated Simulink signal used for the connection between the different S-Functions.

2.3.1.3. Constraints on the communication

To transfer data in Simulink, the different blocks are connected by signals. The signal between the blocks may carry data from one source block to multiple destination blocks. This specific feature determines to use communication units which correspond to point-point

communication schemes and restricts the global shared memory accesses in the target architecture.

2.3.2. Software at System Architecture Level

The software at the system architecture level consists of a set of application functions grouped into tasks.

Example 4. Software in the Token Ring System Architecture Model

```
    The software in the Token Ring system architecture model is
represented by the application functions, e.g. DFT, +, -, /, *,
if, else or mathematical constants.
```

2.3.3. Hardware at System Architecture Level

The hardware at the system architecture level consists of the set of abstract hardware and software subsystems that encapsulate the tasks aimed to be executed on those subsystems and the different communication units introduced between the subsystems to specify the communication protocol.

Example 5. Hardware in the Token Ring System Architecture Model

```
    The hardware in the Token Ring system architecture model is
represented by the processor subsystems XTENSA-SS and ARM7-SS and
the inter-subsystem communication units COMM1 and COMM2 that
connect the 2 processor subsystems.
```

2.3.4. Hardware-Software Interface at System Architecture Level

The hardware-software interface at the system architecture level consists of a set of links which connect the input/output ports of the different subsystems with the input/output ports of the tasks that are mapped on those subsystems.

Example 6. Hardware-Software Interface in the Token Ring System Architecture Model

```
Example of hardware-software interface in the Token Ring
system architecture model represents the Simulink link that
connects the output port of task T3 (Out) with the output port of
the XTENSA subsystem (out) in figure 18.
```

2.4. Execution Model of the System Architecture

The Simulink model is used as a reference model for debugging the application's algorithm. The following sections will describe the adopted configuration in the Simulink simulation engine to validate the system architecture model.

The simulation of the system architecture model uses a variable step discrete solver. This allows validation of the functionality of the application. For the Inter-subsystem and Intra-subsystem communication units, Simulink uses an abstract simulation model for each of these units based on generic Simulink I/Os.

The system architecture model is used to validate the application's algorithm through functional simulation. It is similar to native code execution on the host machine. The performance estimation at this level uses a simulation-based approach. As the system architecture model represents a high level application model, the hardware architecture is completely abstracted, including processor subsystems or communication infrastructure. The memory usage is also abstracted, the application using variables and pointers without taking in consideration details related to aspects such as shared memory or virtual memory. As presented in [Bac06], a relevant metric at this high abstraction level is the simulation time of the application.

The simulation time may give useful information on the efficiency of the application's algorithm in terms of behavioral features. The application's algorithm does not depend on the final operating system that will be running on the target processors, but influences the performance after the application's parallelization.

Example 7. Simulation of the Token Ring System Architecture Model

```
    The simulation of the system architecture model in Simulink
allows to validate the functionality of the application,
including the DFT computation.  The simulation requires 3 seconds
and it stopped when the resulted value after the different
computations had become bigger than 1000000, conform the initial
specification of the application.
```

2.5. Design Space Exploration of System Architecture

2.5.1. Goal of Performance Evaluation

The MPSoC design process relies on several decisions and constraints related to hardware and software architecture, which can influence the overall performance of the system. Examples of hardware architecture decisions are: number and type of processors, memory size, type of memories (local, global), type of communication network (point-to-point, bus, network on chip), communication latency, etc. Examples of software architecture decisions are: scheduling algorithm used by the operating system for the tasks activation/deactivation, type of communication primitives (blocking or non blocking semantic), real time execution requirements, binary code size, synchronization mechanisms between the tasks running on the same processor, etc.

These different decisions influence the overall execution time of the system, cost and power consumption. Therefore, good decisions are required to be able to control de MPSoC design process.

The goal of performance evaluation at the system architecture level is to allow in an early phase of the design process profiling the communication and computation. This can be accomplished by providing information independently from the system behavior in time.

2.5.2. Architecture/Application Parameters

The system architecture model is annotated with application and architecture parameters that can influence the global performance of the final system. These parameters

can be classified in 2 main categories: specific to subsystems or specific to communication units.

a) Architecture/ Application Parameters specific to Subsystems

The parameters specific to subsystems characterize the different subsystems (hardware and software) from both hardware and software points of view.

Examples of hardware architecture parameters that annotate the subsystems are:

- *ResourceType* which specifies the type of the hardware resource. There are 3 types of hardware resources: computation resource (processors), storage resource (memory) and I/O resources (I/O peripherals). In the case of a subsystem that represents a processor subystem, the *ResourceType* specifies the type of the processor cores.
- *NetworkType*. This parameter specifies the type of the network component used to interconnect the different subsystems in the target architecture. Examples of interconnect component are bus and Network-On-Chip (NoC).
- *NoCTopology*. This parameter is used when the *NetworkType* is NoC and it specifies the topology of the NoC. Examples of NoC topologies are mesh, torus, tree or butterfly.
- *NoCRoutingAlgorithm*. This parameter is used when the different subsystems are interconnected by a NoC. The parameter specifies the routing algorithm used by the routers to transmit the received data packet. Example of a routing algorithm is the XY or YX.
- *NoCArbitrationAlgorithm*. This parameter is used to specify the type of the arbitration algorithm used inside a NoC router (e.g. round robin, priority based, etc), in order to select the routing request to be treated, when the router receives more than one request simultaneously for packets transmission.
- *ResourceName* which identifies the hardware resource.

Each subsystem which represents a processor subsystem is annotated with software architecture parameters. Examples of software architecture parameters are:

- *OSType*, which specifies the name of the operating system running on the target processor. Examples of operating systems are: Linux, Mutek, DwarfOS and eCos.

- *SchedulerType* to identify the type of the OS scheduler (preemptive, cooperative), in case that the target OS supports different schedulers.
- *SchedulerAlgorithm* to define the algorithm used for the tasks management by the operating system (round-robin, priority based) etc.

Example 8. Parameters specific to the Subsystems in the Token Ring System Architecture Model

Examples of architecture/application parameters annotating the subsystems of the Token Ring system architecture model are: *ResourceType* with values "ARM7" for the ARM7 subsystem and "XTENSA" for the XTENSA subsystem, *NetworkType* with the value "AMBA" because in the 1AX architecture the different subsystem are interconnected through the AMBA bus, *OSType* with value "DwarfOS" to specific that the target operating system running on each software system is the DwarfOS for both processors.

---

b) Architecture/ Application Parameters specific to Communication Units

The parameters specific to the communication units can be architecture or application parameters.

Examples of hardware architecture parameters that annotate the communication units are:

- *ResourceType* which specifies the type of the communication protocol or storage resource. In the case of an Inter-subsystem communication unit, the *ResourceType* specifies the storage resource on which the communication buffer will be mapped. The communication buffer can be mapped in the sender subsystem, receiver subsystem or in a stand-alone storage resource, such as global memory or hardwarw FIFO. In the case of an Intra-subsystem communication unit, the *ResourceType* specifies the communication protocol implemented in software, such as software FIFO protocol or shared memory.
- *AccessType*. This parameter identifies whether the access to the memory that stores the communication buffer is performed directly by the processor or by using

a DMA mechanism, in case that the target hardware architecture provides such kind of mechanism for the memory access.

- *ResourceName* which identifies the storage resource in case that the target hardware architecture provides several storage resources of the same type, e.g. several hardware FIFOs or several global memories.

The communication unit can also be annotated with software architecture parameters, e.g. *CommType,* which identifies the type of the communication library used during the HdS integration. This parameter specifies the communication APIs used in the tasks code after their generation for the data exchange, e.g. *send(…)/recv(…)* APIs when the *CommType* is "MPI"; or *DOL_read(…)/DOL_write(…)* communication APIs when the *CommType* parameter has the value equal with "DOL". The different communication units can be accessed using different communication primitives in the tasks code.

Example 9. Parameters specific to the Communication Units in the Token Ring System Architecture Model

Examples of architecture/application parameters annotating the communication units of the Token Ring system architecture model are: *ResourceType* with value equal with "GMEM" in case of the communication units COMM1 and COMM2 in order to specify that the corresponding communication buffers used for the data exchange between the 2 processors of the 1AX architecture are mapped on the global memory. The parameter *ResourceType* has the value "SWFIFO" for the communication unit COMM in order to specify that the communication between the tasks T1 and T2 mapped on the same ARM7 processor follows a software FIFO protocol.

Another parameter annotating the communication units COMM1 and COMM2 represents *ResourceName* with values "MEM0" in order to specify that the communication buffers corresponding to these communication units are mapped on the global memory identified through id *MEM0*. The *CommType* parameter has the value "MPI" in case of all the communication units, in order to specify that all the tasks use the communication primitives "*send(...)/recv(…)*" for all the data exchanges.

2.5.3. Performance Measurements

At the system architecture level, the performance measurement consists of profiling the communication and computation for each task and/or for each processor. As the system architecture has no time notion, the result of profiling is a time independent data. Examples of metrics that can be measured at this level are: application data size, buffer size required for the intra-subsystem and inter-subsystem communication, the total quantity of exchanged data between the tasks during the execution, the number of iterations of a function execution, the amount of data transferred between the different processors, etc.

Example 10. Performance Measurements in the Token Ring System Architecture Model

For example, the DFT function of the Token Ring application was required to be called and computed totally 14 times during the execution of the whole application. The application data exchanged between the ARM and XTENSA processors was 112 bytes during the entire execution. The size of the application data sent by the first task mapped on the ARM processor to the second task running on the ARM processor was 64 bytes.

2.5.4. Design Space Exploration

At the system architecture level, the designer can experiment different partitioning and mapping schemes. The designer can regroup the functions into the tasks in several ways, and map these tasks on different subsystems. This exploration influences the total amount of data exchanged between the tasks during the execution, the application data size or the number of iterations of a specific function. By changing the partitioning and mapping of the application on the target architecture, the number and type (intra-subsystem, inter-subsystem) of communication units may vary also. The designer may adopt different communication protocols and may map the communication buffers onto different storage resources by annotating the communication units with the proper architecture parameters.

Example 11. Design Space Exploration for the Token Ring Application

In the case of the Token Ring application, the designer may map the buffers required for the inter-subsystem communication onto different communication architecture resources, such as the local memories of both ARM and XTENSA processors or the shared global memory. The designer may also opt for the dedicated hardware FIFO component directly connected to the local buses of both processor subsystems. Regarding the partitioning and mapping, the Token Ring functions can be grouped forming tasks in different ways, resulting tasks with different levels of granularity. These tasks may be mapped on the processors in several ways. For example, the DFT computation may be mapped on the ARM processor instead of the XTENSA processor, letting the XTENSA processor to be responsible for the control part of the Token Ring application. In this case, the number of the intra-subsystem communication units becomes 1 for the XTENSA processor, while the ARM7 subsystem has no intra-subsystem communication unit.

2.6. Application Examples at the System Architecture Level

In the following paragraphs, two examples will be presented at the system architecture level: the Motion JPEG application mapped on the Diopsis RDT architecture and the H.264 encoder application mapped on the Diopsis R2DT architecture.

2.6.1. Motion JPEG Application on Diopsis RDT

This section presents the system architecture design in case of the Motion JPEG (MJPEG) Decoder application running on the Diopsis RDT architecture with AMBA bus. This consists of mapping the Motion JPEG Decoder application onto the Diopsis RDT platform and modeling the resulted system architecture.

The first step of the system architecture design represents the functional modeling of the application in Simulink. The development of the MJPEG Decoder application in Simulink requires 7 S-Functions in order to integrate the C code of the main parts of the decoding algorithm.

After the functional modeling, the main functions of the application are isolated into separate tasks. Figure 21 shows the application partitioning into tasks. The variable length decoding (VLD) constitutes the first task. The differential pulse code demodulation on the DC component (DPCD), run length decoding on the AC component (RLD), zigzag scan and inverse quantization (IQ) are grouped into a second task. The inverse discrete cosine transformation (IDCT) makes up the third task, and, finally, the display function of the decoded image composes the fourth task.

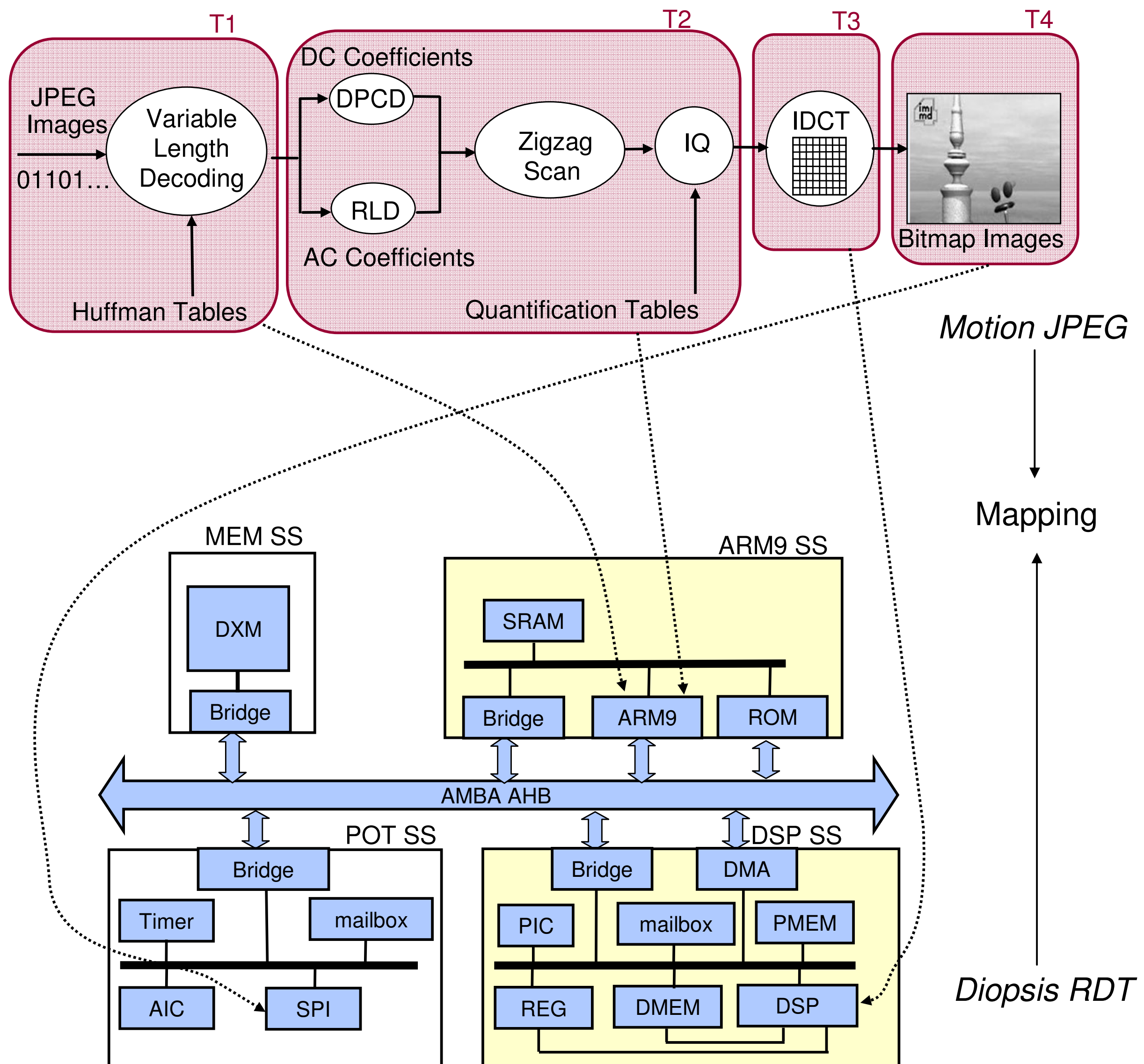


Figure 21. Mapping Motion JPEG on Diopsis RDT

After the partitioning of the application functions into tasks, the next step represents the mapping of these tasks onto the computation and I/O subsystems provided by the target architecture. Thus, the four different tasks are mapped onto the Diopsis RDT architecture

(figure 21). Thus, the first 2 tasks (*T1* and *T2*) are mapped on the ARM9 processor. The third task (*T3*), performing the IDCT computation, takes 68% of the total execution time of the decoding process, being the most computation intensive task. Therefore, it was mapped on the DSP processor. The resulting decoded image of the task *T4* is displayed on a LCD panel connected through the SPI peripheral of the POT. Hence, *T4* was mapped on the POT sub-system.

After the mapping process, in order to specify the communication protocol, several communication units are inserted between the tasks mapped on the same processor and between the different subsystems. The communication buffers used for the communication between the tasks can be mapped on the local memories of both processor subsystems, or on the global memory. Besides the buffer mapping on the storage resources, the mapping has to specify the end-to-end communication path between the two processors. The result of the mapping represents the system architecture model for the Motion JPEG application mapped on the Diopsis RDT architecture. A screenshot of the system architecture modeled in Simulink is illustrated in figure 22.

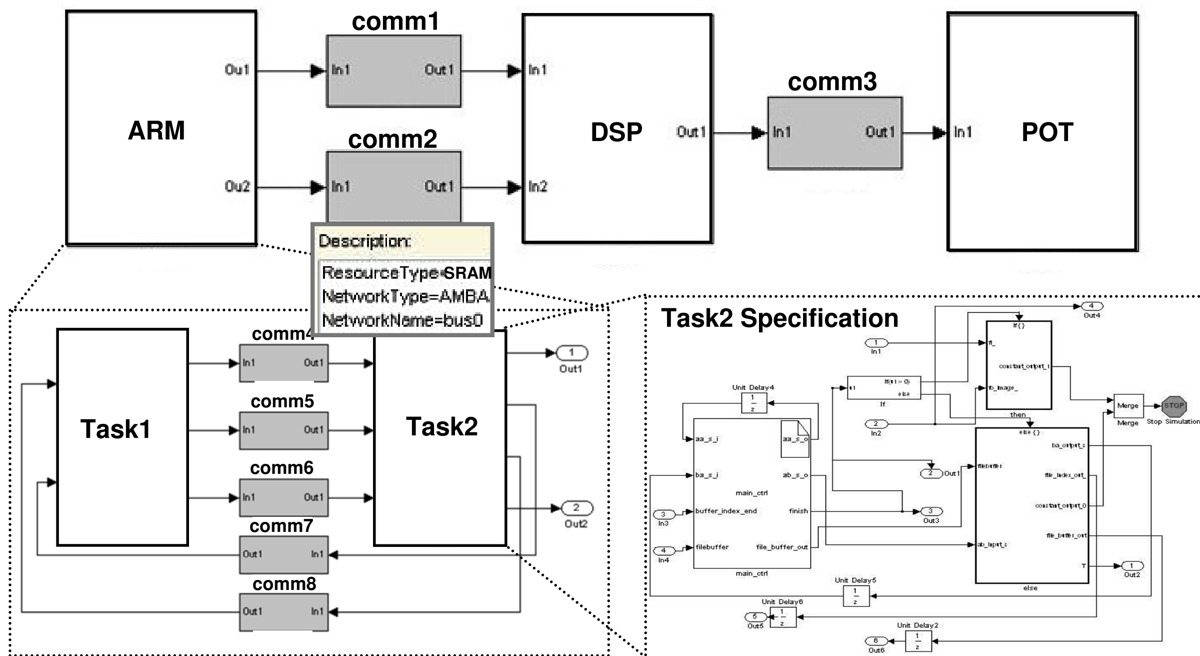


Figure 22. System Architecture Example: MJPEG Mapped on Diopsis

This model includes 3 subsystems: 2 software subsystems (the ARM and the DSP subsystem) and a hardware subsystem (the POT). The Simulink hierarchy is able to capture the mapping of the application onto the architecture at a high abstraction level through the decomposition of the system into tasks and subsystems. The Simulink model includes also

explicit communication units to capture different communication protocols and resources provided by the architecture.

In this case, the Intra-subsystem communication for the tasks mapped on the same subsystem follows a software FIFO protocol (SWFIFO). The Inter-subsystem communication between the different subsystems uses data buffers mapped on different storage resources, such as DSP data memory (DMEM), DSP registers (REG), ARM local memory (SRAM) or the distributed external memory (DXM). Later in the design flow, each of the communication units can be mapped on a specific communication path and protocol of the final architecture. The number of communication units depends on the application partitioning and mapping decisions on the target architecture.

The system architecture in Simulink is annotated with architecture information used for the further software refinement and generation of the hardware simulation platform. The hardware architecture parameters used in this example are:

a) Parameters specific to subsystems:

- *ResourceType* with possible values "ARM9", "DSP" and "POT" in case of a subsystem.
- *NetworkType* parameter, which has the value equal with "AMBA" for the Diopsis architecture with AMBA bus.
- *ResourceName* parameter. This parameter identifies the hardware resource of the target architecture, e.g. the DSP.

On the software side, each processor will execute a tiny operating system, namely *Dwarfos*. Thus the *OSType* parameter for each software subsystem of the system architecture model has the value "Dwarfos". As the *Dwarfos* operating system supports only round robin preemptive scheduling, the software architecture parameters *SchedulerType* and *SchedulerAlgorithm* are not required in this example.

b) Parameters specific to communication units:

- *ResourceType* with possible values or "DXM", "SRAM", "DMEM", "REG" for the inter-subsystem communication and "SWFIFO" in case of a intra-subsystem communication unit.

- *AccessType* parameter, required to specify for an inter-subsystem communication unit whether the DSP will access the local memory of the ARM or the external global memory directly or by initiating a DMA transfer.
- *ResourceName* parameter. This parameter identifies the hardware resource of the target architecture in case of an inter-subsystem communication unit, e.g. the external memory DXM.

The tasks executed by the processors will use the *send(…)/recv(…)* primitives as communication APIs. Therefore, the *CommType* has the value "MPI" to represent the message passing *send(…)/recv(…)* semantics for each communication unit.

To validate the MJPEG algorithm, the system architecture model can be simulated using the Simulink discrete-time simulation engine. The input test image represents a 10 frames bitstream encoded using QVGA YUV 444 format. The simulation time is 15s on a PC running at 1.73GHz with 1GBytes RAM.

The simulation allowed measuring some performance indicators. Hence, the total number of iterations necessary to decode the 10 frames input image was 36000. The communication between the ARM and DSP through the communication unit COMM1 requires a buffer of 1 word (4 bytes), the communication unit COMM2 requires 64 words (256 bytes) buffer size and finally, the communication unit COMM3 requires 4 words (64 bytes). The total number of words exchanged between the different subsystems during the decoding process of the 10 frames was 2484 KWords.

2.6.2. H.264 Application on Diopsis R2DT

The H.264 Encoder is a computation intensive video application and more complex than the Token Ring or Motion JPEG applications. Hence, the Diopsis R2DT with NoC is used to execute the H.264 encoder.

This section presents the system architecture design in the case of the H.264 Main Profile Encoder application running on the Diopsis platform with 2 DSP and one ARM9 processors interconnected through a NoC.

The first step represents the functional modeling of the H.264 application in Simulink, The H.264 Encoder algorithm is composed of several functions and 2 data flows: a forward path and a reconstruction path. The input frame of a video image sequence (Fn) is processed

in units of a macroblock. The reference code used for the H.264 Encoder development is the x264 open source code [X264]. The sequential C code is converted to a dataflow model as explained in [Hwang 06]. The development of the H.264 Encoder application in Simulink requires 4 S-Functions in order to integrate the C code of the main parts of the encoding algorithm.

After the functional modeling, the different functions are grouped into tasks. Figure 23 illustrates the partitioning of the H.264 functions into tasks. The application functions are grouped into three tasks as follows: the CABAC entropy encoder constitutes the task *T2*; the NAL construction and bitrate controller are grouped into task *T3*; and the other computation and control functions are grouped into task *T1*.

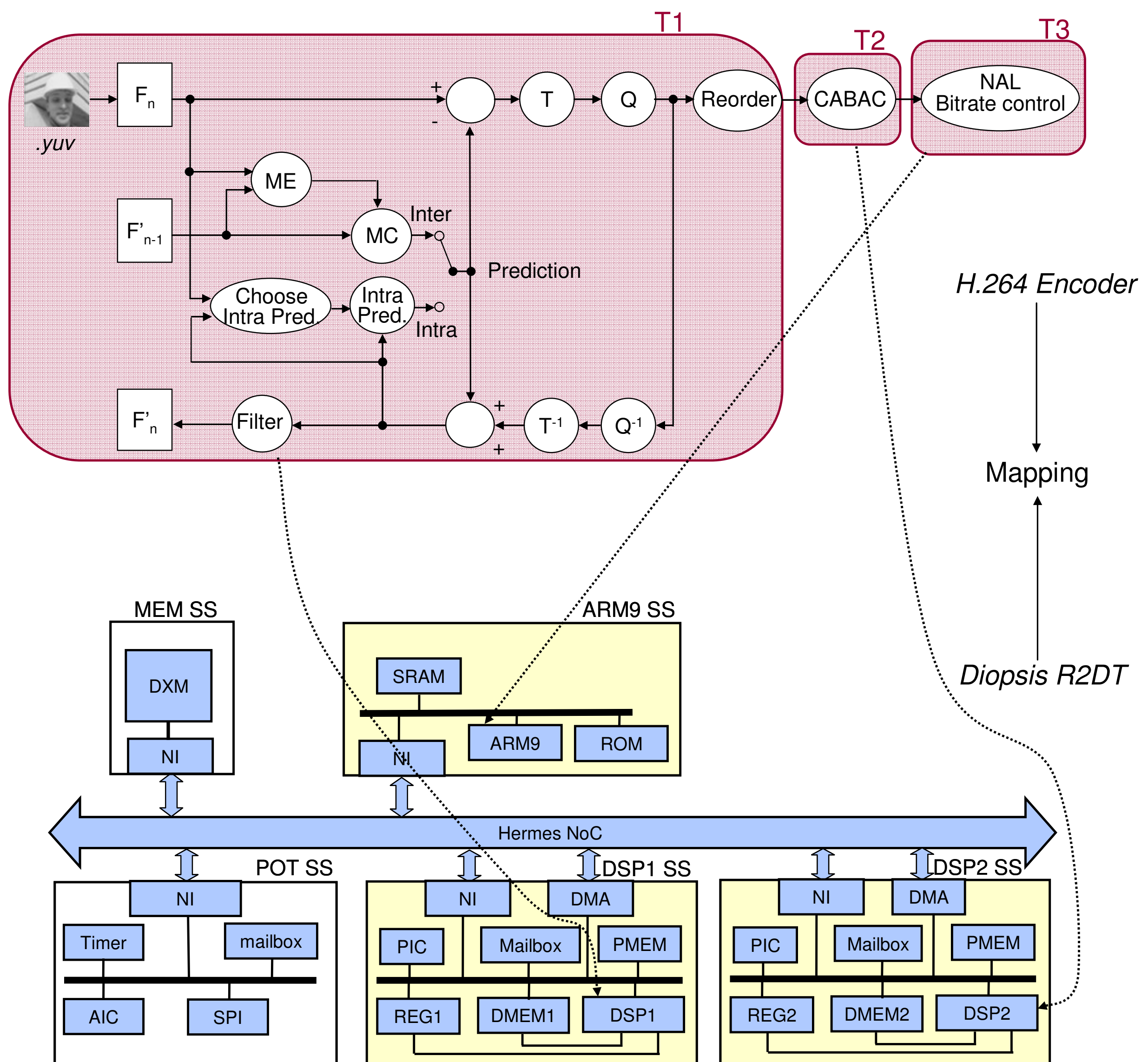


Figure 23. Mapping H.264 on Diopsis R2DT

After the partitioning, the tasks are mapped on the available resources of the Diopsis R2DT architecture. Therefore, each task is mapped on a different CPU. As task T1 and T2 requires a big amount of computation, they are mapped on the DSP processors: task T1 on DSP1, respectively task T2 on DSP2. The task T3 including the control part for the bitrate is mapped on the ARM9 processor. The communication between these tasks mapped on the three different processors represents Inter-subsystem communication and it implies a total of 3 communication units, one between each pair of processors (DSP1->DSP2, DSP2->ARM9, and ARM9->DSP1).

The resulted system architecture model is illustrated in figure 24.

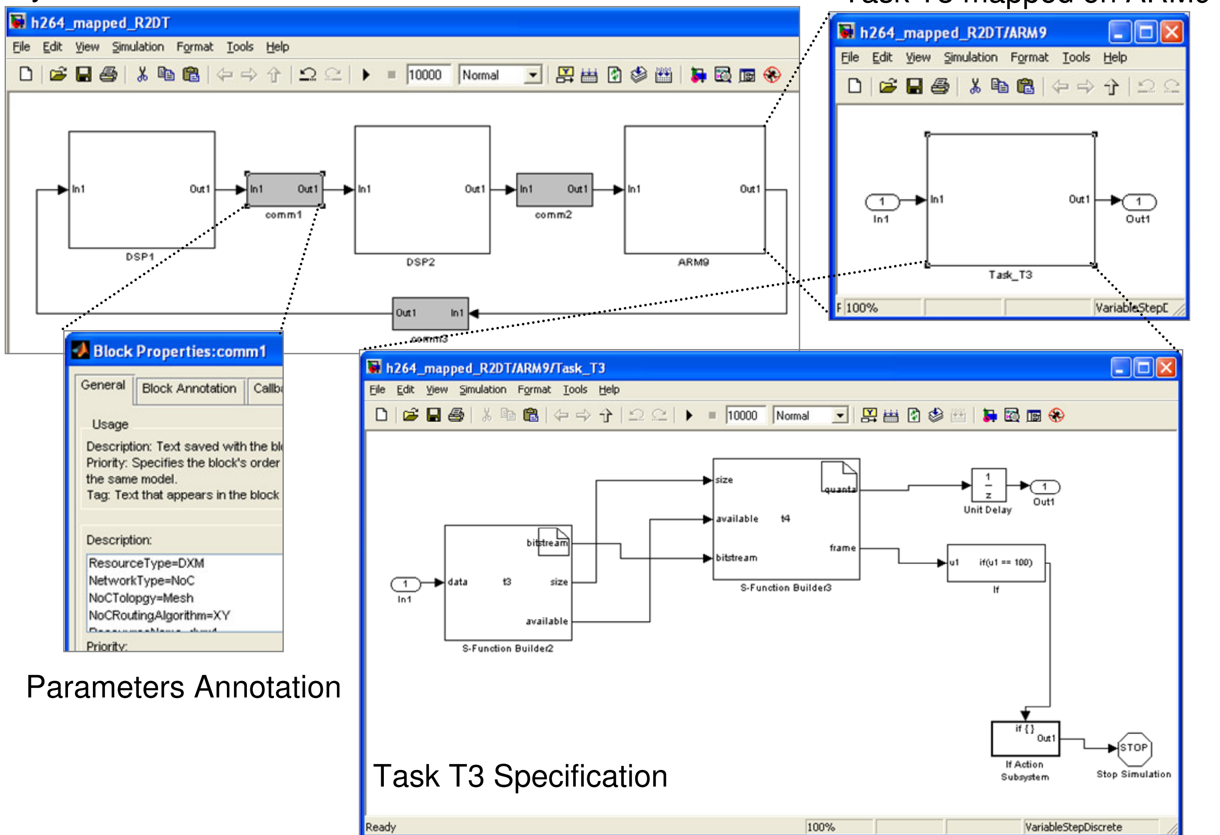


Figure 24. H.264 Encoder System Architecture Model in Simulink

Similar with the system architecture model of the MJPEG application, in order to allow the generation of the next levels, the high level application model of the H.264 contains the following architecture information annotated as parameters:

a) Parameters specific to subsystems:

- *ResourceType* for the subsystem in order to differentiate between the “ARM9” and “DSP” subsystems.
- *NetworkType* parameter, which has the value equal with “NoC”, as the target architecture will contain a NoC in the Diopsis architecture. This NoC implementation is based on the Hermes NoC [Mor 04].
- *NoCTopology*, with possible values “MESH” and “TORUS”, as the target architecture support both types of NoC topologies.
- *NoCRoutingAlgorithm* with possible value “XY” for the mesh topology or “NMWF” (Non Minimal West First) algorithm for the Torus topology [Gla 94].
- *NoCArbitrationAlgorithm* with value “ROUND_ROBIN” to arbitrate the simultaneous packet transmission requests on the router.
- *ResourceName* parameter to identify the hardware resource, e.g. the DSP1 or DSP2 processor as the architecture contains more DSPs of the same type (Atmel magicV VLIW processors).

As in this case the processors will execute single tasks, the software architecture parameters is represented only by the *OSType* parameter with value “DWARFOS” to specify the OS that will be responsible for the application boot, interrupt management and hardware access.

b) Parameters specific to communication units

- *ResourceType* to specify the communication protocol of the inter-subsystem communication unit (“DXM”, “SRAM”, “DMEM”, “REG”).
- *AccessType* parameter, required to specify whether the DSP processors will access the local memory of the ARM or the external global memory directly or by initiating a DMA transfer.
- *ResourceName* parameter to identify the hardware resource, e.g. the memory.

As software architecture parameter, the communication units are annotated with the *CommType* parameter with value “MPI” for the communication primitives.

To validate the H.264 encoder algorithm, the system architecture model can be simulated using the Simulink discrete-time simulation engine. The input test video represents a 10 frames video sequence QCIF YUV 420 format. The simulation time is 30s on a PC running at 1.73GHz with 1GBytes RAM.

The simulation allowed measuring some performance indicators. Thus, the total number of iterations necessary to decode the 10 frames video sequence was equal with the number of frames. This is due to the fact that all the S-functions implemented in Simulink operate at frame level. The communication between the DSP1 and DSP2 processors uses a communication unit that requires a buffer of 288585 words to transmit the encoded frame from the DSP1 processor to the DSP2 in order to be compressed. The DSP2 processor and the ARM9 processor communicate through a communication unit that requires a buffer of 19998 words. The last communication unit between the ARM9 and DSP1 processors requires 1 word buffer size in order to store the quanta value required for the encoder. The total number of words exchanged between the different subsystems during the encoding process of the 10 frames video sequence using Main Profile configuration of the encoder algorithm, was approximately 3085KWords.

2.7. State of the Art and Research Perspectives

2.7.1. State of the Art

Current literature includes several academic and industrial design environments that involve specification of the application mapping on the target architecture at the system architecture level.

The automatic parallelization of sequential program code is an open research topic. Several research works have already focused on automatic partitioning and tasks mapping, such as [Xu 06] and [Mei 07]. In [Fei 02], the authors propose an automatic partitioning of the application and automatic allocation of the computation resources using genetic algorithms. [Paz 04] proposes a programming paradigm that facilitates the translation of sequentially-code software algorithms of the multimedia applications into their parallel implementations.

Other research category focuses on finding the best mapping of an application onto the architecture by using different kinds of optimization algorithms and metrics. Examples of

these kinds of research works and tools are: PISA [Ble 03] which defines the mapping process as a multi-objective search problem, Compaan compiler [Mei 07] which automatically generates the mapping specification of an application modeled as Kahn Process Networks onto the Intel IXP Network Processor, Mat[01] which describes the APOTRES framework for mapping DSP applications on Single Program Multiple Data (SPMD) architectures, Bus[06] which presents a framework that automatically partitions a C application code into hardware and software or Xue[06] which treats the memory and processors allocation problem applying a runtime resource partitioner for multiple applications running on a shared memory MPSoC architecture. The Sesame environment described in [Erb 07][Tho 07] defines the optimal mapping problem taking into account three objectives: maximum processing time in the system, total power consumption and the cost of the architecture. The Sesame environment uses analytical methods to identify a small set of promising mapping candidates, and then uses simulation for performance evaluation. The DOL (Distributed Operation Layer) framework allows multi-objective algorithm mapping onto MPSoC architectures with system level performance analysis [Thi 07].

Besides the partitioning and mapping, other research works are related to the specification, modeling and simulation of the system architecture. In the DOL environment, the application is specified as Kahn Process Network [Thi 07]. The application, platform and mapping information are stored into three separate XML files. Many research efforts focus on the standardization of the XML format to facilitate various IPs exchange among different tools and IP providers. The IP XACT proposed by the Spirit consortium is an example of standardization proposal in form of an XML schema and APIs for representing and manipulating hardware IPs [Spirit].

An example of modeling environment is the well-known Ptolemy [Pto] for high-level system specification that supports description and simulation of multiple models of computation, e.g. synchronous dataflow (SDF), boolean dataflow (BDF), finite state machine (FSM), etc). The Ptolemy environment allows simulation at algorithmic level.

PeaCE [Ha 06] is a Ptolemy based co-design environment that supports hardware and software generation from a mixed dataflow and extended FSM specification. PeaCE also attempts to generate SoC architecture from an algorithm-level model. An extended version, the HOPES framework is a new model based programming environment of embedded software, which supports several environments for the initial specification (PeaCE, UML, KPN) [Ha 07].

Several other research groups investigate the specification and simulation of multimedia applications using Simulink and PeaCE [Kwo 04]. In [Rey 01] a design flow for data-dominated embedded systems is proposed, which uses Simulink environment for functional specification, and analysis of timing and power dissipation. This approach mainly focuses on an IP-based design with single processor.

Recently, UML is investigated as a system-level language. [Kan 06] proposes a UML-based MPSoC design flow that provides an automated path from UML design entry to FPGA prototyping, including the functional verification and the automated architecture exploration.

2.7.2. Research Perspectives

Despite the existing of a huge literature on system architecture design, this is still an open issue for heterogeneous MPSoC. There are 3 key problems concerning the system architecture design:

- Automatic partitioning of the application functions into tasks;
- Automatic mapping of the tasks onto the target architecture;
- Automatic mapping of the communication onto the target architecture. This includes communication buffer mapping on the storage resources and specification of the communication path.

Programming the complex MPSoC architectures and providing suitable software support (compiler and operating system) seems to be a key issue. This is due to the fact that either system designers or compilers will have to make the application code explicitly parallel to run on these architectures.

A first difficulty found in MPSoC design is how the applications running on these multi-processor architectures are decomposed in several processes/tasks and how these parallel tasks can share the same resources provided by the architectures. In particular, allocation of the computation resources (processing units) and storage resources (memories) is critical, as it dictates both performance and power consumption.

Automatic generation of the system architecture model represents one research perspective. Starting from the specification of the application, specification of the architecture and design constraints, the automatic partitioning and mapping could find the best

configuration. The specification of the application can be considered as being the functional model in Simulink, composed of several functions, similar with dataflow models.

The architecture specification has to include information related to the hardware resources of the target architecture, such as number and types of the available processors, size of the local and external memories and possible communication paths/protocols between the different processors. The communication paths can be captured using the notion of graph, where the nodes are the hardware resources of the architectures that may be crossed during a data exchange between the processors. Examples of nodes are the CPUs, coprocessors, DMA engines, memories, local buses or the global interconnect components (AMBA, NoC). The edges of the graph represent the connections between them. In this way, the specification of the communication path in the system architecture model (e.g. *AccessType*) is reduced to solve the shortest path problem between two nodes of a graph. The architecture specification can be stored using an XML format [Thi 07].

The design constraints may specify limitations for the relation between the application specification in Simulink and the resource components. They include:

- The constraints between the functions of the application and processors of the hardware architecture. For example, certain functions have to be grouped on the same task or several tasks must run on certain processors or certain processors types. One may also specify that certain tasks must be executed on the same processor.
- Application constraints. This concerns real-time execution requirements, e.g. deadline meeting constraints, execution time or communication latency.

Another method of system architecture generation may be based on profiling tools. The application specification can be profiled using specific code profiling tools. The profiling tools record summary information during the execution, (e.g. number of a function calls) and help to expose performance bottleneck and hotspots. Based on the profiling data and constraints information, analytical methods can propose an efficient application and communication mapping.

2.8. Conclusions

This chapter presented the system architecture design with case study in Simulink for the Token Ring application mapped on the 1AX architecture, the Motion JPEG decoder application mapped on the Diopsis RDT MPSoC architecture and the H.264 Encoder application mapped on the Diopsis R2DT architecture.

The hierarchical organization of the system architecture model allowed combining the application model with the specification of the partitioning and mapping of the computation and communication onto the hardware architecture resources.

The simulation at the system architecture level allowed to validate the functionality of the application and to profile the communication requirements of the applications (number of bytes that need to be exchanged during the execution).

Chapter 3

VIRTUAL ARCHITECTURE DESIGN

This chapter details the virtual architecture design. The virtual architecture design consists of transforming the application functions into the final application tasks C code and mapping the communication onto the hardware resources available in the target architecture. The key contribution in this chapter represents the virtual architecture definition, organization and design, using SystemC, for the Token Ring application running on the 1AX architecture, the Motion JPEG application targeting the Diopsis RDT architecture and the H.264 Encoder running on the Diopsis R2DT architecture. The simulation of the virtual architecture models allows validating the partitioning and final application tasks code. Different communication mapping schemes are explored in order to analyze their impact on the global performances.

3.1. Introduction

The virtual architecture design consists of mapping the communication onto the hardware platform resources and generating the final C code for each task. At this phase, the different links used for the communication between the different tasks of the system architecture model are mapped onto the hardware communication resources available in the architecture to implement the specified protocol. The system architecture tasks made of application functions are transformed into the final application tasks code. These tasks code designed in C is adapted to the communication mechanism through the use of adequate HdS communication primitives. The result of the virtual architecture design represents the virtual architecture model.

3.1.1. Definition of the Virtual Architecture

The second hardware-software abstraction level is called virtual architecture level (VA). The virtual architecture captures the global organization of the system into abstract software and hardware modules or subsystems and abstract hardware/software interfaces. The virtual architecture model may be manually coded or automatically generated by system architecture parser and analysis tools.

The objectives of the virtual architecture design are:

- Validation of the application partitioning and tasks mapping on the processing subsystems available in the target architecture
- Validation of the final tasks code of the software stack
- Early estimation of the communication requirements.

The virtual architecture is composed of abstract subsystems that are interconnected using abstract communication channels or abstract network components. The abstract hardware or software processing subsystem represents a component which implements the software tasks, respectively the hardware functions. The abstract communication network represents high level communication channels, such as message passing channels, abstract buses or NoC.

Figure 25 illustrates the global view of the virtual architecture, composed of two abstract software subsystems, a memory component and communication network. The left part of the figure corresponds to the hardware architecture, while the right part represents the software code at the virtual architecture level.

3.1.2. Global Organization of the Virtual Architecture

The virtual architecture model is a hierarchical model. The virtual architecture is composed of abstract subsystems that are interconnected using an abstract communication network, such as abstract bus, abstract NoC or abstract point-to-point communication channels.

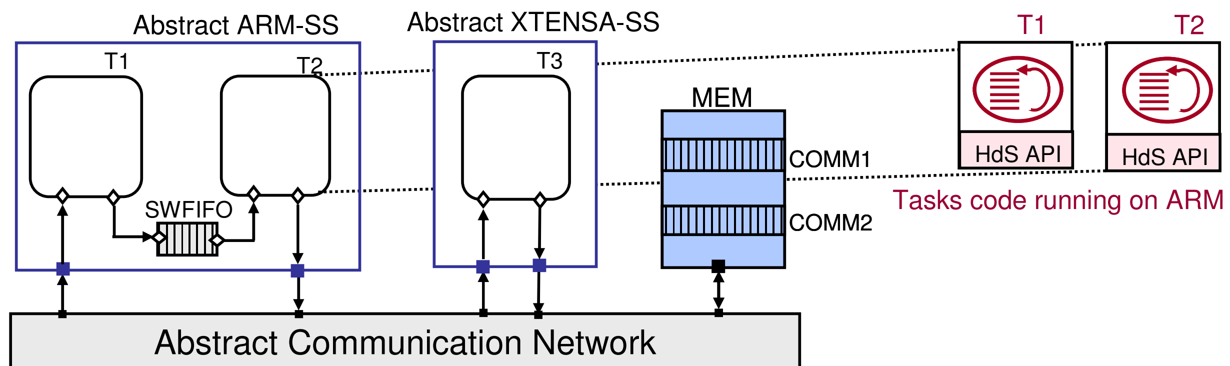


Figure 25. Global View of the Virtual Architecture

The abstract sub-systems may represent a processor subsystem, a hardware subsystem or a memory subsystem. The software processing subsystem represents a component or module which includes a set of task modules that are aimed to be executed on that processing subsystem and a set of abstract communication channels for the communication between the task modules inside the same subsystem. The hardware processing subsystem contains a single task module which implements the hardware functions.

The task modules abstract the hardware/software interface. Each task module can be characterized by two elements: container and ports. The container represents the task code. The task code is represented by a sequential C code which implements the application functions that were grouped together to form the task. The task code also contains communication primitives (HdS API) which allows accessing the ports of the task modules. The ports of the task module represent logic ports of the task, which serve to allow the software code to access the communication channels used for the data exchange with another

tasks. The logic ports of the task modules are connected to the ports of the subsystem that encapsulates them, or to the intra-subsystem communication channels.

At the virtual architecture level, the intra-subsystem communication units become abstract communication channels inside the processor subsystem. The inter-subsystem communication units become abstract communication architecture, and determine the memory modules that serve as storage resources for the communication buffer mapping and the type of global interconnect component. The type of the communication protocol and the topology of the network infrastructure are implemented according to the annotation of the system architecture model.

Example 12. Virtual Architecture of the Token Ring Application

Figure 25 shows a conceptual representation of the virtual architecture for the Token Ring application mapped on the 1AX architecture.

The virtual architecture contains two abstract subsystems (ARM-SS, XTENSA-SS), corresponding to the ARM, respectively XTENSA processors and the global memory module (MEM). All the subsystems are interconnected by an abstract AMBA bus. The different software subsystems encapsulate the application tasks and communication channels for the data exchange between the tasks mapped on the same processor. For instance, the ARM-SS subsystem includes the two task modules (T1 and T2) that were mapped on this processor and a SWFIFO communication channel used for the communication between T1 and T2. SWFIFO represents an abstract communication channel which implements a FIFO communication protocol. The XTENSA-SS subsystem includes the T3 task module.

The inter-subsystem communication units COMM1 and COMM2 are mapped on the global memory (MEM).

The virtual architecture model may be represented using different design languages, such as SystemC [Gro 02] or SpecC [Gaj 00]. In the following paragraphs, the virtual architecture will be detailed using the SystemC design language.

3.2. Basic Components of the Virtual Architecture Model

The basic components of the system architecture model are the software and the hardware components. The software components allow for a description of pure software elements, while the hardware components represent the components of the execution model [Verg 05]. The software components consist of the tasks code and HdS APIs, while the hardware components represent the abstract subsystems and the abstract communication network. The detailed description of these components will be illustrated in the following paragraphs.

3.2.1. Software Components

The application software is refined to tasks C code that contains the final application code and makes use of HdS API. The tasks code represents sequential code, which implements a set of application functions. The communication primitives of the HdS API access explicit communication components. Each data transfer specifies an end-to-end communication path. For example, the functional primitives *send_mem(ch,src,size)/recv_mem(ch,dst,size)* may be used to transfer data between the 2 processors using a global memory connected to the system bus, where *ch* represents the communication channel used for the data transfer, *src/dst* the source/destination buffer and *size* the number of words to be exchanged. Thanks to the HdS APIs, the tasks code remains unchanged at the following abstraction levels (transaction accurate architecture and virtual prototype).

Example 13. Software Components for the Token Ring application at the Virtual Architecture Level

For the Token Ring application, the software is represented by the sequential C code corresponding to tasks T1, T2 and T3. This code implements the equivalent behavior of the different Simulink functions in C and contains the communication primitives *send(ch, src,size)/recv(ch,dst,size)* for the data exchange between the diverse tasks.

3.2.2. Hardware Components

The software tasks are executed using an abstract model of the hardware architecture that provides an emulation of the HdS API. The hardware platform is composed of those components that provide the implementation these HdS APIs. Thus, it includes the abstract subsystems, abstract communication architecture (interconnection component) and the storage resources.

Example 14. Hardware Components for the Token Ring application at the Virtual Architecture Level

```
    For the Token Ring application, the hardware is represented
by the software subsystems (ARM-SS and XTENSA-SS), the global
memory MEM and the abstract communication network (abstract AMBA
bus).
```

3.3. Modeling Virtual Architecture in SystemC

The virtual architecture model is described using SystemC language and is generated according to the parameters specified in the initial Simulink model. SystemC allows modeling a system at different abstraction levels from functional to pin accurate register transfer level. The virtual architecture is modeled using transaction level modeling (TLM) techniques that allow analyzing SoC architecture in an earlier phase of design, software development and timing estimation [Gro 02].

3.3.1. Software at Virtual Architecture Level

At the virtual architecture level, the Simulink functions of the application are transformed into C program code for each task. This step is very similar to the code generation performed by Real Time Workshop (RTW) [Matlab].

Contrary to the RTW which generates only single task code, the software at the virtual architecture level represents a multitasking C code description of the initial Simulink application model.

Each data link of the Simulink model requires a memory space called buffer memory to deliver data from the input block to the output blocks. To reduce the required memory size, the task code generation has to apply buffer memory optimization techniques, such as copy removal or buffer sharing [Han 06].

The task C code is made of two parts: computation and communication. The computation part describes the behavior of various Simulink functions that are grouped in the task, including local memory declaration. The Simulink blocks within a task are scheduled statically according to their data dependency and generated into a task C code. The communication between functions inside the tasks is translated into local memory elements. To implement the external communication between tasks, during the task code generation the function calls of the communication primitives are instantiated from an HdS API template library, preserving the invocation order of the blocks. Then, the allocated memory spaces are mapped onto the arguments of these functions. Before inserting the communication primitives, data dependency between the tasks is checked during the task code generation in order to perform deadlock prevention.

Example 15. Software Task Code for the Token Ring Application

Figure 26 illustrates the C code of the task T2 of the Token Ring application at the virtual architecture level.

The task code contains the declaration of the local variables (*in*, *out*, *var* and *var2*), the *send_data/recv_data* communication primitives and the computation code. The tasks code starts with a receive operation to read the input token, then it performs some computation and finally it sends the new value of the token to the next node.

The semantic of the communication primitives is the following: the first parameter represents the logic port which is connected to a communication channel, the second parameter is the local memory from where the data is transferred in case of a *send* operation or the local address where the data is stored in case of the *receive* operation, and the last parameter defines the size in words of data to be sent or received.

In the computation code, the C code represents the equivalent behavior of the Simulink functions. Thus, the input data stored in the local variable *var* represents the input token. The destination of the token is calculated by a module operation

with 3 of the input token. If the result of this operation is 0, the destination of the token is task T2. In this case the task increments the token with 1 unit. Otherwise, task T2 is not the destination of the token and it increments the token with value 2, conform the application specification. Then, the task forwards the token to the next node of the ring.

```
#include <Task2.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#ifdef VIRTARCH
void Task2::behavior (void) {
#else
void Task2(void) {
#endif

int in,out;                                  //local variables
int var, var2;

for (;;) {
          recv_data(&In1_Task2, &in, 1);     //Communication API
          var = in % 3;                      // Computation
          var = abs(var);
          if (var == 0)
                    var2 = in + 1;
          else
                    var2 = in + 2;
          out = var2;
          send_data (&Out1_Task2, &out, 1); //Communication API
#ifdef VIRTARCH
          wait();
#endif
          }
}
```

Figure 26. Task T2 Code

The multitask C code generation from the system architecture model needs to handle a large subset of pre-defined Simulink blocks, such as mathematical operations (sum, multiplication, division, modulo etc), logical operations (AND, OR, XOR), discrete blocks (delay, mux, demux, merge), conditional structures (if-then-else), repetitive structures (for-loop, while-condition-loop). The generation has to support also user defined C codes integrated in the Simulink model as S-functions. For the S-functions, the task code represents a function call of the user written C function. The semantics of the argument passing are

identical to those of the definition in the configuration panel of the S-Function Builder tool in Simulink.

The resulted tasks code at the virtual architecture level is independent of the target processor, communication protocol and abstraction level. This can be achieved by using HdS APIs that hide many details of the underlying implementation of the architecture and represent the abstraction of the hardware [Vin 04].

3.3.2. Hardware at Virtual Architecture Level

The hardware at the virtual architecture level consists of the set of hardware and software subsystems that encapsulate the tasks aimed to be executed on those subsystems and the abstract communication network introduced to implement the communication protocol.

The hardware is refined to a set of abstract SystemC modules (SC_MODULE) for each subsystem. The SC_MODULE of the processor includes the tasks modules that are mapped on the processor and the communication channels for the intra-subsystem communication between the tasks inside the same processor. The communication channels between the tasks mapped on the same processor are implemented using standard SystemC channels. The tasks modules are implemented as SystemC modules (SC_MODULE).

For the inter-subsystem communication, the hardware architecture integrates also the resources addressed explicitly by the HdS APIs. Typical examples are memories that serve to store the communication buffers. The interconnection between the different components uses an abstract model of the communication network that allows the data transfer from the source to the destination module.

Example 16. Hardware Code for the Token Ring application at the Virtual Architecture Level

Figure 27 details the Top module for the Token Ring application running on the 1AX architecture. The top module is a SC_MODULE which includes the declaration and instantiation of the ARM-SS (*vARM7* in figure 27), XTENSA-SS (*vXTENSA*), abstract bus (*bus*), global memories (*gmem*) and a global clock (*clk*). It also interconnects these different components and fixes the addresses of the communication buffers used for the data exchange between the processors (the inter-subsystem communication units).

Thus, the communication buffer used between tasks T3 running on XTENSA and T2 running on ARM7 processors is mapped in the global memory at address 0x0. The communication buffer required for a data transfer between tasks T1 mapped on the ARM7 processor and T3 executed on the XTENSA is mapped on the global memory at address 0x1000.

```
ifndef _TOP_H
#define _TOP_H

#include <systemc.h>
#include "XTENSA.h"
#include "ARM7.h"
#include "global_bus.h"
#include "mem.h"

SC_MODULE(TOP) {                                  //TOP MODULE

sc_in<bool> clk;
XTENSA *vXTENSA;                                  // XTENSA-SS MODULE
ARM7 *vARM7;                                      // ARM-SS MODULE
global_bus_module *bus;                           // BUS CHANNEL
mem *gmem;                                        // GLOBAL MEMORY MODULE

SC_CTOR(TOP) {
          vXTENSA = new XTENSA("XTENSA");         //INSTANTIATION
          vARM7 = new ARM7("ARM7");
          bus = new global_bus_module("BUS");
          gmem = new mem("mem",0x2000);

          gmem->port(*bus);                       //CONNECTION
          gmem->port.set_map(0x0,0x1FFF);
          vXTENSA->In1_XTENSA(*bus);
          vXTENSA->vTask3->In1_Task3.set_connect_address(0x0);
          vARM7->Out1_ARM7(*bus);
          vARM7->vTask2->Out1_Task2.set_connect_address(0x0);
          vARM7->In1_ARM7(*bus);
          vARM7->vTask1->In1_Task1.set_connect_address(0x1000);
          vXTENSA->Out1_XTENSA(*bus);
          vXTENSA->vTask3->Out1_Task3.set_connect_address(0x1000);
          vARM7->clk(clk);
          vXTENSA->clk(clk);
          }
};
```

Figure 27. SystemC Code for the Top Module

Figure 28 shows the SystemC code for the ARM-SS component. The ARM-SS is a SC_MODULE which encapsulates the instances of the two tasks T1 (*vTask1* in figure 28) and T2 (*vTask2* in figure 28) and the software FIFO channel for communication between them (*ch1* in figure 28).

This ARM-SS has 3 ports: an input port, namely *In1_ARM7*, an output port, namely *Out1_ARM7*, and a clock port (*clk*). The output port of the ARM-SS is connected to the output port of task T2, as

task T2 sends data to a task mapped on the other processor. The input port of the ARM-SS is connected to the input port of task T1, because task T1 needs external data from a task running on another processor.

```
#ifndef _ARM7_H
#define _ARM7_H

#include "Task1.h"
#include "Task2.h"

SC_MODULE(ARM7) {                           // ARM-SS MODULE

Task1 *vTask1;                              // TASK T1 MODULE
Task2 *vTask2;                              // TASK T2 MODULE
SWFIFO_Channel ch1;                         // SOFTWARE FIFO CHANNEL
AMBA_Port Out1_ARM7;                        // PORTS
AMBA_Port In1_ARM7;
sc_in<bool> clk;

SC_CTOR(ARM7) {
          vTask1 = new Task1("Task1");  //INSTANTIATION
          vTask2 = new Task2("Task2");

          vTask1->Out1_Task1(ch1);      //CONNECTION
          vTask2->In1_Task2(ch1);
          vTask2->Out1_Task2(Out1_ARM7);
          vTask1->In1_Task1(In1_ARM7);
          vTask1->clk(clk);
          vTask2->clk(clk);
          }
};

#endif
```

Figure 28. SystemC Code for the ARM-SS Module

At the virtual architecture level, the tasks code uses HdS APIs, whose implementation depends on the hardware platform. The hardware platform includes all the components accessed by HdS API and the resources to implement the required communication paths.

Example 17. Communication primitives implementation for the Token Ring application at the Virtual Architecture Level

Figure 29 shows an example of implementation of the *send_data(…)/recv_data(…)* communication primitives that allow to write or read to/from a software FIFO communication channel.

The FIFO channel is derived from a SystemC channel (sc_prim_channel) and has a blocking implementation. Therefore, if the sender wants to put data into the FIFO, but the FIFO is full, the sender will be blocked until there is enough available space in the buffer. This blocking implementation is employed by calling the *wait ()* statement in the implementation of the *send_data* primitive if there's not enough space available. The *wait()* call suspends the execution of the task. In the same manner, if a task calls a *recv_data* primitive, but there's not enough data stored in the FIFO buffer, the receiver will be blocked until the FIFO contains the requested number of elements. The FIFO buffer can be characterized by size and depth. The size represents the number of elements stored in the buffer. The depth represents the number of bits necessary to store one element. Each element occupies the same number of bits. In this example (figure 29), the FIFO buffer has a size equal with 30000 and depth equal with 1 word (32 bits).

```
class SWFIFO_Channel: public sc_prim_channel,      //SOFTWARE FIFO CHANNEL
public swfifo_if
{
int buffer[30000];
unsigned int sizemax;
int buffer_size;

public:
      SWFIFO_Channel() {
                  sizemax=SIZEMAX_swfifo;
                  buffer_size =0;
                  }
      virtual void recv_data(const SWFIFO_Port& port, void* dst, int size);
      virtual void send_data(const SWFIFO_Port& port, void* src, int size);
      virtual void init(const SWFIFO_Port& port, int size);
};


void SWFIFO_Channel::send_data(const SWFIFO_Port& port,       // SEND_DATA
void* src, int size)
{
 while((sizemax - buffer_size) < (unsigned)size)
          wait();

 for (int i=0; i<size; i++)
          buffer[buffer_size+i]=(((int*)src)[i]);

 buffer_size = buffer_size + size;
}
```

Figure 29. Example of Implementation of Communication Channels

At the virtual architecture level, all the modules are connected to the same clock signal. Typically the clock is created in the main function of the top level and passed down through

the module hierarchy to the rest of the system. This allows subset of components or the entire system to be synchronized by the same clock. The clock signal has a set of attributes, such as default time unit, period, duty cycle, first edge and first value. The default time unit is assumed to be 1 nanosecond. The period represents the number of default time units required by the clock signal to make a complete transition from true (high) to false (low) and back from false (low) to true (high). The duty cycle is the ratio of the high time to the entire clock period. E.g. if the period of a clock signal is equal with 20 default units and the duty cycle is 0.25, this means that the clock would stay in true state for 5 time units and false for 15 time units. The first edge represent the offset time from 0 of the first edge expressed in time units. The first value represents the starting value of the clock (true or false).

Example 18. sc_main for the Token Ring application at the Virtual Architecture Level

Figure 30 presents the main function (*sc_main*) for the Token Ring application at the virtual architecture level. This includes the initialization of the top module, the declaration of the global clock signal, the connection of the clock signal to the clock port of the top module and the launch of the simulation. The clock has a period of 20ns and duty cycle 0.5.

```
int sc_main(int argc, char ** argv)
{
          TOP top_module("TOP");             //TOP MODULE INSTANTIATION

          sc_clock s_clk("s_clk",20,0.5,0);  // CLOCK SIGNAL
          top_module.clk(s_clk);

          sc_start(-1);                      //START SIMULATION
          return 0;
}
```

Figure 30. SystemC Main Function

3.3.3. Hardware-Software Interface at Virtual Architecture Level

The hardware-software interface defines the software-hardware interaction and how the software can access the hardware. At the virtual architecture level, the hardware/software interface consists of a set of task modules. The task module is a SC_MODULE which

encapsulates the software code within a SystemC clocked thread (SC_CTHREAD). The software code may access the hardware through the ports of the task module.

Example 19. Task Module for the Token Ring application at the Virtual Architecture Level

Figure 31 illustrates an example of task module for the task T2. The task module contains the declaration of the logic ports. The type of the ports depends on the type of communication channel which is accessed. For instance, the input port of task T2 *In1_Task2* is connected to the software FIFO channel, thus the port has the type SWFIFO_Port. Task T2 writes to task T3 running on the XTENSA processor via the AMBA bus. Therefore, the type of the output port *Out1_Task2* is AMBA_Port.

```
#ifndef _Task2_H
#define _Task2_H

#include <systemc.h>
#include "swfifo.h"
#include "amba.h"

SC_MODULE (Task2) {                                 //TASK T2 MODULE

          sc_in<bool> clk;

          SWFIFO_Port In1_Task2;
          AMBA_Port Out1_Task2;

          void behaviour();

          SC_CTOR(Task2) {

                    SC_CTHREAD(behavior,clk);   // THREAD

          }
};

#endif
```

Figure 31. Example of Hardware/Software Interface

Task T2 executes a clocked SystemC thread, namely the *behavior* function. The *behavior* function is defined in the task software code, as illustrated in figure 26. A clocked SystemC thread represents a thread of execution which is sensitive only to the positive or negative edge of a clock signal.

3.4. Execution Model of the Virtual Architecture

The virtual architecture level allows debugging the task code. The following sections will describe the simulation model in SystemC and the adopted configuration to validate the virtual architecture model.

The executable model is obtained by compiling the task code and hardware platform together. The resulted executable model uses the SystemC scheduler to activate and deactivate the execution of the different tasks. The processor and memories are SystemC modules. The abstract network component (bus, NoC or point-to-point communication channels) and the software FIFO channels are derived from the SystemC channels. The software tasks are SystemC clocked threads. A clocked thread has its own thread of execution which may accept only positive or negative edge clock event in its sensitivity list. When the simulation starts, the clocked threads are automatically activated.

The simulation at the virtual architecture level allows validating the tasks C code of the refined software and the hardware-software partitioning. It represents a native execution of the software onto the simulation host machine. High simulation speed is usually attained, but abstracting the hardware architecture it lacks some accuracy.

The simulation at the virtual architecture level allows avoiding communication deadlock due to improper scheduling of the communication operations between the different tasks. The debug of the software code may be done using standard C debuggers such as gdb, or by tracing waveforms. SystemC provides functions to create a VCD (Value Change Dump) or ASCII WIF (Waveform Intermediate Format) file that contains the values of variables and signals as they change during the simulation. The waveforms can be viewed using standard waveform viewers that support the VCD and WIF formats, such as gtkwave.

3.5. Design Space Exploration of Virtual Architecture

3.5.1. Goal of Performance Evaluation

The goal of performance evaluation at the virtual architecture level is to allow profiling the communication and computation requirements and improve the overall performances of the system. The objective is to provide through simulation statistical information, such as utilization of the architecture model components (busy/idle times), the

degree of the contention in a system, profiling information (time spent in different executions), critical path analysis or average bandwidth between the architecture components.

Based on the application requirements and the communication traffic resulted after the virtual architecture simulation, the designer can fix some hardware and software architecture decisions. Examples of hardware architecture decisions are: the topology of the interconnect component that will be included in the hardware platform at the next abstraction level (NoC topology) or the communication scheme between the different subsystems fixing the mapping of the communication buffers onto the storage resources. Examples of software architecture decisions are: application partitioning into tasks and mapping onto the processing subsystems and the semantic of the communication primitives used in the final application tasks code.

These different decisions influence the overall execution time of the system, cost and power consumption. Therefore, good decisions are required to be able to control the MPSoC design process.

Example 20. Goal of Performance Evaluation for the Token Ring application at the Virtual Architecture Level

```
    For example, in case of the Token Ring application, the
designer fixes at the virtual architecture level the partitioning
of the application into the 3 tasks and the mapping of the FFT
computation onto the XTENSA processor. Also, the communication
between the processors is decided to be performed via the global
memory.
```

3.5.2. Architecture/Application Parameters

The virtual architecture model has to fix some parameters that can influence the global performance of the final system. The parameters represent a subset of those specified at the system architecture level. The virtual architecture validate some decisions taken at the system architecture level, such as partitioning and mapping, while other parameters are preserved in order to be validated at the next levels. The preserved parameters can be specific to subsystems or communication units, as it will be detailed in the following paragraphs.

a) Architecture/ Application Parameters specific to Subsystems

The parameters specific to subsystems characterize the different subsystems from hardware and software points of view. The hardware architecture parameters that characterize the subsystems at this level and will be validated at the following abstraction levels are:

- *NetworkType* to specify the type of the network component used to interconnect the different subsystems, such as AMBA bus or Network-On-Chip (NoC).
- *NoCTopology* to specify the bus or NoC topology (Mesh or Torus).
- *NoCRoutingAlgorithm* to specify the routing algorithm used by the routers to transmit the received data packet in case of a NoC network component.
- *NoCArbitrationAlgorithm* to specify the type the arbitration algorithm inside a NoC router (e.g. round robin, priority based, etc).

The software architecture parameters that characterize each processor subsystem are:

- *OSType*, which specifies the name of the operating system running on the target processor (e.g. Linux, Mutek, DwarfOS, eCos).
- *SchedulerType* to identify the type of the scheduler (preemptive, cooperative).
- *SchedulerAlgorithm* to define the algorithm used for the tasks management by the operating system (round-robin, priority based) etc.

b) Architecture/ Application Parameters specific to Communication Units

The communication primitives used for the data exchange in the tasks code are fixed at the virtual architecture level. Therefore, the parameters that characterize the communication units and will be validated at the next abstraction levels rely on the hardware architecture. Example of this kind of parameters is the *AccessType* which identifies the type of the access to the memory (directly or through DMA) making the communication path end-to-end.

3.5.3. Performance Measurements

At the virtual architecture level, the performance measurement consists of profiling the communication and computation requirements for each task or for each processor.

The virtual architecture has the notion of time due to the clock signals. Therefore, by simply annotation of the virtual architecture model with adequate execution delays, if such

delay information is available, the simulation at this level can estimate the total clock cycles spent on communication or computation by task or processor. But, the accuracy of the estimation is not yet cycle accurate at this level, since not all the hardware components or hardware features (e.g. final bus arbitration scheme, interconnect topology, peripherals) are explicit in the model. The execution time represents these estimated clock cycles required to run an application on the MPSoC architecture. The simulation time represents the time needed to simulate the behavior of the application running on architecture with their interaction.

Examples of metrics that can be measured at the virtual architecture level are: tasks code and data size, buffer size required for the intra-subsystem and inter-subsystem communication, the total quantity of exchanged data between the tasks during the execution, the number of iterations of a function execution, the amount of data transferred between the different processors, the amount of data passing through the global interconnect component, the buffer size requirements in the worst case scenario for the storage resources in order to support the communication mappings specified at the system architecture level or the amount of read/write operations performed at the storage modules for the communication, etc.

By tracing the waveforms of signals or variables during the simulation, other metrics can be measured, e.g. cycles spent by task on computation and communication, current task executed during the simulation, etc.

Example 21. Performance measurements for the Token Ring application at the Virtual Architecture Level

For example, the total simulation time of the Token Ring application was 3 seconds to execute 68 clock cycles of period 20 nanoseconds, required to run the entire application. But in this example, the model is not annotated with accurate information regarding the operating system and the communication overhead. Thus, the estimation is only message level accurate.

In the case of the Token Ring application, the total code size and data size of the tasks code running on both processors is 32223 bytes, respectively 12 bytes. The total number of bytes passed through the bus during the simulation is 3136 bytes.

Figure 32 shows the waveforms captured during the simulation of the Token Ring application, e.g. at time 13100 ps, the current

task running on the ARM processor was T1, while task T2 was blocked on communication.

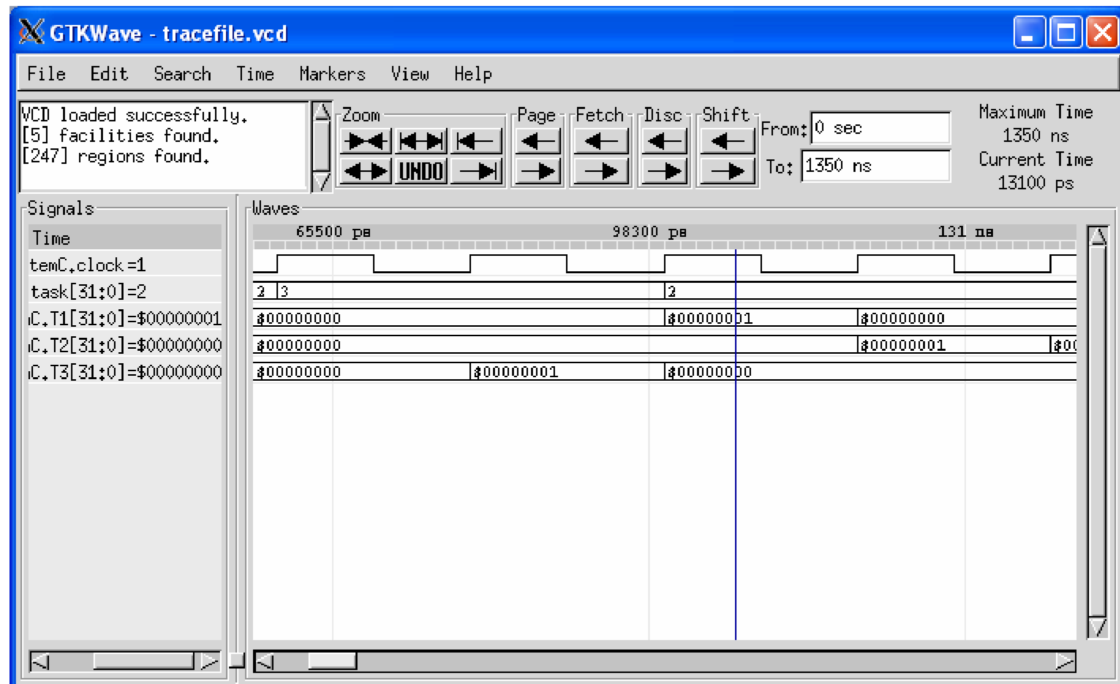


Figure 32. Waveforms Traced during the Token Ring Simulation

3.5.4. Design Space Exploration

At the virtual architecture level, the design space exploration covers architecture exploration, more precisely communication architecture exploration. The designer can experiment different communication mapping schemes and different communication primitives. The designer may adopt different communication protocols and may map the communication buffers onto different storage resources. The different communication and synchronization schemes have advantages and disadvantages in terms of performance (latency, throughput), resource sharing (multitasking, parallel I/O) and communication overhead (memory size, execution time). Also, the tasks code can be generated using different tools, such as Real Time Workshop.

Example 22. Design Space Exploration for the Token Ring application at the Virtual Architecture Level

In the case of the Token Ring application, the designer may map the buffers required for the inter-subsystem communication

```
onto different architecture resources, such as the local memories
of both ARM and XTENSA processors, or the shared global memory or
on the hardware FIFO.
```

3.6. Application examples at the Virtual Architecture Level

The following sections detail the virtual architecture model for the two case-studies considered in this document: the Motion JPEG Decoder application running on the Diopsis RDT architecture with AMBA bus and the H.264 Encoder application running on the Diopsis R2DT architecture with Hermes NoC.

3.6.1. Motion JPEG Application on Diopsis RDT

This section presents the virtual architecture design in case of the Motion JPEG (MJPEG) Decoder application running on the Diopsis RDT platform. The virtual architecture design consists of two steps: software design and hardware design.

Firstly, the C code for each task was generated from the Simulink system architecture model based on the annotation with the software architecture parameters. In the system architecture model, the value attributed to parameter *CommType* is equal with "MPI". Therefore, the generated task code uses *send_data(…)/recv_data(…)* for the communication primitives. Moreover, the C code was optimized by applying buffer-sharing and copy-removal memory optimization techniques.

In order to evaluate the efficiency of the software task code, a comparison with the single task code generation from Simulink using Real Time Workshop (RTW) is given. Table 1 resumes the code and data size of the generated application code.

Table 1. Task code generation for Motion JPEG

Size	Library		RTW		Multitask code	
(Bytes)	Code	Data	Code	Data	Code	Data
MJPEG	6818	32	8225	72	8032	494

The code library contains the user defined C-functions commonly used by all the code generator tools and independent of the software design method. The application task code obtained by applying memory optimization techniques is more efficient in terms of code size

than the code generated using RTW. But the multitasked representation requires communication buffers. Therefore, the data size is bigger than in the case of RTW, which generates only single task code.

The second step of the virtual architecture design represents the hardware design. The hardware design consists of building the software development platform in SystemC considering the hardware architecture parameters that annotate the system architecture model. Figure 33 illustrates a conceptual view of the virtual architecture of the Diopsis RDT architecture with AMBA bus.

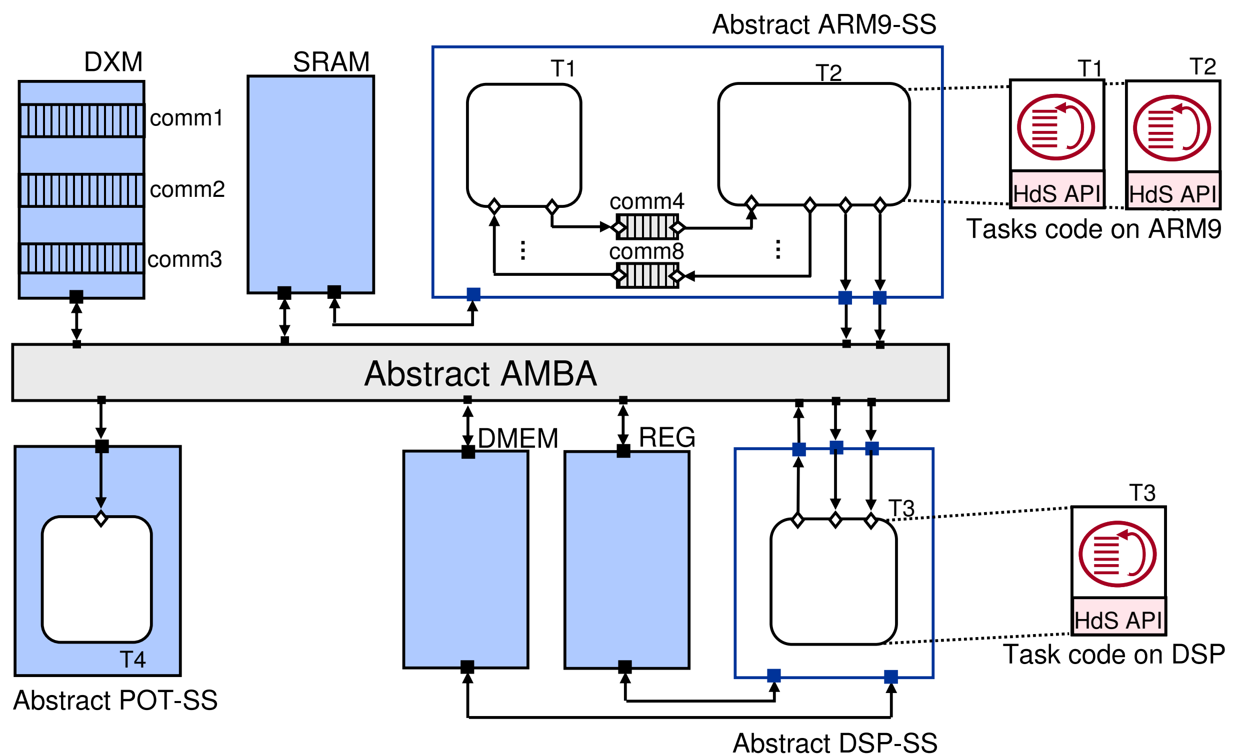


Figure 33. Global View of Diopsis RDT running MJPEG

The virtual architecture platform contains all the components that are accessible by the software through the *send_data(…)/recv_data(…)* HdS APIs. The inter-subsystem communication units are partially mapped on the memory modules (DXM, REG, SRAM, DMEM), attached as slave components to the AMBA bus. The impartiality comes from the abstraction of the hardware architecture and making implicit several hardware characteristics (e.g. local bus, DMA, bus bridges, etc). The address space of the components are automatically assigned and computed by using a template that contains a predefined address size for each component. The communication buffers between the different subsystems are

mapped on the corresponding memory modules based on the protocol specified at the system architecture level.

As showed in figure 33, the virtual architecture contains the following components: abstract ARM9-SS, abstract DSP-SS, abstract POT-SS and the storage resources: DXM, local memory SRAM of the ARM9 processor, data memory DMEM of the DSP processor and data register REG of the DSP. All the components are interconnected using an abstract AMBA bus. The ARM9-SS contains the two task modules that are running on it (T1 and T2) and the five intra-subsystem communication channels (*comm4*, *comm5*, *comm6*, *comm7* and *comm8*). According to the system architecture annotation, all these communication channels are implemented as FIFO channels. The DSP-SS includes the task module T3. The POT-SS includes the task module T4 responsible with the display of the decoded image. The three communication units between the subsystems, more precisely *comm1*, *comm2* and *comm3*, may be mapped on different storage resources. In a first case, the *ResourceTyp*e of each communication unit has the value "DXM". Hence, the system architecture model specified to use the external memory as buffer storage for the communication between the different subsystems. Therefore, all three were mapped on the external memory DXM.

The abstract AMBA bus is implemented as a simple bus which transfers data initiated by a master. It allows connection of several master and slave subsystems, but only one data transfer request can be accomplished in time. The bus has a scheduler or arbiter which controls the data traffic. If data is to be transferred, the requesting master subsystem sends a message to the scheduler. The scheduler checks if the slave subsystem is ready for the data transfer. If it is ready, the scheduler puts the request into a FIFO queue. Otherwise, it waits until the availability of the slave component and checks its status by polling. This mechanism allows avoiding blocking the bus for a data transfer to a destination or source which is not yet ready to receive or send data. The request message contains an identification code of the target subsystem, which represents the address of the slave component. The decoder is responsible to identify the slave subsystem. As soon as the bus is available, the access to the bus is granted to the requesting subsystem, which can perform the data transfer to the destination address. Having completed the data transfer, the bus becomes free for the next request in the scheduler's queue. The AMBA bus allows transfers in burst mode, which means that the master may transfer the whole data message within one access grant to the bus [Arm]. Through polling the status of the destination subsystem, the virtual architecture bus provides synchronization mechanism between the different subsystems similar to semaphores.

The functionality of the software code was validated by execution using the hardware platform. The software code was compiled with the architecture platform. During the execution, the tasks are scheduled by the SystemC simulation engine.

Besides the task code validation, the simulation model allowed also to gather important early performance measurements, e.g. total number of messages transferred through the AMBA bus. The data transfer between the ARM9 and DSP processor subsystems is performed in messages of 64 words for the IDCT coefficients and in 1 word for the decoding pattern; the data transfer between the DSP-SS and POT-SS is performed in messages of 16 words.

Table 2 shows the results for different communication schemes. Using as communication units only the DXM, the bus was accessed to transfer 216000 messages during the decoding process of the ten frames. If the communication units are mapped on different resources, for example *comm1* is mapped on DXM, *comm2* on REG and *comm3* uses DMEM memory to store the communication buffer, the global bus was accessed to transfer 144000 messages during the simulation. In the third scheme, *comm1* and *comm2* are mapped on SRAM, while *comm3* remains mapped on DMEM. Thus, all the communication units make use only of local memories SRAM and DMEM. In this case, the execution required 108000 messages to be transferred via the AMBA bus. In all the communication schemes, the communication units between the two tasks running on the ARM9 processor *comm4*, *comm5*, *comm6*, *comm7*, *comm8* and *comm9* implements the software FIFO protocol.

Table 2. Messages through the AMBA bus

Comm. Unit	Comm1	Comm2	Comm3	Comm4-Comm8	Total messages AMBA	Execution Time (ns)
MJPEG	DXM	DXM	DXM	SWFIFO	216000	4464060
	DXM	REG	DMEM	SWFIFO	144000	3720060
	SRAM	SRAM	DMEM	SWFIFO	108000	2232020

This simulation model was accurate enough to validate the functionality of the task code and ensure that there is no communication deadlock in the scheduling of the data transfer between the tasks. The simulation time required to decode the ten image frames encoded using QVGA YUV 444 format was approximately 14s on a PC running Linux OS at 1.73GHz in all the cases of communication schemes.

The total execution time required by the whole decoding process is illustrated in table 2. These numbers were estimated without annotation of the code with execution delays. Therefore, the accuracy of the estimation relies on message level. As the DXM communication scheme supposes all the data exchange through the AMBA bus, it requires the highest number of execution cycles, in a total time of approximately 4464060ns, due to the conflicts that appear on the shared bus when simultaneous bus requests occur. In the mixed communication scheme with DXM, REG and DMEM, the total number of execution time is estimated to be 3720060ns, while the last communication scheme guarantees the fastest execution with 2232020ns. The numbers for required execution time are obtained by calling *sc_simulation_time()* at the end of the execution in the top module of the SystemC platform. All the subsystems are interconnected to the same clock signal with period of 20 nanoseconds and duty cycle 0.5.

Figure 34 presents a screenshot during the SystemC simulation of the MJPEG Decoder application at the virtual architecture level.

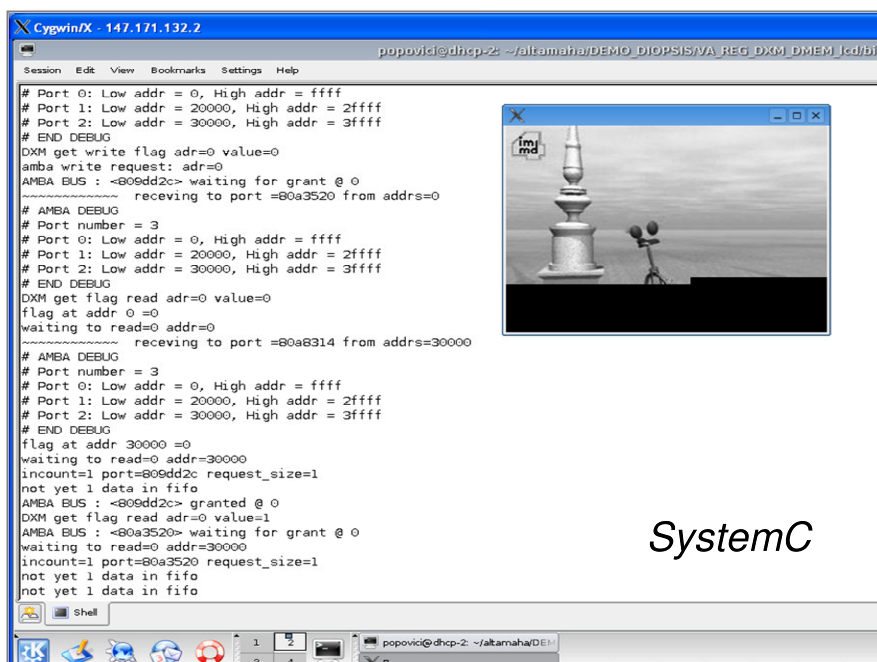


Figure 34. Virtual Architecture Simulation for Motion JPEG

3.6.2. H.264 Application on Diopsis R2DT

This section presents the virtual architecture design in case of the H.264 Encoder application running on the Diopsis R2DT platform. The virtual architecture design is accomplished in two steps: software design and hardware design.

The software design consists of generating the C code for each task from the system architecture model using the software architecture parameters. The *CommType* parameter annotating each subsystem determines the communication primitives supported by the operating system. Similar with the Motion JPEG example, the *CommType* is equal with "MPI". Therefore, the generated task code uses *send_data(…)/recv_data(…)* for the communication primitives. The code is optimized in terms of data memory requirements.

Table 3 shows the task code and data size of the software at the virtual architecture level. The first column represents the code and data size of the functions that are independent of the design method. The second column shows the code and data size in case of generation using Real Time Workshop. Real Time Workshop generates single task code, while the software at the virtual architecture level represents multitask code. The last column represents the results for the software design method with memory optimization techniques.

Table 3. Task code generation for H.264 Encoder

Size (Bytes)	Library Code	Library Data	RTW Code	RTW Data	Multitask code Code	Multitask code Data
H.264	270994	132	296305	148	366060	148

The second step of the virtual architecture design represents the hardware architecture design. The hardware design consists of building the software development platform in SystemC considering the hardware architecture parameters that annotate the system architecture model. Figure 35 illustrates a conceptual view of the virtual architecture for the Diopsis R2DT architecture with Hermes NoC.

The virtual architecture platform contains all the components that are accessible by the software through the *send_data(…)/recv_data(…)* HdS APIs. Thus, the platform contains the following modules: four abstract subsystems, namely the ARM9-SS, DSP1-SS, DSP2-SS and POT-SS, and the local and global memory modules: DXM shared by all the subsystems, SRAM local memory of the ARM9-SS, DMEM1 and REG1 memories of the DSP1-SS, respectively DMEM2 and REG2 memories of the DSP2-SS. All these components are interconnected using an abstract NoC model. The DSP1-SS contains the task module of T1. The DSP2-SS includes the task module of T2. Finally, the ARM9-SS encapsulates the task module of the third task T3. The three communication units between the different processors, more precisely the *comm1*, *comm2* and *comm3*, may be mapped on different storage resources,

according to the system architecture specification. Figure 35 shows an example of mapping of the communication units onto the DXM global memory.

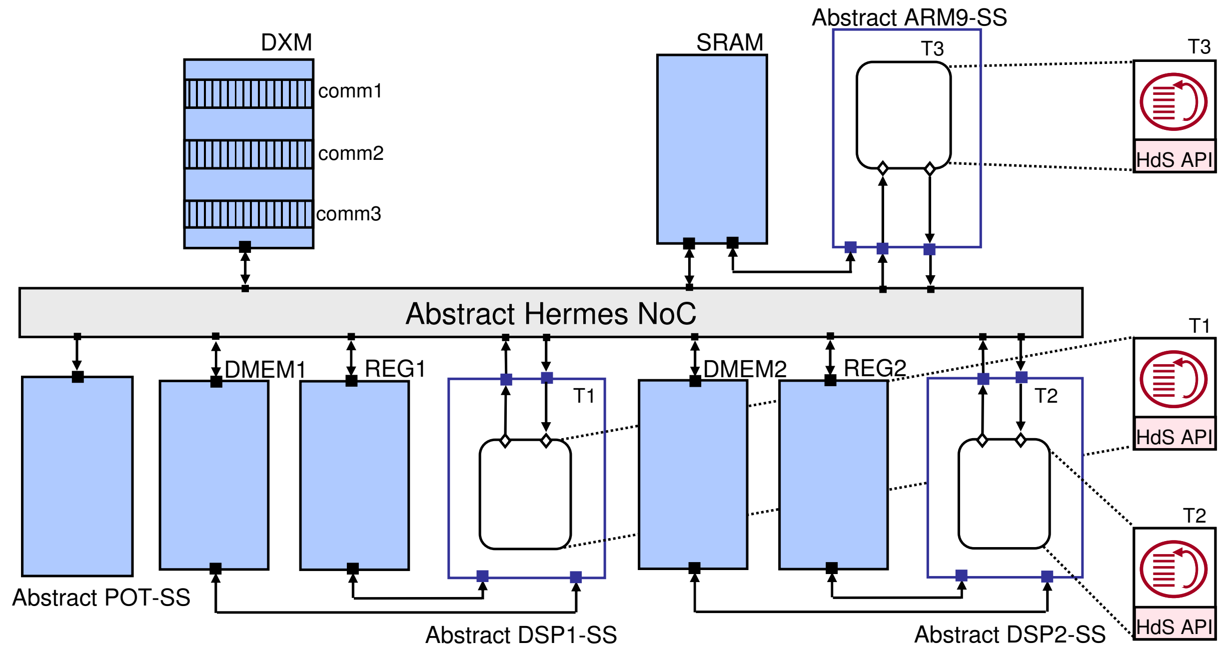


Figure 35. Global View of Diopsis R2DT running H.264

At this level, each local memory has allocated an address space of 4MB. The global memory has an address space of 256MB.

The NoC at the virtual architecture level represents an abstract NoC where information like topology, routing algorithm, arbitration or buffer size information are omitted. Communication architecture is modeled like a crossbar, where any set of communication events may occurs simultaneously.

Figure 36 details the model of the Hermes NoC at the virtual architecture level.

The NoC is composed of three basic elements, which are the network interface (NI), the mapping table (MT) and the router. The network interface is responsible for providing *send/receive* operations for communicating subsystems, encapsulating these requests in packets, capturing and interpreting packets arriving from the NoC and delivering them to the subsystems. The mapping table is responsible for storing and informing the correspondence between the IP cores range address and NoC physical address, i.e. IP core address between 0x00400000 and 0x007FFFFF correspond to NoC physical address 0x0. The router is in charge of transporting packets from the source network interface to the destination network interface.

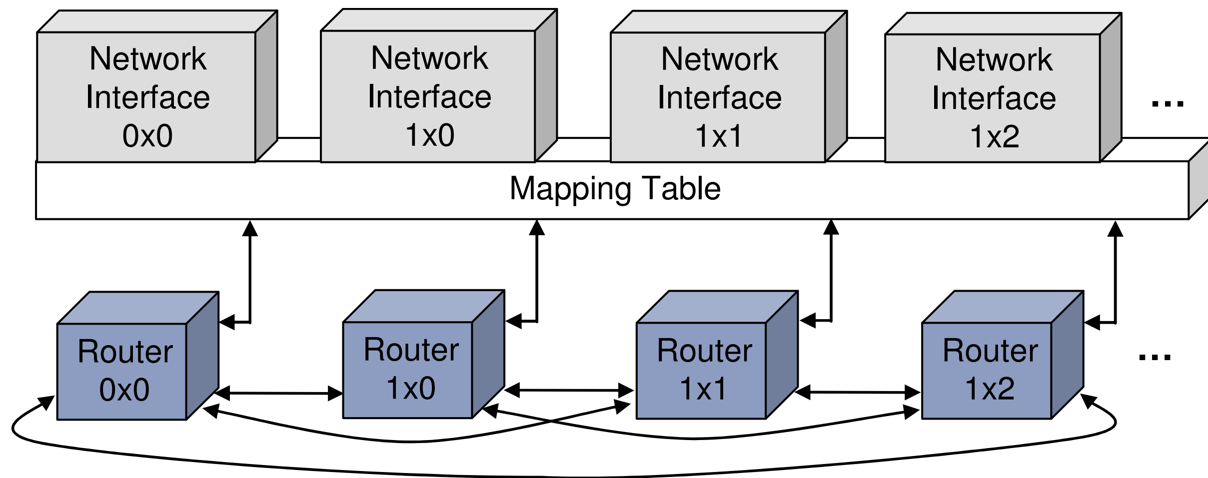


Figure 36. Abstract Hermes NoC at Virtual Architecture Level

The Hermes NoC for the Diopsis R2DT architecture involves 5 routers at the virtual architecture level. Each router is connected to the corresponding network interface and the other four routers. The network interfaces connect the following IP cores to the NoC: ARM9-SS, POT-SS, DXM, DSP1-SS and DSP2-SS. One network interface is associated to each subsystem. Therefore, SRAM and ARM9 share the same network interface with address 1x0. The local memories REG1 and DMEM1 share the network interface with address 1x1 with the DSP1 processor core. The network interface with address 1x2 connects the REG2, DMEM2 and DSP2 components to the NoC. The network interface corresponding to the DXM has address 0x0. Finally, the network interface connecting the POT-SS has address 0x1.

The functionality of the software code was validated by execution using the hardware platform. The software code was compiled with the architecture platform. During the execution, the tasks are scheduled by the SystemC simulation engine. The simulation model is accurate enough to validate the functionality of the task code and ensure that there is no communication deadlock in the scheduling of the data transfer between the tasks.

Besides the task code validation, the simulation model allowed also to gather important early performance measurements, e.g. number of words exchanged between the tasks through the network component. The virtual architecture simulation allows capturing information regarding communication values through the NoC. Such values are the amount of data exchanged between the different subsystems, the storage elements worst case size requirement for the communication buffer, the number of operations (send/receive) originated from each access point of the NoC, the amount of read/write operations performed at the storage elements and the NoC area based on the number of routers.

Figure 37 shows these numbers in case of different communication mapping schemes. Hence, when all the communication buffers are mapped on the DXM memory, as shown in figure 35, the NoC was accessed to transfer 6171680 words during the encoding process of the ten frames. In another case, *comm1* is mapped on DXM, *comm2* on REG2 and *comm3* on DMEM1. This case required 5971690 words to be transferred through the NoC. A third case maps *comm1* on DMEM1, *comm2* on DMEM2 and *comm3* on SRAM and it generates 3085840 words to be operated by the NoC.

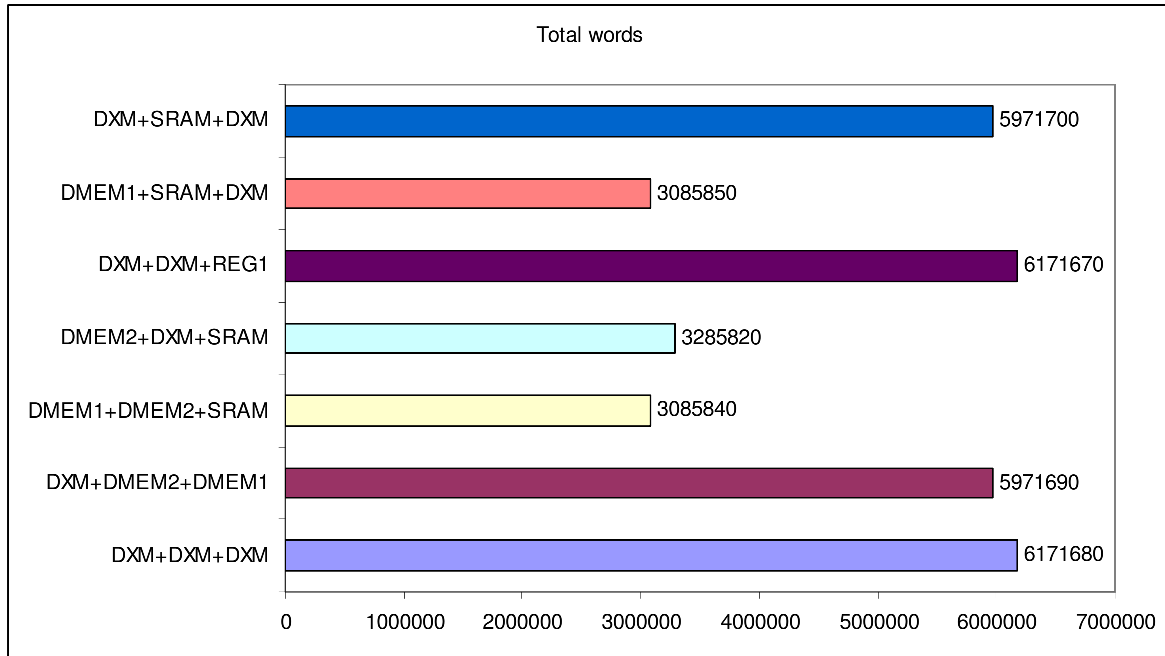


Figure 37. Words transferred through the Hermes NoC

Table 4 shows some results captured during the simulation of the H.264 Encoder application in case of the first communication scheme with all the buffers mapped on the DXM memory.

Table 4. Results captured in Hermes NoC using DXM as communication scheme

H.264	NoC Address	Read/Write Requests	Total packets sent	MBytes Sent
DXM	0x0	0	83352	17324
ARM9-SS	1x0	2426	4853	68
DSP1-SS	1x1	39260	78522	16167
DSP2-SS	1x2	41663	83327	2090

The first and the second columns represent the correspondence between the different cores connected to the NoC and the NoC addresses. The third column represents the total

number of read and writes requested over the NoC. Based on these values the designer may define a better mapping of hardware or the size of packets. The fourth and the fifth columns (Packets and MBytes sent) allow evaluating the real amount of communication injected in the NoC through each network interface. The DXM was the core that inserted the biggest amount of data in the NoC. The DXM packets are originated from read requests and confirmation packets.

In the third communication scheme, the simulation time required to encode the ten image frames using QCIF YUV 420 format was approximately 32s on a PC running Linux OS at 1.73GHz. The execution time of the encoding of the process without accurate execution delay annotation implies 2546540 ns or 127327 clock cycles with a common clock used by all the modules with the following settings: period of 20 time units, a duty cycle of 50%, the first edge will occur at 0 time units and the first value is true. All the modules are synchronized by the same clock. The default time unit is assumed to be 1 nanosecond.

3.7. State of the Art and Research Perspectives

3.7.1. State of the Art

The concept of virtual architecture is used in several academic and industrial MPSoC design environments. There are several modeling and simulation environments of the virtual architecture.

For example, in the ROSES hardware/software co-design tool, the virtual architecture is defined as a system made of an abstract netlist of virtual components. A virtual component consists of an internal component (or module) and its wrapper for adaptation to different communication protocols, abstraction levels or specification languages. The virtual components are interconnected by virtual channels through virtual ports [Ces 02].

A similar definition of the virtual architecture is given in [Shin 06]. They define the virtual architecture as a system in which processing elements communicate via abstract channels.

[Kog 01] defines the virtual architecture as an intermediate phase of the SoC design flow, where the functionality of the system is mapped to the architecture in an abstract manner to enable architecture optimization across heterogeneous computational components. The virtual architecture is annotated with timing characteristics of the target architecture, thus

it allows fast exploration of different design alternatives. In this approach, the timing related aspects are captured by the communication channels.

[Gerst 05] identifies the abstraction levels based on the communication refinement from abstract message passing down to cycle-accurate bus functional implementation. In their work, the communication design starts from a virtual architecture model. The virtual architecture is defined as a system composed of processing elements that communication via abstract channels with untimed synchronous or asynchronous message passing semantics. The virtual architecture presented in this document is similar with the one defined in [Gerst 05], but it contains also explicit mapping of the communication buffer onto the storage components and explicit abstract interconnect component.

Other research works focus on automatic generation of the virtual architecture. Thus, [Nik 06] introduces the ESPAM tool, which automatically generates C/C++ software code for each processor from an application specification in form of KPN. The code contains the main behavior of a process, together with the blocking read/write synchronization primitives and memory map of the system. The resulted code is similar with the task code at the virtual architecture presented in this document.

3.7.2. Research Perspectives

Future research perspectives of the virtual architecture concern the following aspects:

- Automatic generation of the software code
- Automatic annotation of the software and hardware code with timing information for accurate performance estimation
- Formalization of fleeting from system architecture to virtual architecture.

One of the main challenges in a SoC development flow is the consistency between different levels of abstraction of the system to be implemented [Ber 04]. The quality of design can be preserved by automatic generation of the abstraction levels, including the virtual architecture generation. The automatic generation of the virtual architecture implies generation of the software code and the hardware platform. The generation is achievable due to the annotation with architecture attributes of the initial specification in form of the system architecture.

The software and hardware architectures are natively executed on a simulation host, without using a software simulator such as Instruction Set Simulator (ISS). Therefore, to obtain an accurate estimation of the execution time for an application, such as number of cycles spent by the processor on computation or waiting for the communication, the virtual architecture code has to be orchestrated with additional timing information, like the number of cycles required by the processor to compute a function. The automatic annotation of the generated code (software and hardware) with timing information can be accomplished by inserting *wait(delay)* statements in the SystemC code of the architecture.

The passing from the system architecture level to the virtual architecture level needs to be done conform a rigorous method which ensures the right preservation of the initial specification in terms of design constraints. This may be achieved through a formalization of the system architecture, virtual architecture and the formalization of the conversion from the high level to the more detailed level. The considered aspects could be the model of computation and the model of execution that characterize each abstraction level, and the definition of the rules that guarantee a correct translation from one model to another.

3.8. Conclusions

This chapter defined the virtual architecture design. It presented the software representation as final application task code and the hardware organization in abstract subsystems interconnected through an abstract network component.

The virtual architecture design was performed using SystemC for 3 case studies: Token Ring mapped on the 1AX architecture, Motion JPEG running on the Diopsis RDT architecture and H.264 Encoder running on the Diopsis R2DT architecture.

The simulation of the virtual architecture model allowed to validate the final code of the application tasks and the partitioning of the application. It also gave important statistics regarding the communication requirements. These include the total number of bytes exchanged between the subsystems during the execution of the application, the amount of data passing through the interconnect component (bus, NoC) and the buffer size requirements in the worst case scenario for the storage resources in order to support the communication mapping.

Chapter 4

TRANSACTION ACCURATE ARCHITECTURE DESIGN

This chapter details the transaction accurate architecture design. The transaction accurate architecture design consists of integrating the OS and the communication software component with the application task code and adapting the software to specific communication synchronization protocol. The key contribution in this chapter represents the transaction accurate architecture definition, organization and design, using SystemC, for the Token Ring application running on the 1AX architecture, the Motion JPEG application targeting the Diopsis RDT architecture and the H.264 Encoder running on the Diopsis R2DT architecture. The simulation of the transaction accurate architecture model allows validating the execution of the application tasks code upon an OS and early performance validation of the communication mapping scheme. Different interconnect components, communication mapping schemes and IP cores positioning over the interconnect component are explored in order to analyze the performances of the various communication paths.

4.1. Introduction

The transaction accurate architecture design consists of software adaptation to specific communication protocol implementation. At this phase, aspects related to the communication protocol are detailed, for example the synchronization mechanism between the different processors running in parallel becomes explicit. The software code is adapted to the synchronization method, such as events or semaphores. The adaptation is performed through an integration of the tasks codes with the OS and communication components of the software stack. The result of the transaction accurate architecture design represents the transaction accurate architecture model.

4.1.1. Definition of the Transaction Accurate Architecture

The third abstraction level of the hardware-software architecture is called transaction accurate architecture level (TA). The transaction accurate architecture details the local architecture of each subsystem and makes explicit the communication protocol. On the software side, the tasks code is integrated with an operating system and communication library to form the software stack. Each processor subsystem executes a software stack. The transaction accurate architecture model may be manually coded or automatically generated by different tools.

The objectives of the transaction accurate architecture are:

- Early validation of the tasks code execution upon an operating system
- Early performance validation of the communication mapping scheme.

The transaction accurate architecture is composed of processor and hardware subsystems that are interconnected using an explicit interconnection component, such as bus or NoC. The processor subsystems include the local components of the subsystem, such as local memories, peripherals and network interfaces, and an abstract model of the processor cores.

Figure 38 illustrates a global view of the transaction accurate architecture, composed of two abstract processor subsystems, one memory hardware subsystem and the network component. The left part of the figure corresponds to the hardware architecture, while the

right part represents the software stack at the transaction accurate architecture level running on the one of the processor subsystems.

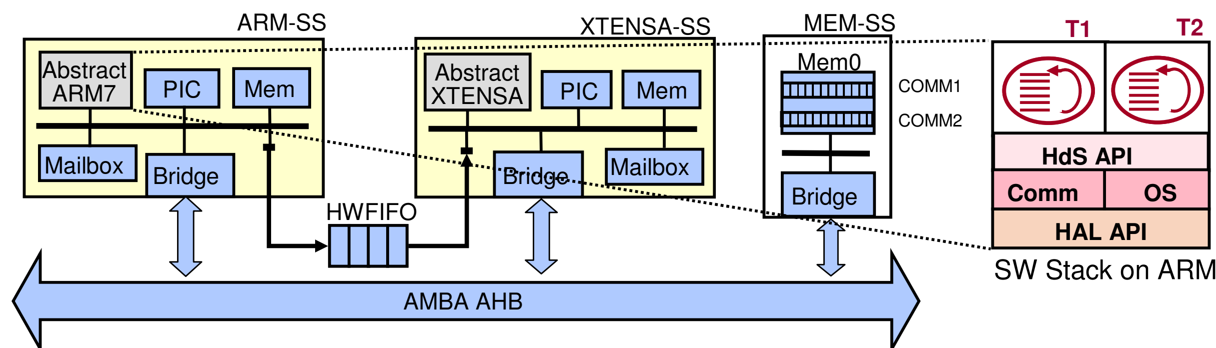


Figure 38. Global View of the Transaction Accurate Architecture

4.1.2. Global Organization of the Transaction Accurate Architecture

The transaction accurate architecture model is a hierarchical model. The transaction accurate architecture is composed of software and hardware subsystems that are interconnected using an explicit network component, e.g. bus, NoC or dedicated hardware components like the hardware FIFO.

The software subsystem represents the processor subsystem. The hardware subsystem represents a memory subsystem or dedicated hardware subsystem that accelerates the computation of specific application functions.

Each subsystem integrates local components that are interconnected using a local simple bus. Usually the processor subsystems are made of one or more abstract computation models of the processor cores, local memories such as program code memory, data memory or dedicated registers, network interfaces for the connection with the external world and other processor specific peripherals. The selection of these components relies on the target architecture and the software requirements at this level.

Each abstract processor model executes a specific software stack made of the tasks code, operating system and communication library. The software stack uses hardware abstraction layer primitives (HAL APIs) for the interaction with the hardware part of the system. In fact, the abstract processor with the implementation of the HAL APIs represents the hardware-software interface.

At the transaction accurate architecture level, the intra-subsystem communication units become communication channels implemented by the communication and operating system components of the software stack. Therefore, the communication between the tasks running on the same processor is managed totally by the OS and communication software libraries.

The inter-subsystem communication units are mapped on full end-to-end communication paths through the architecture. Hence, the communication protocol and synchronization between the processors become explicit. The different communication paths are characterized by different performance indicators, such as throughput of the buses, delay of the communication path or overhead of the HdS layer (device drivers, resource sharing mechanism).

The adopted communication path and the topology of the network infrastructure are implemented according to the annotation of the system architecture model and performance estimation through the simulation of the virtual architecture model.

Example 23. Transaction Accurate Architecture for the Token Ring application

Figure 38 shows a conceptual representation of the transaction accurate architecture for the Token Ring application mapped on the 1AX architecture.

Figure 38 illustrates that for the Token Ring application running on the 1AX architecture, the transaction accurate architecture contains two processor subsystems, corresponding to the ARM, respectively XTENSA processors and the global memory subsystem. All these subsystems are interconnected by an explicit AMBA bus.

The ARM-SS processor subsystem includes an abstract ARM module, local memory, programmable interrupt controller (PIC), mailbox for the communication synchronization and bridge for the interface to the AMBA bus, all interconnected through a local bus.

The local architecture of the XTENSA-SS subsystem is similar with the ARM-SS subsystem, but only it includes abstract model for the XTENSA processor instead of the ARM7 processor. The global memory subsystem includes the global memory and the bridge for the connection with the global bus.

The communication through a FIFO between the tasks T1 and T2 mapped on the ARM-SS is implemented by the software components of the ARM software stack.

At the transaction accurate architecture level, the inter-subsystem communication units COMM1 and COMM2 are mapped on full communication path. Therefore, a data sent by the ARM and received by the XTENSA processor using as storage buffer the global memory follows the next data path:

ARM -> BUS_ARMSS -> BRIDGE_ARMSS -> AMBA -> BRIDGE_MEMSS -> BUS_MEMSS -> MEM -> BUS_MEMSS -> BRIGDE_MEMSS -> AMBA -> BRIDGE_XTENSASS -> BUS_XTENSASS -> XTENSA

where:

- *BUS_ARMSS* represents the local bus of the ARM-SS
- *BRIDGE_ARMSS* is the bridge of the ARM-SS
- *BUS_MEMSS* is the local bus of the MEM-SS
- *BRIDGE_MEMSS* is the interface of the global memory to the AMBA bus
- *BUS_XTENSASS* specifies the local bus of the XTENSA-SS
- *BRIDGE_XTENSASS* represents the bridge inside the XTENSA-SS.

This kind of data transfer requires synchronization mechanism between the two processors using the mailbox components. Thus, when the data to transmit is stored in the global memory, the ARM sends an event to the mailbox of the XTENSA to notify that there is available data. After checking the appropriate register status of the mailbox, the XTENSA processor may transfer the data from the global memory.

Other path of communication between the processors offered by the architecture involves the following route:

ARM -> BUS_ARMSS -> HWFIFO -> BUS_XTENSASS -> XTENSA

The communication through the hardware FIFO does not require explicit synchronization because the hardware resource manages also the synchronization between the processors.

The transaction accurate architecture model may be represented using different design languages, such as SystemC [Gro 02] or SpecC [Gaj 00]. The following paragraphs will present the transaction accurate architecture using SystemC as design language.

4.2. Basic Components of the Transaction Accurate Architecture Model

The basic components of the transaction accurate architecture Model are the software and hardware components. The software components consist of the tasks code, operating system, communication library and HAL APIs, while the hardware components represent detailed subsystems and explicit communication network.

4.2.1. Software Components

At the transaction accurate architecture level, a software stack is build for each processor subsystem. This software stack is composed of the previously generated tasks code enriched with an OS and communication library. The HdS software represents the assembly of OS, communication library and HAL APIs. The HdS refines the communication APIs (HdS APIs) to custom hardware specific low level APIs (HAL APIs) and is responsible for task and hardware resources management. The HAL APIs abstract the underlying hardware architecture. Their implementation is not yet defined for the target processor, allowing keeping the software code still processor independent. Based on OS and communication libraries, the proposed approach sets aside flexible building and configuration of the software stack. Therefore, it allows easy customization for specific architectures and/or applications. At this level, the data transfers use explicit addresses, e.g. *read_mem(addr, dst, size)/ write_mem(addr, src, size).*

Example 24. Software Components for the Token Ring application at the Transaction Accurate Architecture Level

For the Token Ring application, a software stack is executed by each processor (ARM7 and XTENSA). The software stack running on the ARM7 is made of 2 application tasks code (T1 and T2), OS and communication library. The software stack running on the XTENSA is made of task code of T3, OS for the interrupt management and communication software component. For both processors, the software stack has the same OS running, namely DwarfOS the same

communication library, that implements the primitives *send_data(…)/ recv_data(…)* and are based on the same HAL APIs (*read_mem(…)/ write_mem(…),ctx_swich(…)*).

4.2.2. Hardware Components

The hardware architecture at the transaction accurate level represents a more detailed platform than the virtual architecture level. It includes the components explicitly used by the HAL APIs. The different subsystems of the architecture are detailed with explicit peripherals and abstract computation model for the processor cores. Design decisions such as subsystems positioning over the global interconnect component, NoC size definition, NoC topology, NoC routing algorithm and communication buffer size are implemented at the transaction accurate architecture level.

Example 25. Hardware Components for the Token Ring application at the Transaction Accurate Architecture Level

For the Token Ring application, the hardware platform has a detailed local architecture for each subsystem. Thus, the ARM-SS and XTENSA-SS contain an abstract ARM, respectively XTENSA processor, a local memory, an interrupt controller, a local bus and a bridge for the interface with the AMBA. The global memory subsystem contains the global memory and the bridge for the connection to the AMBA. The hardware FIFO is connected directly to the local bus of each processor subsystem.

4.3. Modeling Transaction Accurate Architecture in SystemC

The transaction accurate architecture model is described using SystemC TLM language and is designed according to the annotated architecture parameters of the initial system architecture model and the results of the virtual architecture model simulation.

4.3.1. Software at Transaction Accurate Architecture Level

The software design at the transaction accurate architecture level consists of integration of the tasks code with an OS and communication implementation for each processor subsystem. In the following examples, the considered operating system is called DwarfOS, an in-house tiny operating system which supports a set of basic services, such as interrupts management, FIFO software communication protocol, a cooperative scheduling policy based on static priority and application tasks initialization [Gue 07] [Pop 08]. The communication primitives are based on blocking message passing interface semantic. The synchronization is made using events. At this level, the generated tasks are dynamically scheduled by the OS scheduler according to the availability of data for read operations or the availability of space for write operations.

The tasks C code remains unchanged from the virtual architecture level and it uses HdS APIs such as *send_data(…)/recv_data(…)*. Compared with the virtual architecture, the implementation of these APIs is not anymore handled by the SystemC architecture. The implementation relies on the OS and communication libraries. Hence, the tasks are blocked on communication and scheduled by the OS scheduler and not by the SystemC scheduler as at virtual architecture level.

The OS and communication components make use of HAL APIs. At this level, the implementation of the HAL APIs is not yet defined for the target processors. Therefore, the software code is still processor independent at the transaction accurate architecture level, but it is adapted to specific hardware communication implementation such as synchronization. The HAL APIs i.e. *__ctx_switch(…)* gives to the operating system, communication and application software an abstraction of the underlying architecture. Furthermore, the HAL APIs ease OS porting on new hardware architecture. There are different categories of HAL APIs [Yoo 03]:

- Kernel HAL APIs, such as context creation, delete or switch APIs
- Interrupt management APIs which enable/disable interrupt and implements the interrupt routine services
- I/O HAL APIs, which configure the I/O devices and allows their access
- Resource management APIs for power management (check battery status, set CPU clock frequency, set or reset timer)

Example 26. Software Code for the Token Ring application at the Transaction Accurate Architecture Level

Figure 39 illustrates an example of software code for the Token Ring at the transaction accurate architecture level.

```
#include <config.h>                     // OS dependent header files
#include <support/os_types.h>
#include <comm/os_comm.h>
#include <comm/event.h>
#include <stdio.h>

extern void Task1( );
extern void Task2( );

unsigned char SWFIFO_buf1[4];                   // software channels
unsigned char SWFIFO_stat_send1 = OS_EVENT;
unsigned char SWFIFO_stat_recv1 = OS_NO_EVENT;

void thread_main( ) {
   int id;

   vector_attach(UNIX_IRQ, 0, _mailbox_isr, NULL);
   vector_enable(UNIX_IRQ);

   id=thread_create(Task1,0);                // tasks initialization
   id=thread_create(Task2,0);                // for scheduling
   return;
}
```

Figure 39. Initialization of the Tasks running on ARM7

Figure 39 shows the main file. The main file contains the function "thread_main" which represents the first function executed on the processor after boot. The main file is responsible to initialize the application tasks and software communication channels. It includes the OS dependent header files, it declares the software FIFO communication channels, it attaches the interrupt routine services to the interrupt numbers and initializes the tasks in the list of scheduling tasks for the operating system. As illustrated in figure 39, in case of the Token Ring application, the initialization file of the ARM7 processor declares the two tasks running on ARM7 and the software FIFO used for the communication between them. It also attaches the interrupt routine service of the mailbox to interrupt number 0.

Figure 40 shows a fragment of the code implementing the communication primitive *recv_data(…)*. If the protocol of the

communication channel is based on a FIFO mechanism, the implementation checks the status of the FIFO. If the FIFO is empty, the scheduler of OS is called (*__schedule(…)*).

```
void recv_data (ch,dst,size){      //implementation of recv_data HdS API
…
  switch (ch.protocol){
          case FIFO:
            if  (ch.state == EMPTY)
                    __schedule();  // OS scheduler
…
```

Figure 40. Implementation of *recv_data(…)* API

The communication primitives access the logic ports of the tasks that are declared in the header files of each task. Figure 41 shows the header file of task T2 running on the ARM7 processor in case of the Token Ring application.

Task T2 has two logic ports:

- One input port (*In1_Task2*) bonded to the software FIFO channel, that connects task T1 and T2 and it was declared in the main file of the ARM7 processor as pointed up in figure 39.
- One output port (*Out1_Task2*) for the external communication with the task T3 running on the XTENSA processor.

The logic ports are declared of type *port_t,* as illustrated in figure 41. The *port_t* represents the data structure which implements the logic port in case of the DwarfOS. It combines the following fields: communication protocol associated to the port, status of the local synchronization register, status of the remote synchronization register, destination buffer used to store the data to be exchanged, list of tasks that are waiting for the port to acquire a synchronization event, and a specific field which stores special protocol characteristics.

The input port of task T2 is characterized by a software FIFO protocol and has the synchronization and buffer associated with the software FIFO channel. The output port of task T2 notes a global FIFO protocol with the communication buffer mapped onto the external memory at the address 0x40500000 and the synchronization making use of the registers of the local and remote mailbox corresponding to the communication channel. The local mailbox represents the mailbox corresponding to the ARM

processor accessed at address 0x300808. The remote mailbox stands for the mailbox of the XTENSA-SS with address 0x700808.

```
#ifndef _Task2_H
#define _Task2_H

#include <support/os_types.h>
#include <comm/os_comm.h>
#include <comm/event.h>
#include <stdio.h>

extern unsigned char SWFIFO_buf1[4];          //software fifo channel
extern unsigned char SWFIFO_stat_send1;       //status of sender
extern unsigned char SWFIFO_stat_recv1;       // status of receiver

extern port_t Out1_Task1;

port_t In1_Task2 = {OS_SWFIFO_PROTOCOL,       // SOFTWARE FIFO protocol
                    &SWFIFO_stat_recv1,       // local synchronization
                    &Out1_Task1,              // remote port
                    SWFIFO_buf1,              // buffer address
                    NULL,
                    OS_DEFAULT};

port_t Out1_Task2 = {OS_GFIFO_PROTOCOL,       // GLOBAL FIFO protocol
                    (void*)0x300808,          // mailbox local register
                    (void*)0x700808,          // mailbox remote register
                    (void*)0x40500000,        // buffer address
                    NULL,
                    OS_DEFAULT};
```

Figure 41. Example of Task Header File

```
typedef struct {
          protocol_t protocol;
          void *l_status;
          void *r_port;
          void *d_buffer;

          thread_t *requesting_thread;
          unsigned char specific;
} port_t;
```

Figure 42. Data Structure of Tasks'Ports

Figure 43 shows a portion of the OS scheduler implementation. The scheduler searches for a new task in status ready for execution. If there is a new ready task, the scheduler performs a context switch, by calling the HAL API *__cxt_switch(…)*. During the context switch, the OS saves the status and registers (program counter, stack pointer, etc) of the processor running the current task and loads those of the new task.

```
void __schedule (void){
   int old_tid = cur_tid;
   cur_tid = get_new_tid();         //get new task ready for execution

   __ctx_switch (old_tid,cur_tid);  //context switch HAL API
  …
```

Figure 43. Implementation of the *__schedule()* Service of OS

4.3.2. Hardware at Transaction Accurate Architecture Level

The hardware at the transaction accurate architecture level consists of the set of hardware and software subsystems interconnected using an explicit communication network. The hardware architecture implements the communication protocol, including buffer mapping, synchronization mechanism used by the processors and the entire communication path for inter-subsystem communication.

The different subsystems represent SystemC modules (SC_MODULE) which include the local components. A top module includes the declaration, instantiation, interconnection and address space allocation of these subsystems. Each subsystem incorporates the local hardware modules. The local components are also SystemC modules.

The transaction accurate architecture makes use of library of transaction accurate components. This library implements parametric hardware components such as mailbox, bridge, network interface, interrupt controller, interrupt signals, buses and abstract execution model for distinct types of processor.

Example 27. Hardware Code for the Token Ring application at the Transaction Accurate Architecture Level

Figure 44 details the Top module for the Token Ring application running on the 1AX architecture.

```
#include "XTENSA_SS.h"
#include "ARM7_SS.h"
#include "AMBA.h"
#include "GMEM_SS.h"

SC_MODULE (TOP)
{
 public:
  AMBA *vAMBA;
  GMEM_SS *vgmem_ss;
  XTENSA_SS *vxtensa_ss;
  ARM7_SS *varm7_ss;

  SC_CTOR(TOP)
  {                                                       //AMBA BUS
    vAMBA = new AMBA("AMBA");

                                                          //MEMORY SUBSYSTEM
    vgmem_ss = new GMEM_SS("gmem",0x1000000);

    vgmem_ss->bridge->port(*vAMBA);
    vgmem_ss->bridge->port.set_map(0x40000000,0x40FFFFFF);

                                                          //XTENSA SUBSYSTEM
    vxtensa_ss = new XTENSA_SS("XTENSA_SS","../sw/XTENSA/XTENSA.bin");
    vxtensa_ss->bridge->port(*vAMBA);
    vxtensa_ss->bridge->port.set_map(0x400000,0x7FFFFF);

                                                         //ARM7 SUBSYSTEM
    varm7_ss = new ARM7_SS("ARM7_SS","../sw/ARM7/ARM7.bin");
    varm7_ss->bridge->port(*vAMBA);
    varm7_ss->bridge->port.set_map(0x800000,0xBFFFFF);
  }
};
```

Figure 44. SystemC Code for the Top Module

The top module is a SC_MODULE which includes the declaration and the instanciation of the ARM-SS (*varm7-ss* in figure 44), XTENSA-SS (*vxtensa_ss*), AMBA bus (*vAMBA*) and global memory subsystem MEM-SS (*vgmem_ss*). It also interconnects these different subsystems by linking the bridges of each subsystem to the AMBA bus. A 4Mbytes address space is allocated to each processor subsystem. Thus, the ARM-SS has the address space 0x800000-0xBFFFFF and the XTENSA-SS has the address space 0x400000-0x7FFFFF. The global memory is identified between addresses 0x40000000-0x40FFFFFF.

Figure 45 shows the SystemC module of the ARM7 subsystem of the 1AX architecture.

```
#include "ARM7_SS.h"
extern int debug_flag;

ARM7_SS::ARM7_SS(sc_module_name name, char *bin)          // ARM7-SS
:sc_module(name)
{
    sys_bus = new TlmBus("sys_bus");                      // local bus

    core = new ArmUnixCore("ARM7Core",bin,debug_flag);    // abstract ARM7 core
    core->rw_port(*sys_bus);

    mem = new Sram("mem0",0x300000);                      // local memory
    mem->port(*sys_bus);
    mem->port.set_map(0x0,0x2FFFFF);

    bridge = new AhbIf("bridge");                         // bridge
    bridge->master(*sys_bus);
    bridge->slave(*sys_bus);
    bridge->slave.set_map(0x400000,0x7fffffff);

    pic = new Pic<1>("pic",0x20);                         // PIC
    pic->port(*sys_bus);
    pic->port.set_map(0x300000,0x30001f);

    sync = new Sync("sync",0x400);                        // mailbox
    sync->port(*sys_bus);
    sync->port.set_map(0x300800,0x300bff);

    TlmIntrSig *sig_sync = new TlmIntrSig("sig_sync");   //interrupt signals
    sync->intr(*sig_sync);
    pic->in_irq[0](*sig_sync);
    s1 = new TlmIntrSig("sig_intr1");
    s2 = new TlmIntrSig("sig_intr2");
    pic->out_fiq(*s1);
    pic->out_irq(*s2);
    core->nIrqPort(*s2);
}
```

Figure 45. SystemC Code for the ARM7-SS Module

The ARM7 subsystem includes a local bus (*sys_bus*), an abstract execution model of the processor core (*ArmUnixCore*), a local memory (*mem*), a bridge (*bridge*) for the connection to the AMBA bus, a programmable interrupt controller (PIC) (*pic*), the mailbox synchronization component (*sync*) and some interrupt signals (*sign_sync*, *s1* and *s2*) . The local peripherals have associated address space. Thus, the local memory is addressable between addresses 0x0-0x2FFFFF, the PIC between addresses 0x300000-0x30001F, the mailbox between addresses 0x300800-0x300BFF. Each processor subsystem has the local address space between 0x0-0x400000. The accesses to addresses higher than 0x400000 will be forwarded by the local bus to the bridge for external access through the AMBA bus.

```
As illustrated in figure 46, the transaction accurate
architecture of the 1AX architecture contains a global clock used
by all the processors. This clock has a period of time 1 unit,
where a time unit represents one nanosecond.
```

```
sc_clock SystemClock("SystemClock", 1, SC_NS);    //SYSTEM SYSTEMC CLOCK
```

Figure 46. SystemC Clock

4.3.3. Hardware-Software Interface at Transaction Accurate Architecture Level

The hardware-software interface at the transaction accurate architecture level is represented by the abstract model of each processor core and the implementation of the HAL APIs. This is responsible to guarantee the software access to the hardware and implements the interaction between hardware and software.

The abstract model of the processor defines an execution environment of the software stack [Schir 07]. This is implemented as a SystemC module which interacts with the software. The abstract processor is modeled as a bus functional model, which allows operations onto the local bus, such as read and write operations [Shin 04].

The implementation of the HAL APIs allows a simulation model of the OS and inter-processor communication on the host machine [Bac 05]. For example, the implementation of the HAL API *ctx_switch (old_tid, cur_tid)* to perform a context switch between two tasks relies on the APIs provided by the operating system running on the host machine (Windows, Linux, UNIX, etc). Figure 47 exemplifies the implementation of the context switch on the host machine running Linux OS that uses *sigsetjmp* and *siglongjmp* APIs to save and switch the context of a task.

```
void __ctx_switch(int old_tid, int new_tid)
{
          sigjmp_buf old_buf, new_buf;

          old_buf = task[old_tid].buf;
          new_buf = task[new_tid].buf;

          if(!sigsetjmp(old_buf, 1))                        //LINUX APIs
                              siglongjmp(new_buf, 1);
}
```

Figure 47. Implementation of the *__ctx_switch* HAL API

4.4. Execution Model of the Transaction Accurate Architecture

The full hardware-software executable model is based on a co-simulation between SystemC for the hardware components including the abstract processors, and the native execution of the software stacks [Nic 02].

Each software stack is a SystemC thread which creates a Linux process for the software execution. At the beginning of the simulation, the SystemC platform launches a GNU standard debugger (gdb) Linux process for each software stack in order to start its execution. The software stack interacts with the corresponding SystemC abstract processor module through the Linux IPC layer. The hardware-software interface uses Linux shared memory (IPC Linux *shm*) for the interaction, data and synchronization exchange between the software and the hardware.

The simulation at the transaction accurate architecture level allows validating the integration of the tasks code with the OS and the communication protocol and debug of the HdS access to the hardware resources (e.g. access to the AMBA bus, interrupt lines assignment, OS scheduling, etc). On the software side, it makes possible the debug of the access of the OS functions to the hardware resources through the HAL APIs, e.g. *read(…)/ write(…)* in the memory, explicit synchronization using mailboxes or the interrupt routine services. On the hardware side, it gives more precise statistics on the communication and computation performances, such as number of exchanged bytes during the application execution, network congestion or estimation of the processors cycles spent on communication.

Example 28. Execution Model for the Token Ring application at the Transaction Accurate Architecture Level

Figure 48 shows the execution model of the software stacks running on the ARM7 and XTENSA processors in case of the 1AX architecture. This represents a co-simulation between the gdb Linux processes of each software stack *gdb1* and *gdb2* (one gdb for each software stack) and one SystemC Linux process for the whole hardware platform simulation. The interface between the three Linux processes is performed using the Linux IPC shared memory.

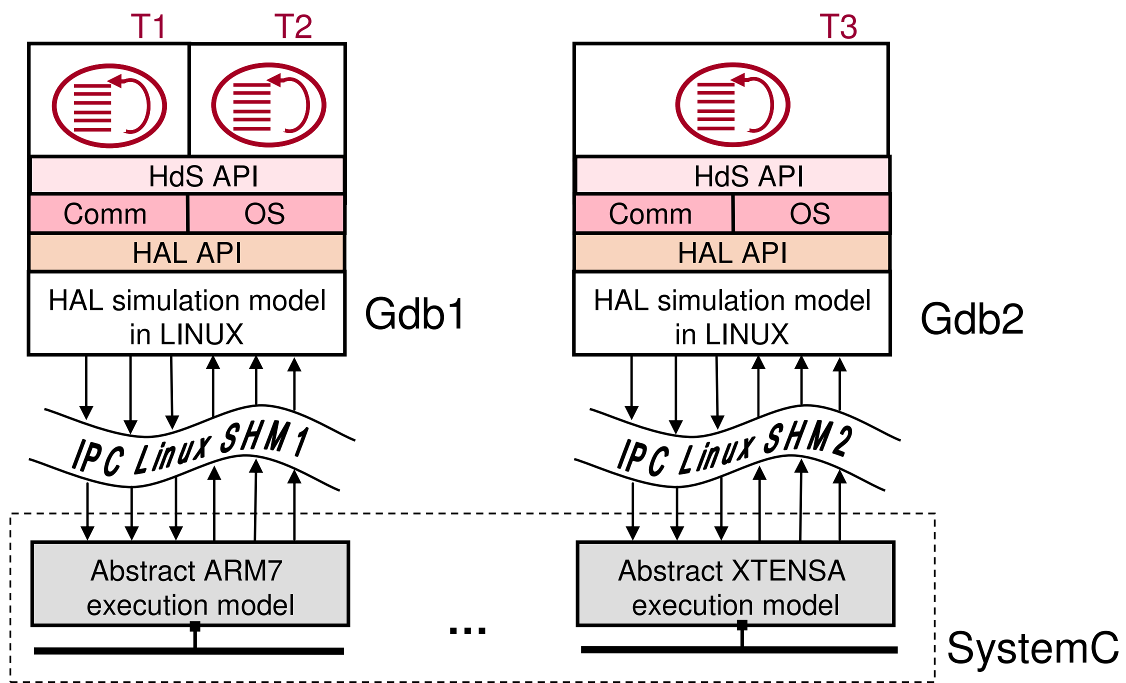


Figure 48. Execution Model of the Software Stacks running on the ARM7 and XTENSA Processors

4.5. Design Space Exploration of Transaction Accurate Architecture

4.5.1. Goal of Performance Evaluation

The goal of performance evaluation at the transaction accurate architecture level is to allow profiling the communication requirements and improve the overall performances of the system. The objective is to provide through simulation statistical information, such as utilization of the global interconnect component or degree of contention in the network component and validate the communication protocol and the execution of the tasks under the control of a dedicated operating system

Based on the communication traffic resulted after the transaction accurate architecture simulation, the designer can fix hardware and software architecture decisions. Examples of hardware architecture decisions are: the entire end-to-end communication path used for data exchange between the processors, the size of the NoC in number of routers, the positioning of the IP cores over the NoC, the final topology of the interconnect component, the routing algorithm used in a NoC, the buffer size inside the NoC routers or the communication

protocol between the different subsystems fixing the mapping of the communication buffers onto the storage resources and the synchronization mechanism. Examples of software architecture decisions are: operating system used for the scheduling of the tasks running on the same processing units, implementation of the communication primitives and synchronization mechanism managed by software.

These different decisions influence the overall execution time of the system, cost and power consumption. Therefore, good decisions are required to be able to control the MPSoC design process.

4.5.2. Architecture/Application Parameters

The transaction accurate architecture validates some hardware and software architecture characteristic specified at the system architecture level, such as:

- Integration of the tasks code with the OS and communication libraries
- Implementation of the communication protocol: buffers mapping, synchronization mechanism and end-to-end data path between the processors
- Adaptation of the software to specific hardware communication implementation
- Scheduling algorithm of the tasks
- Type of global interconnection algorithm with its configuration parameters such a as topology, buffer size, routing algorithm, arbitration algorithm.

The transaction accurate architecture still keeps the implementation of the communication protocol independent of the type of processor cores. Therefore, the *CPUCoreType* represents an architecture parameter that will be considered only at the next abstraction level, the virtual prototype level. This will determine the adaptation of the software to particular CPU through the explicit implementation of the low level processor specific HAL software layer.

4.5.3. Performance Measurements

At the transaction accurate architecture level, the performance measurement consists of profiling the interconnect component and the communication and computation requirements for each processor.

Using annotation of the transaction accurate architecture model with adequate execution delays, the simulation at this level can estimate the total clock cycles spent on communication or computation by each processor. The achieved precision can be cycle accurate only for the inter-subsystem communication, since all the hardware components of the communication path are explicit. The accuracy of the software execution is transaction level.

On the hardware side, the transaction accurate architecture may give more precise statistics on the communication architecture such as number of conflicts on the shared global bus due to the simultaneous access requests in the case of a bus-based architecture topology. For a NoC based architecture topology, useful information deduced during the simulation are related to the amount of NoC congestion, number of routing requests, number of transmitted packets, the average amount of transmitted bytes per packet or the number of times some routers failed to transmit the packed due to the conflicts. For both topologies (bus and NoC), the transaction accurate architecture simulation allows extracting the total amount of transmitted bytes through the global interconnect component and the amount of data transferred between the different processors.

Example 29. Performance Measurements for the Token Ring application at the Transaction Accurate Architecture Level

For example, the total simulation time of the Token Ring application was 12 seconds to run the whole application and the bus was required 108 times to transfer data. But in this example, the model is not annotated with accurate information required for an accurate estimation due to operating system and communication overhead.

4.5.4. Design Space Exploration

At the transaction accurate architecture level, the design space exploration consists of communication mapping exploration. The designer can experiment different communication mapping schemes, different communication protocols and diverse global interconnect components in distinct configurations. For example, the designer may adopt a bus such as STBus or AMBA bus or a NoC such as Hermes or STNoC. Moreover, the NoC may support

different topologies (mesh, torus, hypercube, ring, tree), the routers may be positioned in different dimensions (2D, 3D), the number of routers is configurable, and the IP cores may be located through different access points to the NoC. Thus, the NoC offers flexibility and scalability in terms of number of routers, number of network interfaces and interconnected IP cores.

Example 30. Design Space Exploration for the Token Ring application at the Transaction Accurate Architecture Level

At this level, the designer can still map the communication buffers onto different storage resources provided by the architecture, such the local memories of both ARM and XTENSA processors, or the shared global memory or on the hardware FIFO in case of the 1AX architecture running the Token Ring application. These different communication mapping schemes involve different communication paths and synchronization mechanisms between the processors.

4.6. Application Examples at the Transaction Accurate Architecture Level

The following paragraph presents the transaction accurate architecture model for the two case-studies: the Motion JPEG Decoder application running on the Diopsis RDT architecture with AMBA bus and the H.264 Encoder application running on the Diopsis R2DT architecture with Hermes NoC in Torus and Mesh topologies.

4.6.1. Motion JPEG Application on Diopsis RDT

The transaction accurate architecture design consists of two steps: software and hardware design. The software design consists of linking the tasks code with an operating system and communication library. For the Motion JPEG application, in order to produce an executable software code, the tasks code is compiled with the DwarfOS operating system and the communication library that implements the *send_data(…)/recv_data(…)* communication primitives. The tasks are schedules by the OS. The communication between the tasks of the same processor is implemented by the OS and communication library.

The hardware architecture of the Diopsis RDT tile contains the components that can be accessed by HAL APIs (figure 49). The ARM subsystem includes the abstract processor core, local data memory (SRAM), local bus and bridge for the connection with the AMBA bus. The DSP subsystem includes the DSP core, data memory (DMEM), registers (REG), DMA, interrupt controller (PIC), mailbox, local bus and the bridge for external connection. The POT includes the system peripherals of the RISC processor, e.g. timer, interrupts controller (AIC), synchronization component (mailbox), but also I/O components like the serial peripheral interface (SPI).

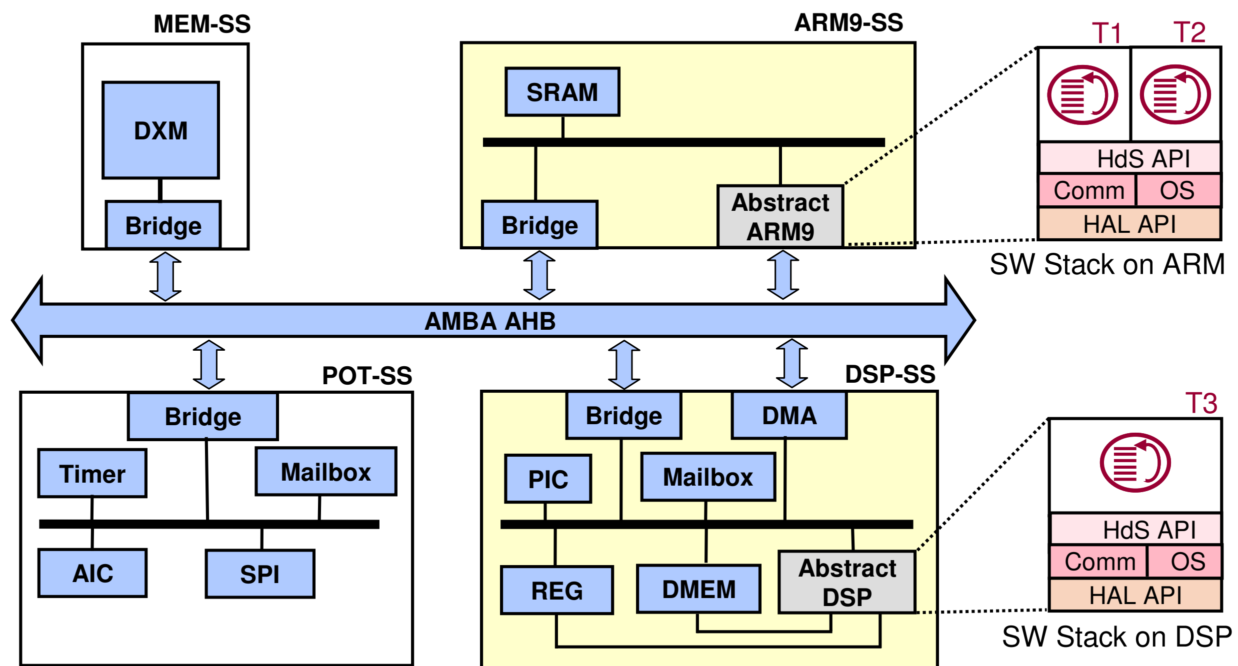


Figure 49. Transaction Accurate Architecture Model of the Diopsis RDT Architecture running Motion JPEG Decoder Application

The AMBA bus implementation is based on the implementation at the virtual architecture level, but the synchronization between the different subsystems connected to the global bus is handled explicitly through the operating system and dedicated hardware components. The AMBA supports burst mode transfer at this level.

The assignment of addresses and mapping of the communication buffers into the memories with the corresponding interrupt mechanism used for synchronization is performed during the hardware platform design. The address space of components is different from the virtual architecture platform, because the generated platform at Transaction Accurate level is more detailed and fully implements the communication protocol.

The full hardware-software executable model is based on a cosimulation between SystemC for the hardware components including the abstract processors, and native execution of the software stacks. Each software stack is a UNIX process created and launched at the beginning of the simulation by the SystemC platform, in order to start their execution. The software stack interacts with the corresponding SystemC abstract processor module through the Unix IPC layer. Besides the software debug, the execution model at this level also provided more precise idea on performances, that allowed some architecture experimentation, as detailed in the next section. The simulation of the 10 QVGA frames at transaction accurate level takes 5m10s. Figure 50 shows a screenshot taken during the simulation, which captures the execution of the 2 software stacks running on the ARM, respectively DSP, and the SystemC simulation of the platform with the POT displaying the decoded image.

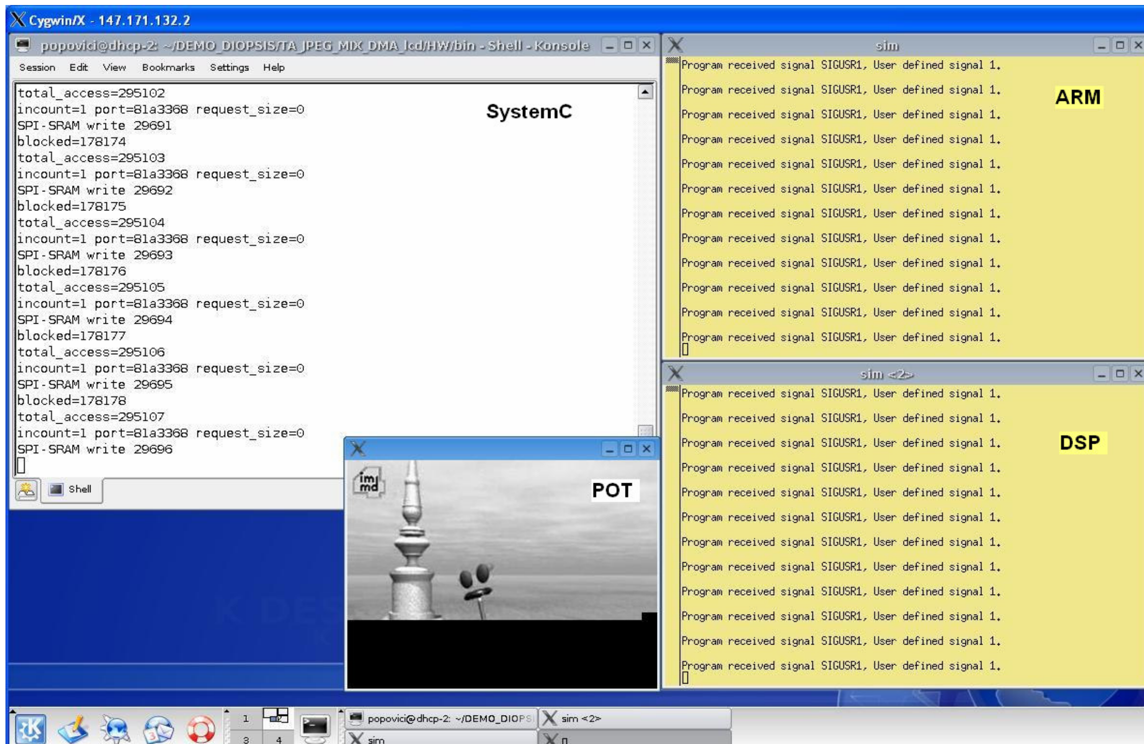


Figure 50. MJPEG Simulation Screenshot

Using transaction accurate simulation, in this document, three experiments are conducted with different communication schemes between the DSP and RISC. The results are summarized in table 5. In the first scheme, the data exchange is made only via DXM. This generated 5256000 transactions to DXM. The second communication scheme makes use of DXM and REG communication units between the processors and DMEM between the DSP and the POT. This generated 4608000 transactions to the DXM, 72000 to the register

and 576000 to the DMEM. The third case uses the SRAM as communication unit between the processors and DMEM between the DSP and POT and needs 4680000 transactions to the SRAM and 576000 to the DMEM. One transaction to the memory means one read/write operation of 1 word (4 bytes) to the memory.

Starting from quantitative estimators provided by ATMEL Inc., the number of clock cycles, needed by ARM and DSP to access data buffers of length N words located in different memories, can be estimated. The DMA engine of the DSP needs 14+(N-1) cycles for DXM read, 10+(N-1) for DXM write, 5+(N-1) for SRAM read and 8+(N-1) for SRAM write. A data movement between REG and SRAM driven by the DSP core costs N/4 cycles plus a movement to/from the SRAM driven by the DMA engine. The ARM processor is not natively equipped with a DMA engine. The cost of ARM isolated access is 11*N for DXM read and 8*N for DXM write. Forcing the compiler to use the assembler instruction which moves blocks of 8 registers, the cost of burst can be reduced to 11*(N/8)+N for DXM read and 2*N for DXM write. On the Diopsis tile, the ARM processor runs at a clock frequency which is double of the AMBA bus used as unit of measure. This factor 2 can be taken in account in the estimate of time of ARM access to SRAM. The DSP data memory can be accessed by ARM in 6*(N/8)+N cycles for write and 8*N cycles for read.

The performance estimation results are summarized in table 5. The overall number of cycles required for the communication using AMBA burst mode is: approximately 8856k when all the data transfer is made via DXM; 7884k in the second case using REG, DXM and DMEM storage resources and 3960k in the third case using the SRAM and DMEM local memories. Thus, if the software code makes use of the existing hardware resources, an improvement in communication performance can be obtained. This improvement corresponds to 11% in the second communication mapping case and 55% in the third case. The communication protocol is specified in the initial Simulink model by annotating the communication units.

Table 5. Memory accesses

Communication Scheme	Transactions [KWords]				Total cycles	—
	DXM	SRAM	REG	DMEM		
DXM+DXM+DXM	5256k	0	0	0	8856k	100%
DXM+REG+DMEM	4608k	0	72k	576k	7884k	89%
SRAM+SRAM+DMEM	0	4680k	0	576k	3960k	45%

4.6.2. H.264 Application on Diopsis R2DT

The transaction accurate architecture of the Diopsis R2DT tile with Hermes NoC is illustrated in figure 51.

The tasks code is combined with the DwarfOS operating system and the implementation of the *send_data(…)/recv_data(…)* communication primitives to build each software stack running on the processors. The processors execute single task on top of the operating system. The OS is required for the interrupt routine services and the application boot.

The hardware platform is composed of the detailed three processor subsystems (ARM9-SS, DSP1-SS and DSP2-SS), one global memory subsystem (MEM-SS) and the peripherals on tile subsystem (POT-SS). The different subsystems are interconnected through an explicit Hermes NoC available in Torus and Mesh topologies.

Figure 51 presents the transaction accurate architecture of the Diopsis R2DT tile with NoC running the H.264 encoder application. The local architectures of each subsystem are detailed, including network interfaces, local bus, data memories and registers, abstract processor models, synchronization components, interrupt controller or DMA engines.

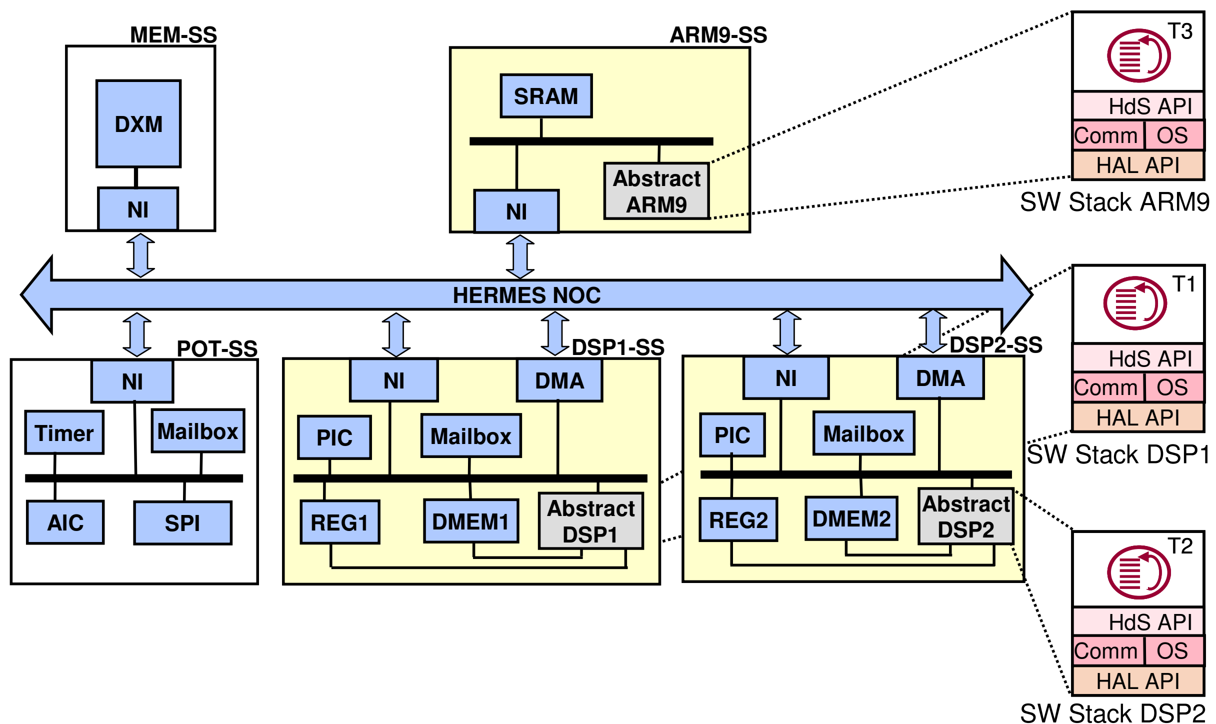


Figure 51. Global View of the Transaction Accurate Architecture for Diopsis R2DT with Hermes NoC running H.264 Encoder Application

The Hermes NoC at the transaction accurate architecture adds more architectural details such as topology, routing algorithm and router buffer size. The Hermes NoC model is composed of the same basic elements as the virtual architecture level: network interface, mapping table and routers but with a more detailed implementation. Topology (e.g. mesh, torus), routing algorithm (e.g. pure XY, west first), arbiter algorithm (e.g. round robin, priority based) and buffer size (e.g. number of flits) can be varied. The packet structure in this model is composed of destination address, size and body fields, similar to that assumed in the synthesizable NoC description. The Hermes NoC allows at the transaction accurate architecture level extracting information from the system communication architecture, like: (*i*) number of routing requests; (*ii*) number of packets inserted into the NoC; (*iii*) amount of bytes exchanged; (*iv*) the average of bytes per packet; (*v*) the number of packets transmitted, (*vi*) number of routing request failed due to NoC congestion.

At the transaction accurate architecture level, the DMA components belonging to the DSP subsystems become explicit and have direct link to the interconnect component. Thus, the Hermes NoC for the Diopsis R2DT architecture requires seven access points: five for the different subsystems, as previously presented in the virtual architecture model and two additional for the DMA components.

The different subsystems can be mapped over the NoC in different ways. The following paragraphs describe with details an example of IP cores mapping scheme. Thus, in a first scheme, the network interfaces connect the following IP cores to the NoC:

- The ARM9-SS is connected to network interface with address 1x0.
- The network interface with address 2x1 connects the DSP1-SS.
- The network interface with address 1x1 connects the DMA of the DSP1-SS.
- The network interface with address 1x2 connects the DMA of the DSP2-SS.
- The network interface with address 2x2 connects the DSP2-SS to the NoC.
- The network interface corresponding to the MEM-SS has address 0x0.
- The network interface connecting the POT-SS has address 0x1.

The NoC was adopted in two topologies: mesh and torus. In both cases, the NoC has 9 routers (3x3). Each router is connected to the corresponding network interface and the neighbor routers.

Figure 52 shows the NoC employing a 2D mesh topology, a pure XY routing algorithm and a round robin arbiter algorithm at each router and wormhole as packet switching strategy.

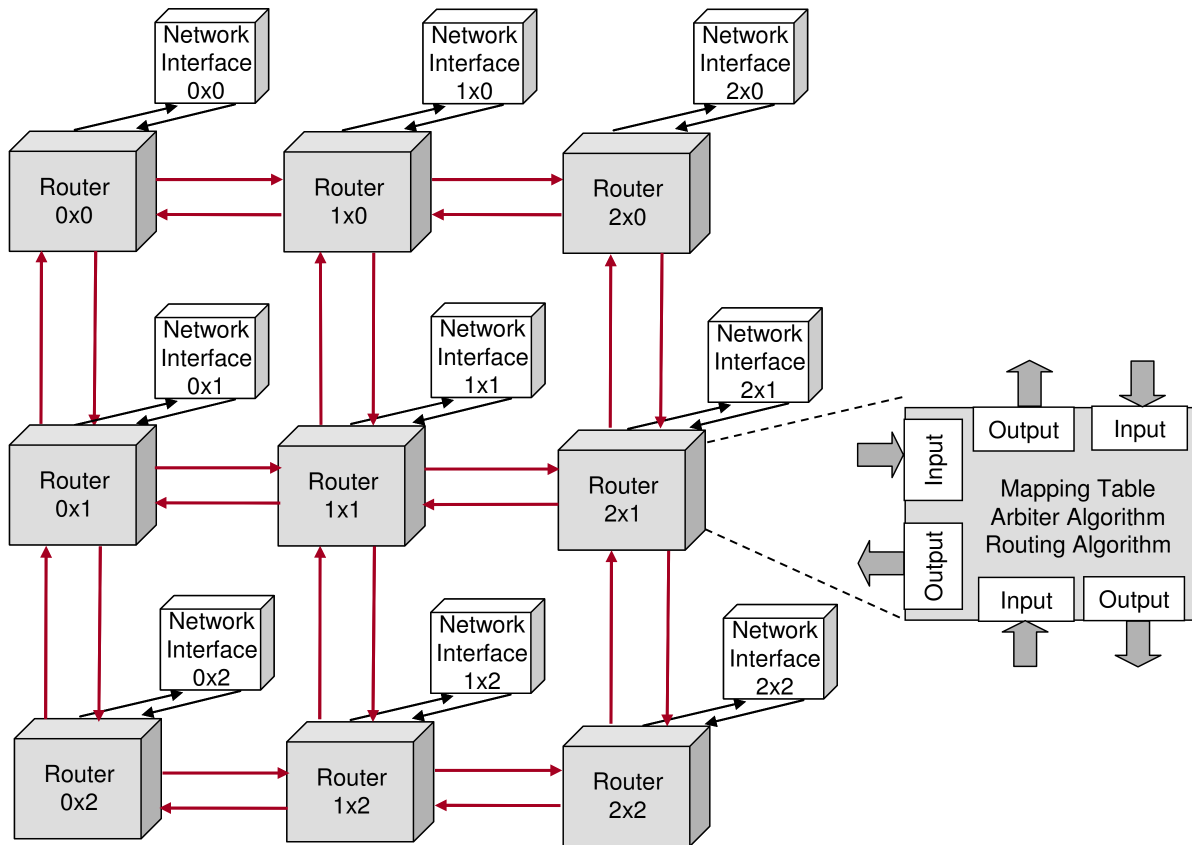


Figure 52. Hermes NoC in Mesh Topology at Transaction Accurate Level

Table 6 shows the results captured during the transaction accurate architecture Mesh model simulation in case of the H.264 encoder application.

Table 6. Mesh Noc Routing Requests

IP core	NoC @	TOTAL	LOCAL	NORTH	SOUTH	EAST	WEST
MEM-SS	0x0	20,00%	6,39%	6,18%	0,00%	7,43%	0,00%
POT-SS	0x1	20,63%	7,22%	3,19%	7,22%	2,99%	0,00%
	0x2	3,19%	0,00%	0,00%	0,00%	3,19%	0,00%
ARM9-SS	1x0	21,04%	7,43%	7,22%	0,00%	0,00%	6,39%
DSP1-SS (DMA)	1x1	10,21%	0,00%	0,00%	0,00%	2,99%	7,22%
DSP2-SS (DMA)	1x2	3,19%	0,00%	0,00%	0,00%	3,19%	0,00%
	2x0	6,18%	0,00%	0,00%	0,00%	0,00%	6,18%
DSP1-SS (NI)	2x1	9,17%	2,99%	0,00%	6,18%	0,00%	0,00%
DSP2-SS (NI)	2x2	6,39%	3,19%	0,00%	3,19%	0,00%	0,00%

The first and the second columns represent the correspondence between the different subsystems and the NoC access points. A routing request is performed at least once per packet per router that it will cross. Depending on the application, the NoC structure, routing algorithm, NoC congestion state, the routing request can occur as many times as needed inside a router. For the H.264 encoder simulation with 10 frames QCIF YUV 420 format, 96618508 routing requests were issued. The third column of table 6 presents the percentage of routing requests at each router, while the other columns detail this information related to the router port (local to the corresponding network interface, north, south, east or west). These results were captured in the case of mapping all the communication buffers onto the external memory.

Figure 53 shows the amount of data that traverses each router in the Mesh NoC for the H.264 encoder application with using external memory for communication between the processors.

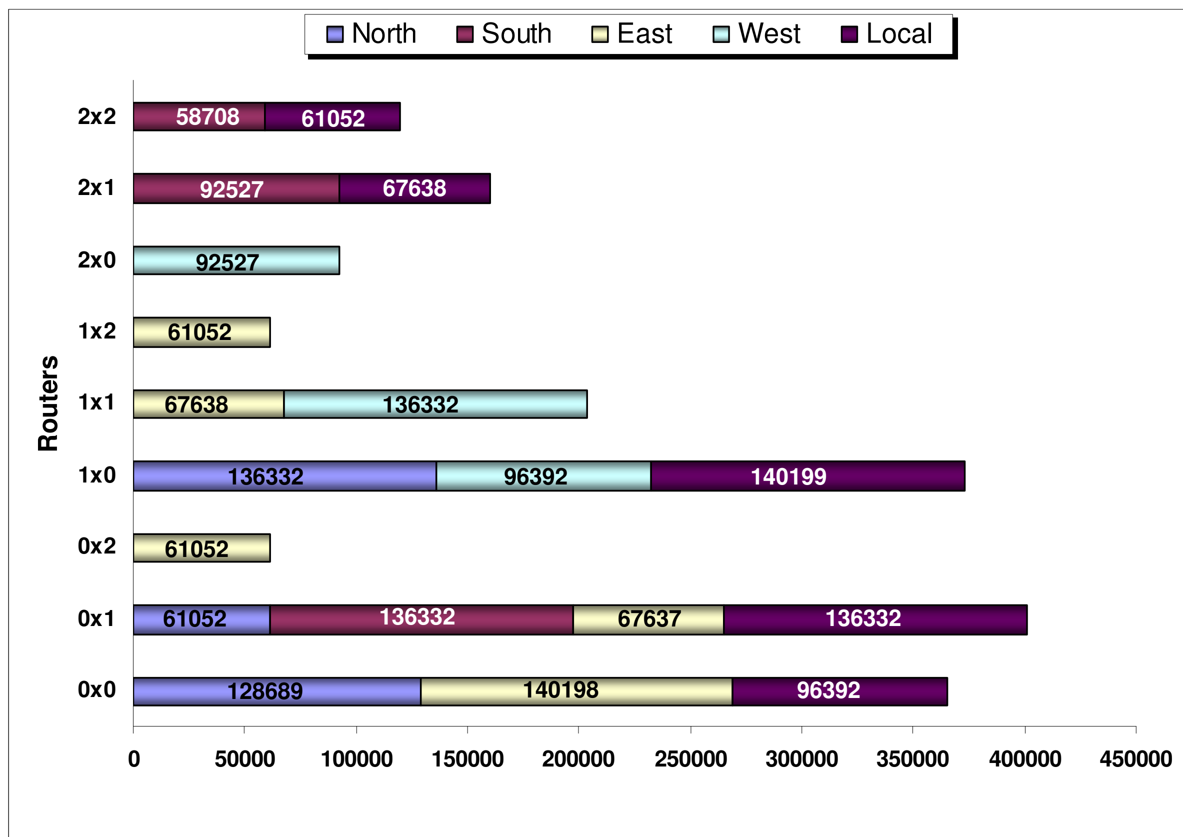


Figure 53. Total KBytes Transmitted through the Mesh

The local port of each router inserts packets to the NoC, while the remaining ports transfer them inside the NoC. The value assigned to the local port of the router 0x0 (MEM

SS) corresponds to response packets due to read requests or confirmation packets due to write requests. Block transfer operations (amount of operation that will be transferred in one packet) permit to optimize the amount of data exchanged inside the NoC by minimizing the amount of control data.

In the second topology, the adopted NoC was a 2D Torus topology and deadlock free of a non-minimal west-first routing algorithm proposed by Glass and Ni [Gla 94]. Figure 54 presents the Hermes 3x3 Torus NoC.

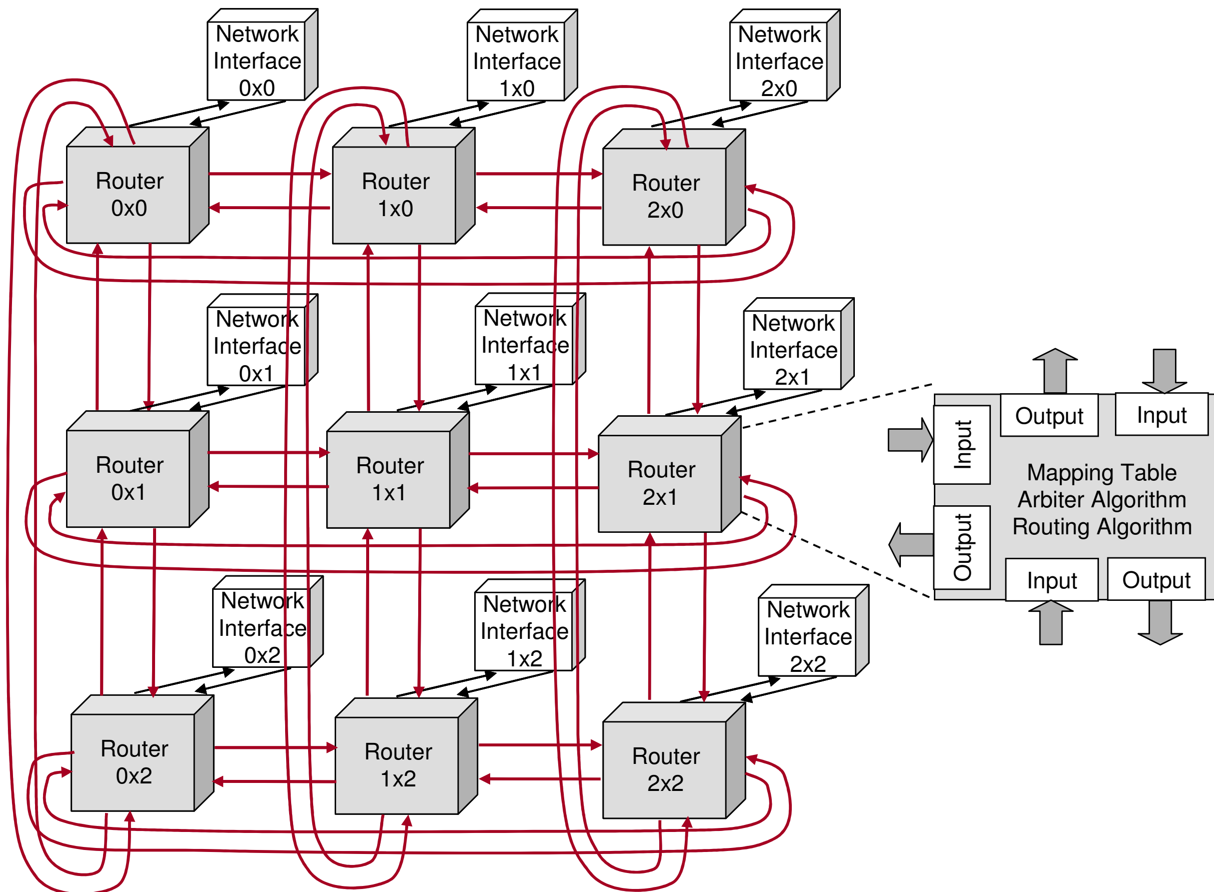


Figure 54. Hermes NoC in Torus Topology at Transaction Accurate Level

The H.264 encoder simulation with 10 frames QCIF YUV 420 format using Torus NoC topology involved approximately 78217542 routing requests, representing 19% of reduction when compared to the Mesh NoC. This was possible because the 2D torus topology has longest minimum paths that are only half of those in 2D meshes. Also, torus networks have better path diversity than meshes, which, if exploitable by the routing algorithm, leads to diminished network congestion, thus reducing routing requests.

Table 7 presents these results. The first columns represent the correspondence between the IP cores and network interfaces, while the others show the distribution of the routing requests along the local, north, south, east and west ports of each router. The results were captured in case of the mapping all the communication buffers onto the external memory.

Table 7. Torus Noc Routing Requests

IP core	NoC @	TOTAL	LOCAL	NORTH	SOUTH	EAST	WEST
MEM-SS	0x0	25,67%	8,90%	4,28%	4,34%	8,14%	0,00%
POT-SS	0x1	20,00%	7,86%	0,00%	7,86%	0,00%	4,28%
	0x2	4,33%	0,00%	0,00%	0,00%	0,00%	4,33%
ARM9-SS	1x0	16,28%	8,14%	7,86%	0,00%	0,00%	0,28%
DSP1-SS (DMA)	1x1	7,86%	0,00%	0,00%	0,00%	0,00%	7,86%
DSP2-SS (DMA)	1x2	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
	2x0	8,62%	0,00%	0,00%	0,00%	8,62%	0,00%
DSP1-SS (NI)	2x1	8,57%	4,28%	0,00%	4,28%	0,00%	0,00%
DSP2-SS (NI)	2x2	8,68%	4,34%	4,34%	0,00%	0,00%	0,00%

Table 8 sums up the amount of data transferred through the Torus NoC during the H.264 encoder simulation. The third column of the table represents the amount of data and control information exchanged (e.g. operation request, confirmation response, etc). The other columns of the table show the amount of data transmitted per each router port.

Table 8. Torus Noc Amount of Transmitted Data [Bytes]

	NoC @	LOCAL	NORTH	SOUTH	EAST	WEST
MEM-SS	0x0	110724784	80393092	68768172	127341684	0
POT-SS	0x1	122941472	264	122941856	0	80393360
	0x2	0	0	0	0	68768436
ARM9-SS	1x0	127342228	1229941340	132	0	4399692
DSP1-SS (DMA)	1x1	0	0	0	132	122941208
DSP2-SS (DMA)	1x2	0	0	0	132	0
	2x0	0	0	0	106325092	528
DSP1-SS (NI)	2x1	80393908	396	40196920	264	0
DSP2-SS (NI)	2x2	68768964	66128700	528	0	0

Figure 55 shows a screenshot captured during the simulation of the H.264 encoder running on the Diopsis R2DT architecture with Torus NoC.

In order to analyze the communication performances, the AMBA bus is also experimented as global interconnect instead of the Hermes NoC. The average throughput of the interconnect component in order to execute the H.264 in real time (25 frames/s) was 235Mbytes/s for the NoC and 115Mbytes/s for the AMBA.

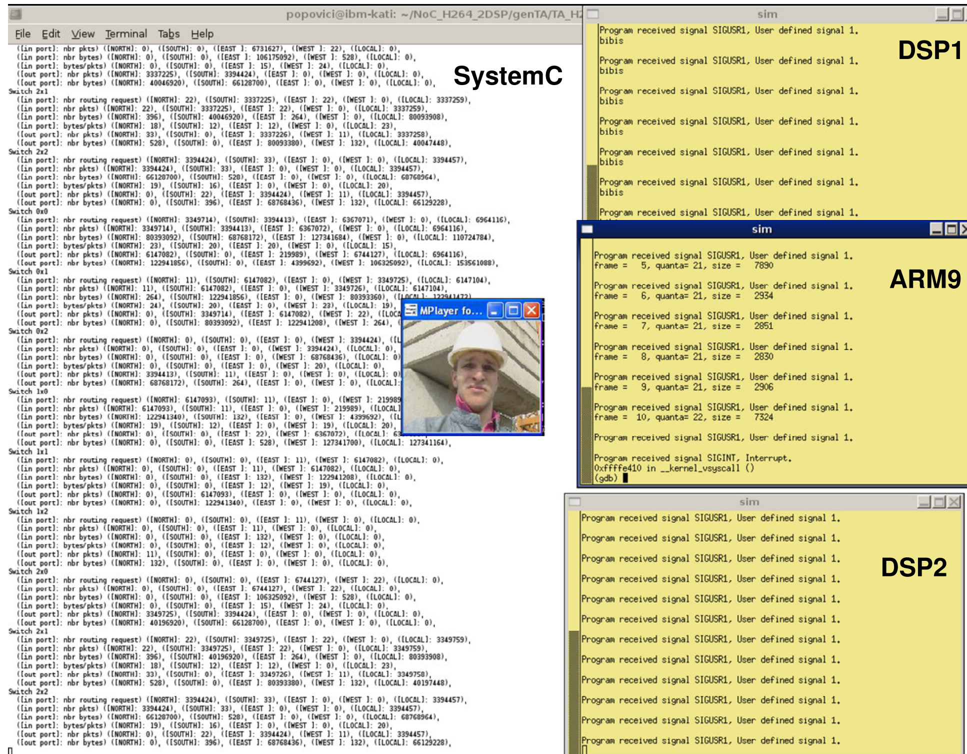


Figure 55. Simulation Screenshot of H.264 Encoder Application running on Diopsis R2DT with Torus NoC

The NoC allows various mapping schemes of the IPs over the NoC with different impact on performances. In this document, two different mappings of the IP cores over the Mesh and Torus NoC are experimented: scheme *A*, detailed in the previous paragraphs and scheme *B* with the MEM-SS connected at network interface with address 1x1 (both x and y coordinates are 1). Figure 56 summarizes the correspondence Network Interface and IP core in case of these two IP mapping schemes.

Y \ X	0	1	2
0	MEM-SS	POT-SS	-
1	ARM9-SS	DMA1	DMA2
2	-	DSP1-SS	DSP2-SS

Scheme A

Y \ X	0	1	2
0	DMA1	-	-
1	ARM9-SS	MEM-SS	DSP2-SS
2	POT-SS	DSP1-SS	DMA2

Scheme B

Figure 56. IP Cores Mapping Schemes A and B over the NoC

Table 9 presents the results of the transaction accurate simulation: estimated execution cycles of the H.264 Encoder, the simulation time using the different interconnect components on a PC running at 1.73GHz with 1GBytes RAM and the total routing requests for the NoC. These results were evaluated for the two considered IP mapping schemes shown in figure 56 (*A* and *B*) and for three communication buffer mapping schemes: *DXM+DXM+DXM*, *DMEM1+DMEM2+SRAM* and *DMEM1+SRAM+DXM*. The AMBA had the best performance, as it implied the fewest clock cycles during the execution for all the communication mapping schemes. The Mesh NoC attained the worse performance in case of mapping all the communication buffers onto the *DXM* and similar performance with the Torus in case of using the local memories.

Table 9. Execution and Simulation Times of the H.264 Encoder for Different Interconnect, Communication and IP Mappings

Communication Mapping Scheme	Interconnect	IPs Mapping over NoC	Execution Time at 100MHz [ns]	Simulation Time [min]	Execution Cycles	Simulation Cycles/Second	NoC Routing Requests	Average Interconnect Latency [Cycles/Word]
DXM+DXM+DXM	Mesh	Scheme A	64028725	36min	3201436	1482	96618508	25
DXM+DXM+DXM	Torus	Scheme A	46713986	28min29s	2335699	1527	78217542	16
DMEM1+DMEM2+SRAM	Mesh	Scheme A	28573705	12min54s	1428685	1846	13118044	10
DMEM1+DMEM2+SRAM	Torus	Scheme A	26193039	12min	1309652	1819	12674692	9
DMEM1+SRAM+DXM	Mesh	Scheme A	26233039	14min55s	1594237	1466	13144538	11
DMEM1+SRAM+DXM	Torus	Scheme A	26193040	14min48s	1309652	1475	14479723	10
DXM+DXM+DXM	Mesh	Scheme B	35070577	18min34s	1753529	1574	24753610	9
DXM+DXM+DXM	Torus	Scheme B	35070587	19min8s	1753529	1527	24753488	9
DMEM1+DMEM2+SRAM	Mesh	Scheme B	31964760	17min8s	1598238	1555	18467386	13
DMEM1+DMEM2+SRAM	Torus	Scheme B	31924752	16min14s	1595238	1639	15213557	13
DMEM1+SRAM+DXM	Mesh	Scheme B	31964731	18min38s	1598237	1430	18512403	15
DMEM1+SRAM+DXM	Torus	Scheme B	31924750	16min42s	1596238	1593	18115966	14
DXM+DXM+DXM	AMBA	-	17436640	8min24s	871832	1730	-	9
DMEM1+DMEM2+SRAM	AMBA	-	17435445	7min18s	871772	1990	-	9
DMEM1+SRAM+DXM	AMBA	-	17435476	7min17s	871774	1995	-	9

This is explained by the small numbers of subsystems interconnected through the NoC. In fact, NoCs are very efficient in architectures with more than 10 IP cores interconnected, while they can have a comparable performance results with the AMBA bus in less complex architectures. Between the NoCs, the Torus has better path diversity than Mesh. Thus, Torus reduces network congestion and decreases the routing requests. Also, scheme *A* of IP cores mapping provided better results than scheme *B* for the *DMEM1+DMEM2+SRAM* buffer mapping. For the other buffer mappings the performance of scheme *A* was superior to scheme *B*. In fact, the ideal IP cores mapping scheme would have the communicating IPs

separated by only one hope (number of intermediate routers) over the network to reduce latency.

Comparing with the virtual architecture, the transaction accurate interconnects fully implement the bus, respectively the NoC protocol. Thus it provides accurate characteristics. Therefore, the simulation of the transaction accurate interconnects requires higher simulation time compared with the virtual architecture. But, during both design steps, the NoC needs more time for the application simulation than buses due to its high complexity.

4.7. State of the Art and Research Perspectives

4.7.1. State of the Art

Current literature offers large set of references dealing with transaction accurate architecture design and software native execution using an abstract hardware platform.

ChronoSym [Bac 05] presents a fast and accurate SoC cosimulation that allows validation of the integration of the tasks code with the operating system. It is based on an OS simulation model and annotation of software with execution delays. The abstract execution model of the processors in the transaction accurate architecture presented in this document is similar with the timed bus functional model used in the ChronoSym approach, but it is not annotated for accurate estimation.

[Bou 05-b] presents an abstract simulation model of the processor subsystem. In this work, the processor subsystem is not defined as a set of hardware components, but it is viewed from a software point of view. Thus, the processor subsystem is made of execution, access and data unit elements to allow early validation of the MPSoC architecture and native time accurate simulation of the software.

[Ger 07], based on the work described in [Bou 04], resumes a hardware-software interface modeling approach in SystemC at the transaction accurate architecture level. This work uses the concept of required and provided services in the modeling of the hardware-software interfaces. The hardware-software interface is assembled using software, hardware and hybrid elements.

[Kempf 05] illustrates a configurable event-driven Virtual Processing Unit (VPU) to capture timing behavior of multiprocessor multithreaded platforms through flexible timing

annotation. The VPU enables investigation of the mapping of the application tasks with respect to time and space and early design space exploration.

[Schir 07] deals with abstract modeling of embedded processors using TLM. This work develops a high level abstract processor model that allows fast simulation, acceptable accuracy in simulated timing and exposing the structure of the software architecture (e.g. drivers and interrupts). This approach is similar with the abstract execution model of the processor belonging to the transaction accurate architecture.

[Ber 04] details the Synopsys System Studio design tool that allows a SoC design flow from system level to implementation by passing through several abstraction levels. One of the intermediate refinement steps corresponds to the development at the platform level, which represents a TLM platform of the hardware that allows starting the development of the software. The software development itself uses a specific development and simulation kernel such as RTLinux, together with an interface layer to the virtual processors on the platform.

[Has 05] [Has 06] presents a simulation model of μTRON-based RTOS kernels in SystemC. They developed a library of APIs that supports preemption, task priority assignment or scheduling RTOS services by native execution and a SystemC wrapper to encapsulate the OS simulation model into the bus functional model (BFM) of the hardware platform. Their approach is similar with the presented approach, but they do not give details on the hardware side.

[Shin 06] presents a communication design flow based on automatic TLM model generation. They allow generation and refinement of bus based communication architectures, including bus bridges and transducers. But they do not address software code adaptation to specific communication protocol implementation, in order to optimize the overall communication performance.

[Kli 07] proposes a hardware procedure call (HPC) protocol to abstract the platform dependent details of the TLM communication between the different subsystems, by providing an additional layer for the software modeling on top of transaction-level models.

4.7.2. Research Perspectives

The most important research perspective regarding the transaction accurate architecture design consists of annotation of the software code with execution delays for accurate software performance estimation and annotation of the hardware code for accurate

communication architecture performance estimation. This could be managed by applying a similar approach with timed bus functional model used in ChronoSym [Bac 05].

Other research perspective represents the automatic generation of the transaction accurate architecture. The generation could be made possible by applying a service-based modeling of the hardware-software interface as described in [Ger 07]. The composition of the services easies the automatic generation tools to reduce design time. The generation can be performed from the system architecture or virtual architecture. Generation from the system architecture enables generation of different detail levels from the same specification (virtual architecture, transaction accurate architecture and virtual prototype). The generation from the virtual architecture enables gradual refinement of the hardware/software architecture based on the performance estimation performed at this level.

On another proposed research perspective refers to the design at the transaction accurate architecture level of more complex multi-tile architectures such as Tile64 [Tilera] or AM2000 [Ambric] running massive parallel applications.

4.8. Conclusions

This chapter defined the transaction accurate architecture design. It presented the software organization as final application task code running upon a real time OS and the hardware organization in detailed subsystems interconnected through an explicit network component.

The transaction accurate architecture design was performed using SystemC for 3 case studies: Token Ring mapped on the 1AX architecture, Motion JPEG running on the Diopsis RDT architecture and H.264 Encoder running on the Diopsis R2DT architecture.

The simulation of the transaction accurate architecture model allowed to validate the integration of the final application tasks code with an OS and communication software adapted to the synchronization protocol. It also gave more precise information on the interconnect model. This includes the number of conflicts in the global bus, the amount of NoC congestion, the number of transmitted bytes through the bus or NoC, the number of routing requests, the number of times some routers failed to transmit the packet due to conflicts inside the NoC or the average bytes per packet.

The transaction accurate architecture design also allows exploration of different IP cores mapping over the NoC in order to analyze their impact on the overall performances.

Chapter 5

VIRTUAL PROTOTYPE DESIGN

This chapter details the virtual prototype design. The virtual prototype design consists of integrating the HAL implementation into the software stack and establishing the final memory mapping. The validation of the software is performed by using classical co-simulation with Instruction Set Simulators (ISS). The key contribution in this chapter represents the virtual prototype definition, organization and design using SystemC for the Token Ring application running on the 1AX architecture, Motion JPEG running on the Diopsis RDT architecture and H.264 Encoder running on the Diopsis R2DT architecture. The Motion JPEG application is executed using ISS on different types of single processor (ARM7, ARM9 and DSP) and the H.264 Encoder is simulated using ISS running both on multiprocessor architecture with 3 ARM7 processors and single processor (ARM7 and ARM9). The simulation of the virtual prototype model allows to validate the software binary and the memory mapping.

5.1. Introduction

The virtual prototype design consists of software adaptation to specific target processors and memory mapping. This includes the integration of the processor dependent software code into the software stack, more precisely the HAL integration with the tasks code, OS and communication software components. The result of the virtual prototype design represents the virtual prototype model.

5.1.1. Definition of the Virtual Prototype

The lowest MPSoC abstraction level is called virtual prototype (VP). The software stack is fully explicit, including the HAL layer to access the hardware resources and it is detailed to ISA (Instruction Set Architecture) level to be adapted for a specific processor. The hardware architecture incorporates an ISS for each processor to execute the final binary code. At the virtual prototype level the communication consists of physical I/Os, e.g. *load/store*.

According to [Hong 06], the virtual prototype has the following objectives:

- Measure system performance and analyze its bottlenecks
- Find out optimization points from the bottleneck analysis by using traces and profile data
- Allow full software stack and memory mapping validation before the real hardware is available
- Evaluate architectural decisions of both hardware and software side.

The virtual prototype is characterized by three issues: timing accuracy, simulation speed and development time. The virtual platform has to be accurate enough to analyze system performance including hardware-software interaction, fast enough to execute the software and it has to be available earlier than the real-chip development. Unfortunately, these criteria are difficult to be accomplished simultaneously: accurate platforms usually require detailed information, thus they impose slow simulation speed and substantial time to develop.

The simulation at the virtual prototype level allows performance validation and it corresponds to classical hardware/software cosimulation models with Instruction Set Simulators [Row 94] [Sem 00] for the processors. The simulation performed at this level is

cycle accurate. It allows validating the memory mapping of the target architecture and the final software code. It also provides precise performance information such as software execution time, computation load for the processors, the number of clock cycles spent on communication, etc. The hardware platform includes all the hardware components such as cache memories or scratchpads.

Figure 57 illustrates a global view of the virtual prototype, composed of ISS for the processors and the other hardware components, such as local resources of the processor subsystems, memory subsystem and the network component. The left part of the figure corresponds to the hardware architecture, while the right part represents the software stack at the virtual prototype level running on the one of these processor subsystems.

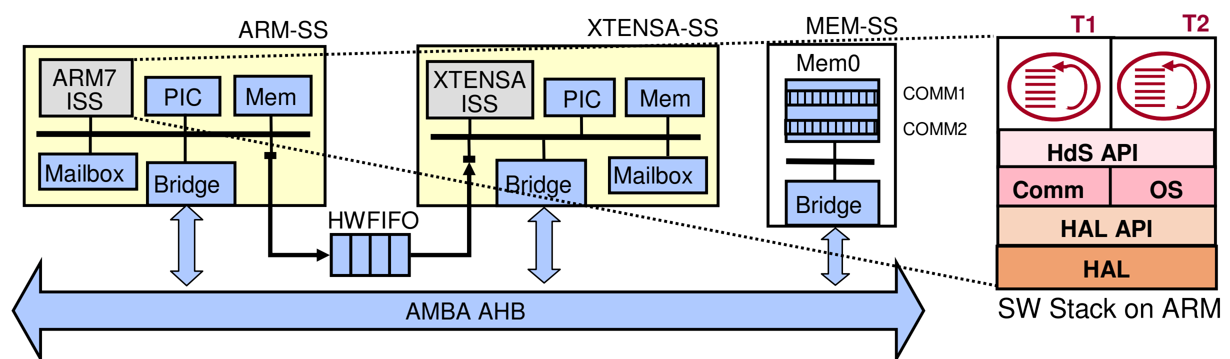


Figure 57. Global View of the Virtual Prototype

5.1.2. Global Organization of the Virtual Prototype

The virtual prototype model is a hierarchical model. The virtual prototype is composed of detailed software and hardware subsystems interconnected through a global interconnect component. The software subsystems incorporate an Instruction Set Simulator (ISS) for each processor to execute the final binary code and cycle accurate components for the rest of the architecture. The ISS is a software environment which can read microprocessor instructions and simulate their execution. Most of these tools can provide simulation results like values in memory and registers, as well as timing information (e.g. clock cycle statistics).

Example 31. Virtual Prototype for the Token Ring application mapped on the 1AX architecture

Figure 57 shows a conceptual representation of the virtual prototype for the Token Ring application mapped on the 1AX

architecture. Figure 57 illustrates that for the Token Ring application running on the 1AX architecture, the virtual platform contains two processor subsystems, corresponding to the ARM, respectively XTENSA processors and the global memory subsystem. All the subsystems are interconnected by an explicit AMBA bus. The processor subsystems encapsulate the ISS for the ARM7 processor, respectively XTENSA processor ISS.

The software stack represents the final software code adapted to specific processor implementation. The communication consists of physical I/Os, e.g. *load/store.*

5.2. Basic Components of the Virtual Prototype Model

The basic components of the transaction accurate architecture Model are software and hardware components. The software components consist of the tasks code, operating system, communication library and HAL, while the hardware components represent detailed subsystems with ISS for processor.

5.2.1. Software Components

At the virtual prototype level, the software stack running on each processor is completely detailed and represents the final binary of the software. The binary image will run on the hardware simulation platform or on the physical architecture board in case it is available.

The software stack is composed of all the software components: application tasks code, communication implementation, operating system, HAL and the APIs to pass from one component to another. Thus, the software stack is fully explicit, including the HAL layer to access the hardware resources and it is detailed to ISA (Instruction Set Architecture) level for a specific processor. The HAL represents a thin low software layer, totally dependent of the target processor core. The HAL allows the software to access and configure the hardware peripherals.

Example 32. Software Components for the Token Ring application at the Virtual Prototype Level

```
     For the Token Ring application, both software stacks
running on the ARM7, respectively XTENSA processor are made of
the application tasks code (T1 and T2 for the ARM7 processor and
T3 for the XTENSA), the DwarfOS as operating system,
communication library and the HAL specific implementation for the
ARM7, respectively XTENSA processors. For both software stacks,
the data and program code are mapped explicitly on the memory,
conform to the final memory mapping.
```

5.2.2. Hardware Components

The components of the hardware platform are those at the previous abstraction levels but detailed with cache memories, scratchpads, memory management units and special registers. The hardware architecture contains all the resources required to validate the final software stack. Therefore, it contains the local components of each processor and hardware subsystem. In order to execute the software stack, the virtual platform contains an Instruction Set Simulator (ISS) corresponding to each processor core.

Example 33. Hardware Components for the Token Ring application at the Virtual Prototype Level

```
     For example, in the case of the Token Ring application
running on the 1AX architecture, the hardware platform contains
ISS encapsulated in the processor subsystems, specific to the
ARM7, respectively XTENSA processors.
```

5.3. Modeling Virtual Prototype in SystemC

The virtual prototype is modeled according to the annotated architecture parameters of the initial system architecture model and the results of the virtual architecture and transaction accurate architecture model simulation.

5.3.1. Software at Virtual Prototype Level

The software design at the virtual prototype level consists of developing the final software binary that will run on each processor of the hardware platform. The binary image is obtained from the final software stack. This software stack contains all the software components: those validated at the transaction accurate architecture level, namely the application tasks code, operating system and communication library, and an additional low level component, more precisely the HAL.

[Yoo 03] defines the HAL as all the software that is directly dependent on the underlying hardware. Example of HAL code represents the software code written in assembly language interpretable by the processor, such as context switch, boot code or enabling and disabling the interrupt vectors, respectively code for configuration and access to hardware resources, such as MMU (Memory Management Unit), system timer, on-chip bus, bus bridge, the hardware dependent part of the device drivers that allow to access the I/O devices, resource management, such as tracking system resource usage (check battery status) or power management (set processor speed).

To create the complete binary software image, the designer has to develop the configuration and build files (e.g. Makefile) which select and configure the library components (OS, communication, HAL) and controls compilation and linking of the different software components. Using a cross compiler, the final target binary is created for each processor, than can be executed on the target processor of the virtual platform.

Example 34. Software Code for the Token Ring application at the Virtual Prototype Level

Figure 58 presents an example of HAL code performing a context switch between two tasks running on the ARM7 processor in the case of the Token Ring application. Instead of using a simulation model of the HAL APIs as it was employed at the transaction accurate architecture level, the virtual prototype gives the final implementation of the HAL API *__ctx_switch(…)* by using an explicit HAL software code. The context switch needs two basic operations to be performed: store the registers of the current task and load the registers of the new task.

```
__ctx_switch                  ; r0 old stack pointer, r1 new stack pointer
STMIA r0!,{r0-r14}            ; save the registers of current task
LDMIA r1!,{r0-r14}            ; restore the registers of new task
SUB pc,lr,#0                  ; return
END
```

Figure 58. HAL Implementation for Context Switch on ARM7 Processor

Figure 59 illustrates another example of low level code implementation of the HAL APIs that enables and disables the IRQ interrupts for the ARM processor. The interrupts are enabled and disabled by reading the CPSR (Current Program Status Registers) flags and updating bit 7 corresponding to bit I.

```
__inline void enable_IRQ(void)                      //HAL API
{
          int tmp;
          __asm
          {
                    MRS tmp, CPSR
                    BIC tmp, tmp,#0x80
                    MSR CPSR_c, tmp
          }
}

__inline void disable_IRQ(void)                     //HAL API
{
          int tmp;
          __asm
          {
                    MRS tmp, CPSR
                    ORR tmp, tmp, #0x80
                    MSR CPSR_c, tmp
          }
}
```

Figure 59. Enabling and Disabling ARM Interrupts

In order to select properly the libraries of all these software components (OS, communication, HAL) for the compilation of the software stack, a Makefile is required. Figure 60 details a Makefile used for cross-compilation of the software stack running on the ARM7 processor of the 1AX architecture. The Makefile contains the path to the application tasks code, target OS (*los-kernel*), communication library (*los-comm*) and the HAL library corresponding to the ARM7 processor (*lib/arm7*). It also identifies the compiler to be used, which in this case represents the *arm-elf-gcc* cross compiler provided by GNU [Gnu].

It also includes the path to the linker script *ldscript*, used to coordinate the linking process of the different object files obtained after the compilation. The *ldscript* guides also the loading process of the software image into the memory, by specifying explicitly the addresses where to load the program and data code of the software stack.

More details about the memory mapping will be given in the next paragraphs.

```
CC      =  arm-elf-gcc                                          # cross compiler
OBJDUMP= arm-elf-objdump
OSHOME =  /home/popovici/dwarfos

INCDIR = .
SRCDIR = .
OBJDIR = .
BINDIR = .

FLAGS  = -Wall -D_SIMULINK_ -DSTACK_SIZE=0x1000 -g -I$(INCDIR)
FLAGS += -I$(OSHOME)/include -I$(OSHOME)/include/libc
FLAGS += -DARCH_ARM7 -T$(OSHOME)/lib/arm7/ldscripts
FLAGS += -nostdinc -nostdlib -nodefaultlibs -g
LIBS   = -lh264-arm7 -L$(OSHOME)/lib/arm7                    # HAL library
LIBS  += -los-kernel -los-comm -lgcc -lc                     # OS & Comm libs
OBJSUF  = .o

SRC     = $(wildcard $(SRCDIR)/*.c)
OBJ     = $(SRC:$(SRCDIR)/%.c=$(OBJDIR)/%.o)
OSOBJS  = $(OSHOME)/lib/arm7/libos-hds.o

TARGET  = sw.bin


all:      $(TARGET)

$(TARGET): $(OBJ)
          @echo
          @echo 'creating binary "$(TARGET)"'
          $(CC) -o $(TARGET) $(OSOBJS) $(OBJ) $(LIBS) $(FLAGS)
          $(OBJDUMP) -D $(TARGET) > sw.d
          @echo '... done'
          @echo
```

Figure 60. Example of Compilation Makefile for ARM7 Processor

Loading software image in memory

An important aspect of the virtual prototype design consists of loading the binary image of the software into the memories of the chip. Usually, MPSoC architectures provide complex memory hierarchies composed of different memories, such as ROM, SRAM, DRAM, FLASH, etc. The binary image obtained after the compilation and linking is divided in two sections: read-only (RO) which contain the code and data only for read operations and read-

write (RW) section which contain the data that can be both read and written. Usually the RO part is loaded into a ROM memory. The RW part is stored in the ROM before the execution, and then it is initialized from the ROM into a RAM memory.

The structure of a binary image is defined by the number of regions and output sections, the positions in the memory of these regions and sections when the image is loaded and the positions in the memory of these regions and sections when the image is executed. Each output section contains one or more input sections. Input sections are the code and data information from the object files obtained after the compilation.

The image regions are placed in the system memory map at load time. Then, before execution of the image, some regions are moved to their execution addresses and some parts of memory are set to zero creating the ZI (zero initialize) sections. Thus, there are two different views of the memory map: load view and execution view. The load view defines the memory in terms of addresses where the sections are located at the load time, before execution of the image. The execution view describes the address of the sections while the image is executing. Figure 61 shows the load and execution view of the memories.

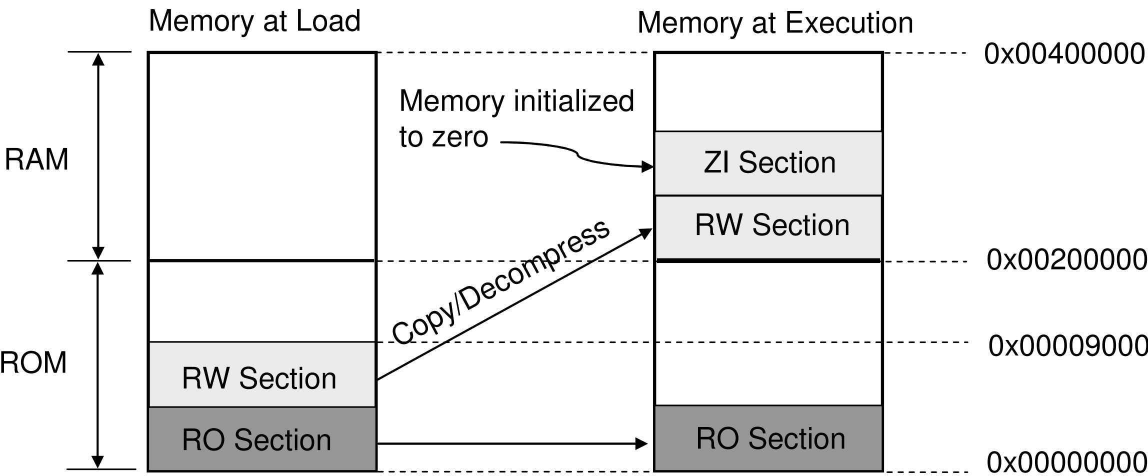


Figure 61. Load and Execution Memory View

The image memory map is specified during the linking phase. The linking can be done using command line options for software images with few loading and execution sections or by using scatter-loading description file for more complex cases. The scatter-loading description file represents a text file that specifies the grouping information of sections into regions and the placement addresses of the regions to be located in the memory maps. The scatter loading description file also allows to place the data at a precise address in the memory

map to access memory mapped I/Os and peripherals. Moreover, stack and heap addresses are defined using the same description file.

Figure 62 shows an example of scatter loading description file for an ARM processor according to the memory mapping described in figure 61 [Arm]. This scatter loading descriptor example defines one load region (*ROM_LOAD*) and two execution regions (*ROM_EXEC* and *RAM*). The entire program, including code and data is placed in ROM at *ROM_LOAD*. The RO code will execute from *ROM_EXEC*. Its execution address (0x0) is the same as its load address (0x0), so it does not need to be moved being a root region. The RW data will get relocated from *ROM_LOAD* to RAM at address 0x00200000. The ZI data will get created in RAM, above the RW data.

```
ROM_LOAD 0x0                   ; Start address of load region
{
    ROM_EXEC 0x0 0x9000        ; Start address and maximum size of exec region
    {
        * (+RO)                ; Place all code and RO data in this exec region
    }

    RAM 0x00200000 0x00200000 ; Start address and maximum size of exec region
    {
        * (+RW,+ZI)            ; Place all RW and ZI data into this exec region
    }
}
```

Figure 62. Example of Scatter Loading Description File for the ARM Processor

Before the execution of the binary image, the processor runs an initialization sequence code to setup and configure the system. Figure 63 presents an example of initialization code using HAL for the ARM processor [Arm].

The initialization sequence has two principal functions: *__main* and *__rt_entry*. The *__main* function is responsible for setting the run-time image memory map. It also performs the copy of code and data and initializes the ZI section with zero. The *__rt_entry* (run-time entry) function is responsible to set up the application stack and heap memories, to initialize the library functions and static data and it calls the constructors of global objects declared in C++. Then, the *__rt_entry* function continues with the *main* user function, which represents the entry point of the software stack. For instance, the *main* function can be the initialization function of the OS that declares and initializes the tasks running on the processor.

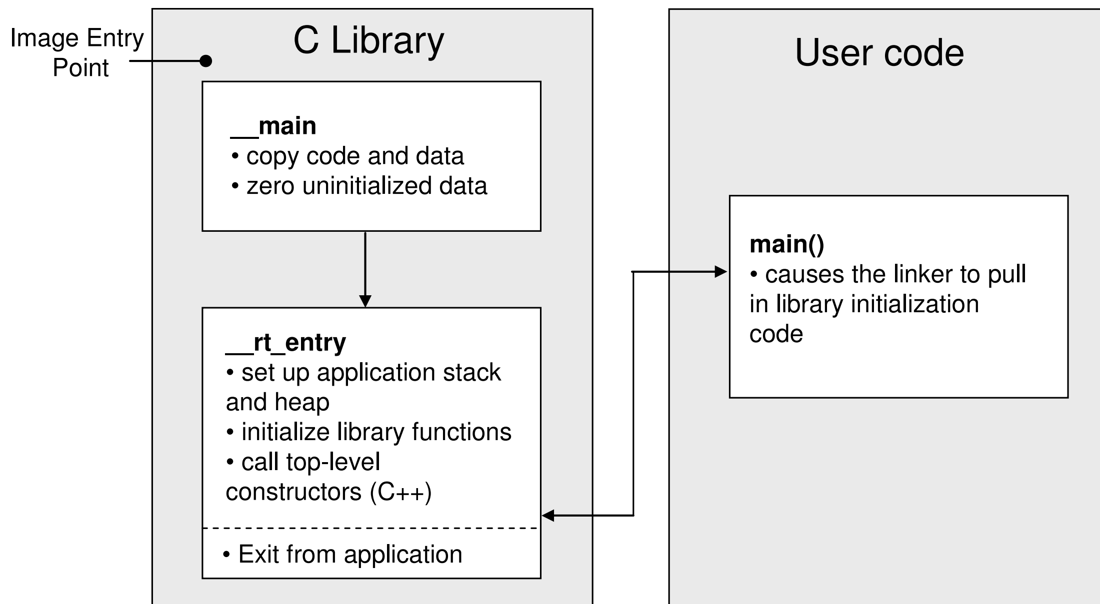


Figure 63. Example of Initialization Sequence for the ARM Processor

5.3.2. Hardware at Virtual Prototype Level

The hardware platform is fully detailed with cycle accurate TLM or RTL components for the hardware resources. The hardware at the virtual prototype level is composed of the same components as at the transaction accurate architecture level. In order to reach accurate performance estimation, the hardware modules are modeled at this level with cycle accuracy. Cycle accuracy can be achieved in two modeling methods:

- TLM modeling of the virtual prototype and use of execution delay annotation for cycle accuracy
- RTL (Register Transfer Level) modeling of the virtual prototype.

Both methods can make use of SystemC design language. The TLM modeling method has the advantage to ensure a fast simulation environment, while the RTL modeling may allow synthesizing the hardware architecture within a hardware-software MPSoC co-design flow.

The virtual prototype contains ISS for the processors in the processor subsystem to execute the software stack.

Example 35. Hardware Code for the Token Ring application at the Virtual Prototype Level

The virtual prototype in case of the Token Ring application running on the 1AX architectures is modeled using cycle accurate TLM. Figure 64 shows an example of processor subsystem for the ARM7-SS of the 1AX architecture running the Token Ring application. The ARM7-SS includes the processor core ArmCore SystemC module, which encapsulates an ISS of the software. The rest of the components of the ARM7-SS are those from the transaction accurate architecture level (local bus, local memory, bridge, interrupt controller, and mailbox).

```
#include "ARM7_SS.h"
extern int debug_flag;

ARM7_SS::ARM7_SS(sc_module_name name, char *bin)          // ARM7-SS
:sc_module(name)
{
    sys_bus = new TlmBus("sys_bus");                       // local bus

    core = new ArmCore("ARM7_ISS",bin,debug_flag);         // ARM7 ISS
    core->rw_port(*sys_bus);

    mem = new Sram("mem0",0x300000);                       // local memory
    mem->port(*sys_bus);
    mem->port.set_map(0x0,0x2FFFFF);

    bridge = new AhbIf("bridge");                          // bridge
    bridge->master(*sys_bus);
    bridge->slave(*sys_bus);
    bridge->slave.set_map(0x400000,0x7fffffff);

    pic = new Pic<1>("pic",0x20);                          // PIC
    pic->port(*sys_bus);
    pic->port.set_map(0x300000,0x30001f);

    sync = new Sync("sync",0x400);                         // mailbox
    sync->port(*sys_bus);
    sync->port.set_map(0x300800,0x300bff);

    TlmIntrSig *sig_sync = new TlmIntrSig("sig_sync");    //interrupt signals
    sync->intr(*sig_sync);
    pic->in_irq[0](*sig_sync);
    s1 = new TlmIntrSig("sig_intr1");
    s2 = new TlmIntrSig("sig_intr2");
    pic->out_fiq(*s1);
    pic->out_irq(*s2);
    core->nIrqPort(*s2);
}
```

Figure 64. SystemC Code of the ARM7-SS Module

5.3.3. Hardware-Software Interface at Virtual Prototype Level

At the virtual prototype level the communication consists of physical I/Os, e.g. *load/store*. The hardware-software interface is represented by the ISS for the processors. An Instruction Set Simulator (ISS) is a simulation model, usually coded in a high-level language such as C language, which mimics the behavior of a microprocessor by "reading" instructions and maintaining internal variables which represent the processor's registers.

5.4. Execution Model of the Virtual Prototype

The integration of instruction set simulators for the software execution on specific processors with hardware simulators of the architecture behavior is largely used in MPSoC domain. By using ISS, this approach allows simulating a detailed hardware-software interaction. The timing information can be measured instead of estimated as at the previous abstraction levels and design steps.

The execution model of the virtual prototype resides on a cosimulation between the software stack simulator and the hardware simulator [Nic 02]. Two types of simulators are combined: one for simulating the programmable components running the software and one for the dedicated hardware part [Erb 07]. The software stack is executed using processor specific ISS. Instruction-level or cycle accurate ISS simulators are commonly used. The hardware simulation is performed using hardware RTL descriptions realized in VHDL, Verilog or SystemC or cycle accurate TLM description realized in SystemC. In the following examples, we use SystemC for the hardware simulation.

The hardware-software simulation is driven by SystemC. The SystemC initializes the processor SystemC modules that encapsulate the ISS. During the simulation, the ISS features a simulation loop which fetches, decodes and executes instructions one after another. The ISS is developed as sequential software running on a single processor. The simulation performed at this level is cycle accurate. The simulation of the virtual prototype allows validating the memory mapping of the target architecture and the software binary.

Example 36. Execution Model for the Token Ring application at the Virtual Prototype Level

Figure 65 shows the execution model of the 1AX architecture running the Token Ring application. The model contains two ISS to

```
execute the binary codes, corresponding to the ARM7, respectively
XTENSA processors. The rest of the architecture components are
cycle accurate SystemC components modeled at TLM with execution
timing information.
```

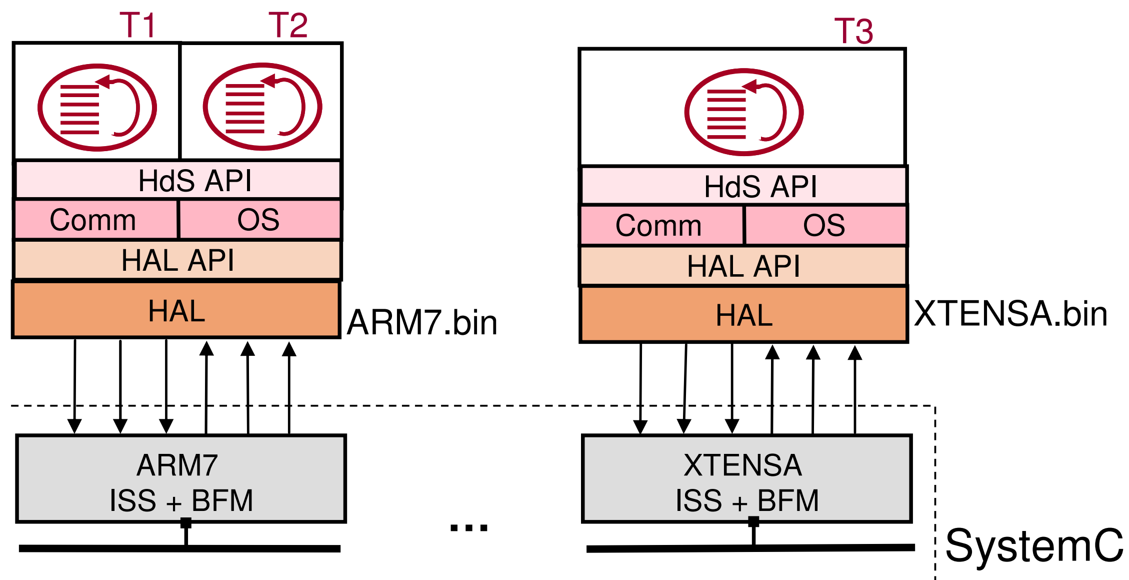


Figure 65. Execution Model of the Virtual Prototype

5.5. Design Space Exploration of Virtual Prototype

5.5.1. Goal of Performance Evaluation

The goal of the performance evaluation at the virtual prototype level is to validate the final software stack and the overall performance of the system. The performance evaluation is related to both computation and communication aspects.

Based on the results obtained by execution of the final software on the virtual prototype model, the designer may need to improve some parts of the design or revise design options due to unsatisfied design constraints, for example if real time requirements are not met, such as number of frames processed per second in multimedia applications, usually defined as 25 frames/second. Software optimization aims to decrease program and data size, usually achieved through application algorithm optimization or communication overhead reduction.

5.5.2. Architecture/Application Parameters

The virtual prototype validates the adaptation of the final software code to a specific processor.

The designer may choose different types of processor cores from the same processor family or different processor families. The different kinds of processor cores of the same family have a common architecture, but are differentiated by some specific features, such as size of data and instructions cache memories, bus interfaces, the availability of tightly coupled memory, power consumption, area, clock frequency [Mhz] or DSP extensions. Table 10 shows a subset of different characteristics of processors belonging to the ARM7 family compared to the ARM926EJ-S processor of ARM9 family [Arm]:

Table 10. ARM7 and ARM9 processors family

	Cache Size (Inst/Data)	Tightly Coupled Mem	Mem Mgmt	Bus Interface	Thumb	DSP	Jazelle
ARM720T	8k unified	-	MMU	AHB	Yes	No	No
ARM7EJ-S	-	-	-	Yes	Yes	Yes	Yes
ARM7TDMI	-	-	-	Yes	Yes	No	No
ARM7TDMI-S	-	-	-	Yes	Yes	No	No
ARM926EJ-S	16k/16k	Yes	MMU	AHB	Yes	Yes	Yes

The designer may change these parameters and may set up different configuration schemes, including target compilation optimizations, to increase the overall performance.

5.5.3. Performance Measurements

The simulation of the virtual prototype provides precise performance information such as software execution time, computation load for the processors, the number of cycles spent on communication, number of cycles spent by processors in idle state, etc.

Other important metrics that can be measured at this level are: program and data memory size requirements of the final software stack, number of cycles spent by the processor on certain application functions or number of instructions executed per clock cycle. This kind of data can be gathered thanks to the precise profiling capabilities of the most instruction set simulators. Usually, the virtual prototype is a cycle accurate model, thus it implies long simulation time. Therefore, the simulation time represents another key feature to be measured at the virtual prototype level.

Example 37. Performance Measurements for the Token Ring application at the Virtual Prototype Level

In case of the Token Ring application, the execution of the three tasks on a single ARM7TDMI processor without operating system requires 484775 clock cycles running at 60 MHz. The application compiled for a single ARM7 processor produces a code size of 1112 bytes and 108 data bytes. The computation of the FFT on the ARM7 processor involves 33329 clock cycles.

5.5.4. Design Space Exploration

At the virtual prototype level, the design space exploration consists of processor core configuration and exploration. The different types of processors cores or differently configured processors have different performances in terms of speed, power consumption and cost.

Example 38. Design Space Exploration for the Token Ring application at the Virtual Prototype Level

For instance, the XTENSA processor is a configurable processor provided by Tensilica [Ten]. The SoC designers may customize functional blocks to exactly match the required application. Because these processors are fully programmable, changes can be made in firmware even after the silicon production.

Generally, the configurable processors have two essential features:

- Configurability, which allows the designers to pick and configure the features they need.
- Extensibility, which allows designers to add multi-cycle execution units, registers, or register files. For instance, the Tensilica Instruction Extension (TIE) of the XTENSA processors is a methodology that allows designers to specify and verify the functional behavior of the new data path and the RTL is automatically generated [Ten].

Another space that can be explored at the virtual prototype level represents the memory mapping. Thus, the different data structures can be mapped on different memories at different addresses accessible through *load/store* instructions.

5.6. Application Examples at the Virtual Prototype Level

The following paragraph presents the virtual prototype model for the two case-studies: the Motion JPEG Decoder application running on the Diopsis RDT architecture with AMBA bus and the H.264 Encoder application running on the Diopsis R2DT architecture with Hermes NoC.

5.6.1. Motion JPEG Application on Diopsis RDT

At the virtual prototype level, the software stacks of the Motion JPEG Decoder application running on the two processors contain all the components. A processor specific HAL layer is linked with the application tasks, operating system and communication libraries. Usually, the HAL layer is provided by the processor vendors. Thus, a specific ARM7 HAL is implemented in the final software running on the ARM7. Similarly, the HAL of the DSP is integrated in the software stack. The two software stacks produce two different binary images that will be interpreted and executed by the ISS corresponding to each of these processors.

The hardware platform contains cycle accurate detailed components using TLM modeling with timing annotation or RTL modeling. Figure 66 illustrates a global view of the virtual prototype platform with the use of ISS as processor execution model.

Figure 67 summarizes the total execution cycles measured when executing the whole Motion-JPEG application on single processor single task configuration. The experimentation was done using three types of processor cores. The first processor core represents the ARM7TDMI-S processor of family ARM7. This processor works at frequency 60 MHz and has not data cache or instruction cache memories. The second core belongs to ARM9 processors family and represents the ARM926EJ-S type of core. This runs at 200 MHz frequency and is equipped with 16KBytes data cache and 16KBytes instruction cache memories. The third processor represents the magicV VLIW DSP processor, running at 100 MHz. In the all cases, the real-time execution requirement defines an image rate equal with 25 images/second to be decoded.

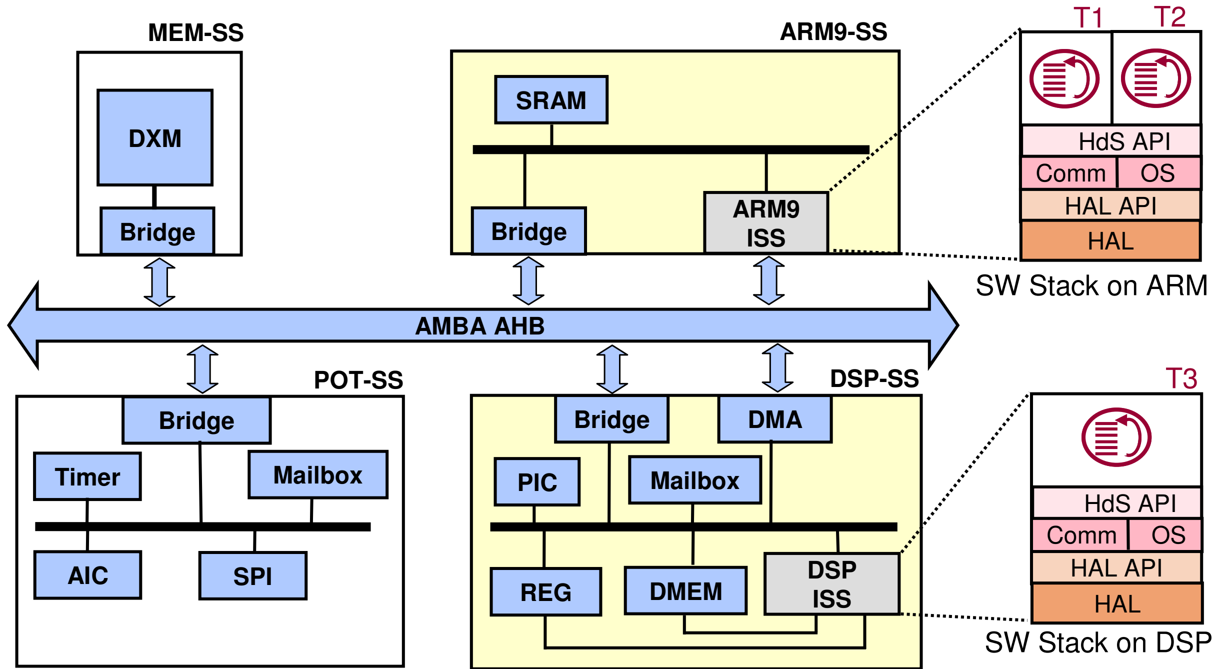


Figure 66. Global View of the Virtual Prototype for Diopsis RDT with AMBA Bus running Motion JPEG Decoder Application

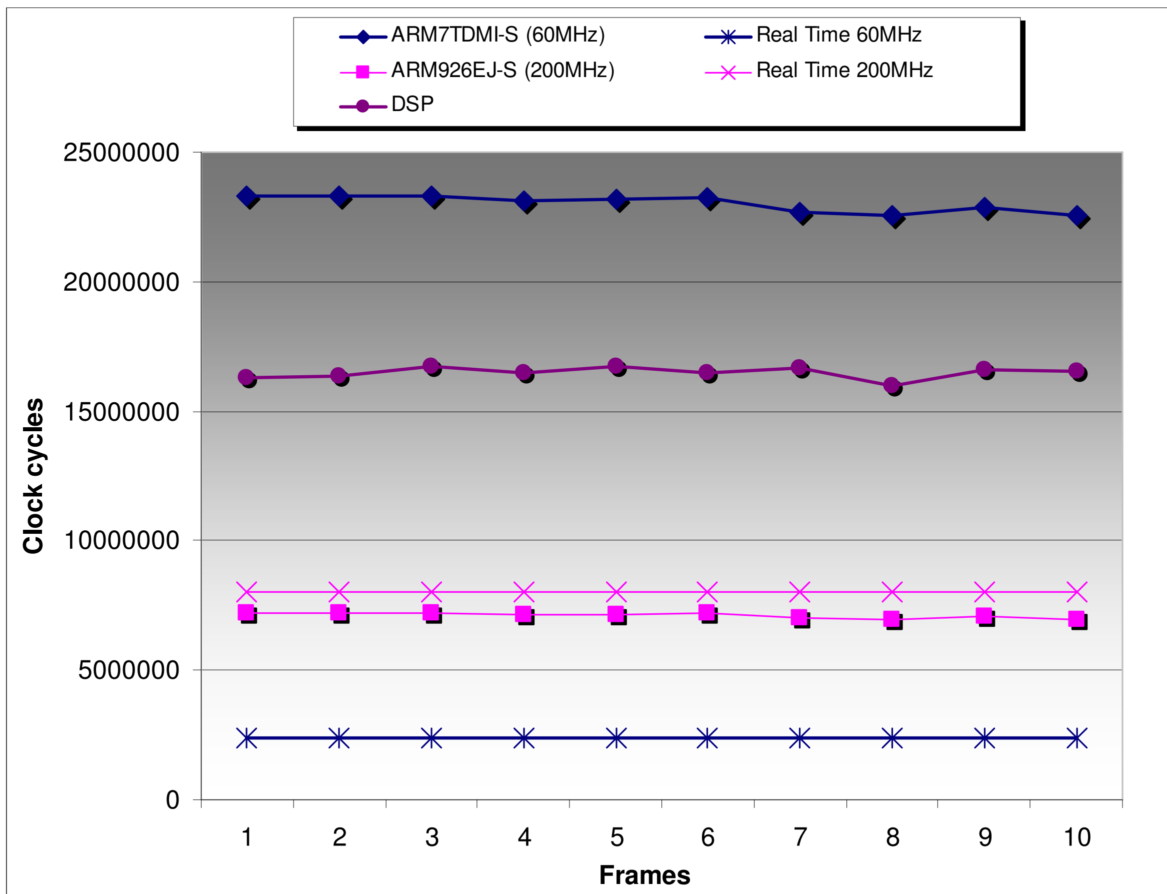


Figure 67. Execution Clock Cycles of Motion JPEG Decoder QVGA

As shown in figure 67, the number of execution cycles required to decode an image is approximately 7 Mega cycles on the ARM9 processor, 16 Mega cycles on the DSP processor and 23 Mega cycles on the ARM7 processor.

The performance difference between the two ARM processors is explained by the availability of the additional cache memories and improvement in number of cycles required for load/store operations characterizing the ARM9 core family compared with the ARM7 core. The real time requirement implies 8 Mega cycles on a CPU running at 200 MHz and 2,4 Mega cycles on a CPU running at 60 MHz. Thus, the M-JPEG decoder can be executed in real-time by using the ARM9 processor, while the execution on a single ARM7 processor requires application code optimization. The execution on the DSP can be improved by using the DSP specific optimization features.

After the compilation of the MJPEG decoder application, the memory requirements are as follows: 7592 bytes of code size for the program memory and 1402 bytes data memory. These values were obtained in case of targeting both processors ARM7 and ARM9 cores using the CodeWarrior development tool [Metrowerks]. In case of the DSP processor, the MJPEG decoder requires 614 bytes data memory and 2806 bytes program memory.

5.6.2. H.264 Application on Diopsis R2DT

The H.264 Encoder running on the Diopsis R2DT architecture at the virtual prototype level is illustrated in figure 67. In the same way as in the case of the Motion JPEG decoder, there are three final software stacks running on the architecture, one per each processor. The HAL libraries were included in the software stack for each particular CPU.

The hardware platform includes ISS to execute the final software. The ISS allows determining the execution cycles spent on each task. The virtual prototype of the Diopsis R2DT running the H.264 encoder application is illustrated in figure 68.

Figure 69 captures the results of executing the H.264 Encoder application, Main Profile, QCIF video resolution on the ARMTDMI-S and ARM926EJ-S processors. In this single task fashion, the H.264 encoder requires around 30 Mega cycles to encode a P frame and 16 Mega cycle for encoding one I video frame on the ARM9 CPU running at 200 MHz. If the target processor is the ARM7 core, the encoder requires approximately 50 Mega cycles for a frame type I and 90 Mega cycles for a frame type P.

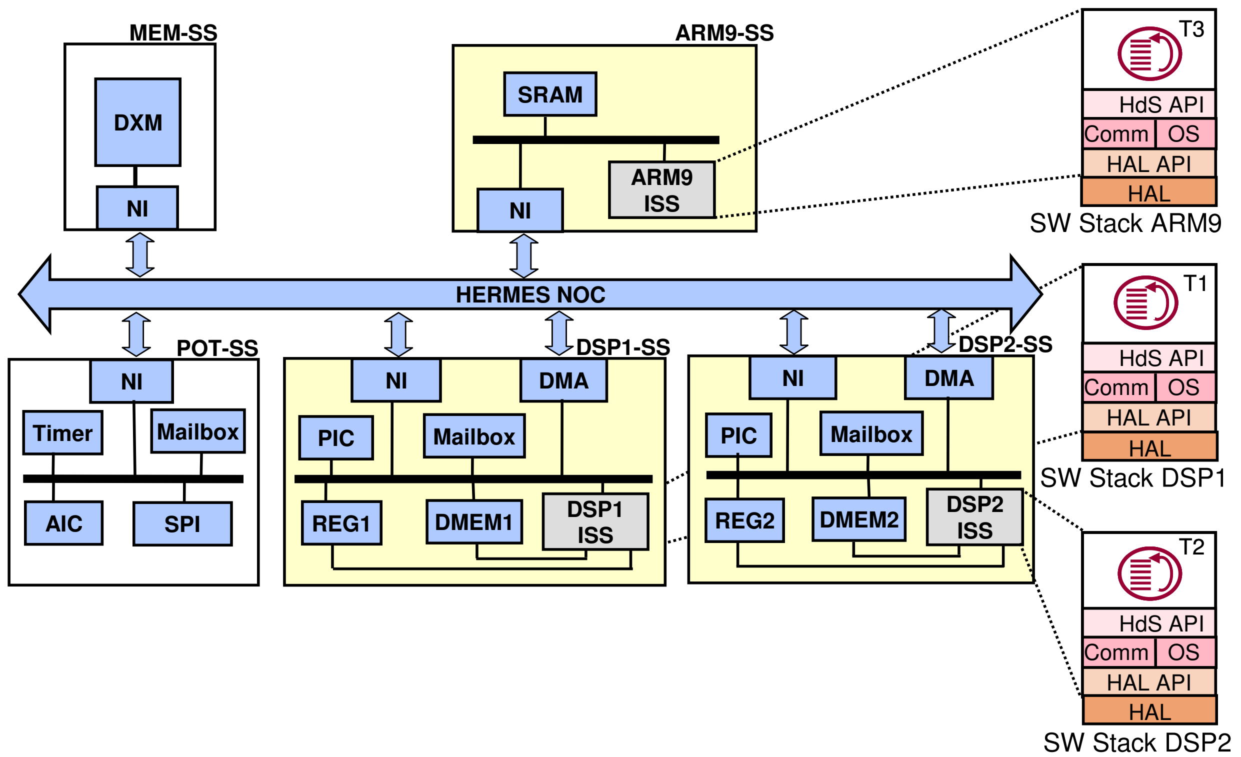


Figure 68. Global View of the Virtual Prototype for Diopsis R2DT with Hermes NoC running H.264 Encoder Application

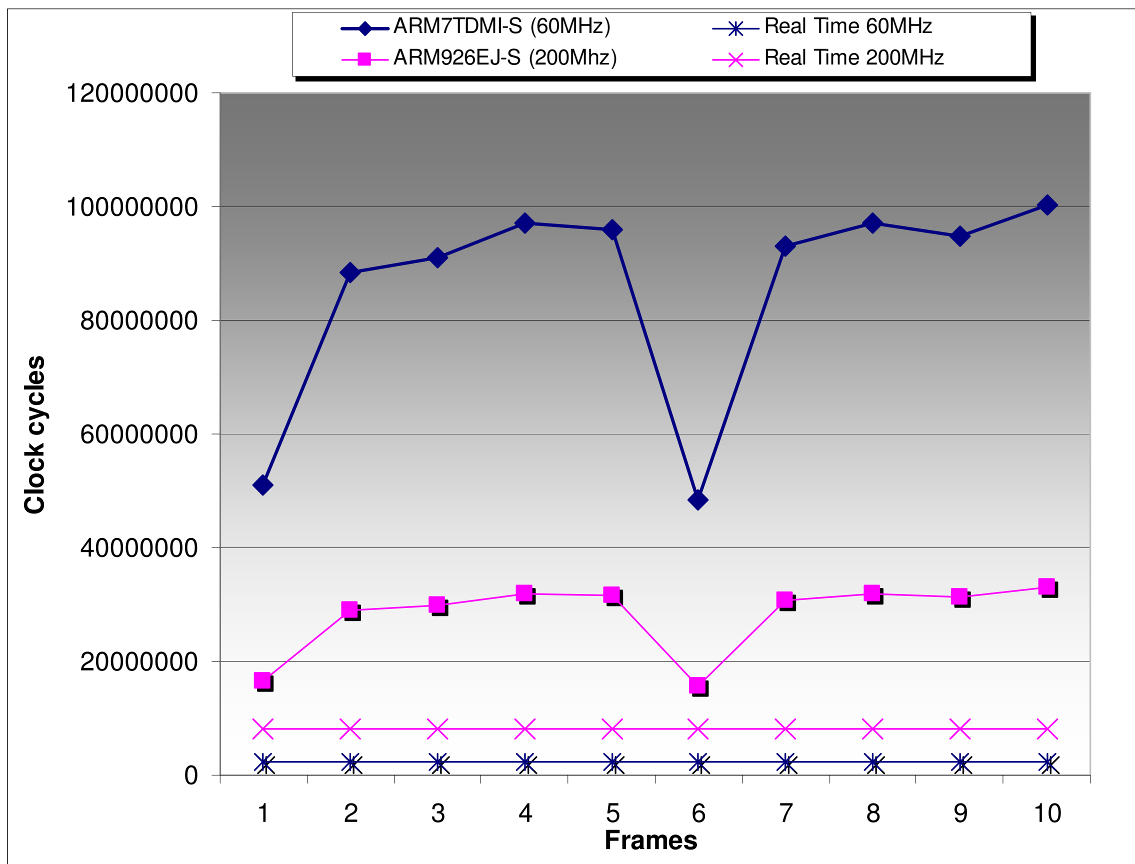


Figure 69. Execution Clock Cycles of H.264 Encoder, Main Profile, QCIF Video Format

As shown in figure 69, both results do not respect the real-time encoding requirement established at 25 frames/second. Running on a single processor, the achieved frame rate is 9 frames/second for a P frame and 12 frames/second for the I video frame. The H.264 results represented in figure 69 consider a key frame of 5 frames, which mean that between two I frame there are 5 video frames that will be encoded as P frame.

Figure 70 shows the program and code size of the H.264 application compiled with the CodeWarrior tool targeting the ARM7 and ARM9 processors. The data size has the same value for both types of processor cores, being equal with 2799 Kbytes, while the program size is 474 Kbytes compiled for the ARM7 processor and 277 Kbytes for the ARM9 core.

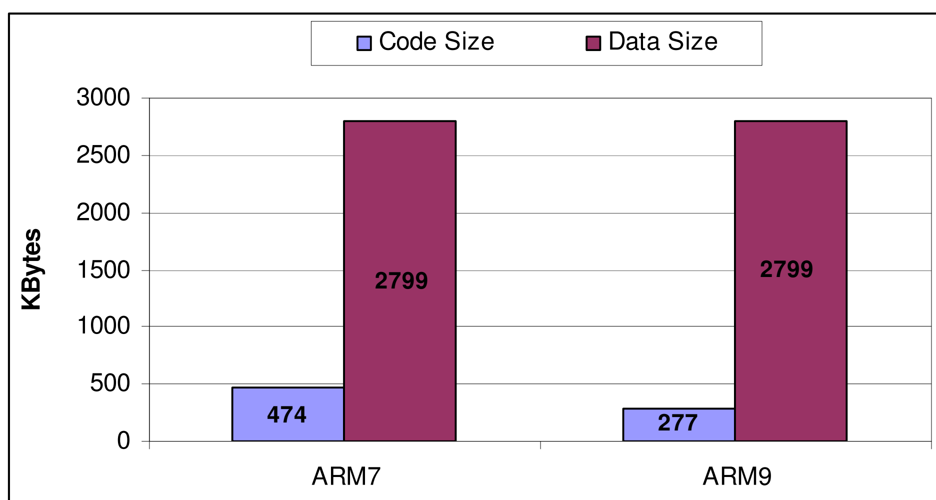


Figure 70. Program and Memory Size

5.7. State of the Art and Research Perspectives

5.7.1. State of the Art

Currently, virtual prototype environments for modeling and simulation based on SystemC, such as Maxsim [Arm-b], Coware ConvergenSC [Coware] and Synopsys System Studio [Synopsys], provide a rich set of components such as processors, memories, and peripherals that can be extended by user-defined modules.

The concept of Virtual Platform appears in [Hong 06] with the purpose to allow software development and code optimization before the real board is available. [Oya 07] uses virtual prototype simulation to perform a software profiling, such total execution cycles and software performance analysis.

Execution of software using an ISS still suffers from low simulation speed compared to real hardware. Therefore, many researchers focus on developing new techniques to attain high simulation speed. In this context, [Qin 06] mixes the interpreted ISS simulation with compiled ISS simulation in order to allow a multiprocessing simulation approach to increase simulation speed. [Sin 07] describes an ultra-fast ARM and multi-core DSP instruction set simulation environment based on just-in-time (JIT) translation technology, which refers to the dynamic translation of the target instructions (ARM, DSP) to the host instructions (x86) during the execution.

[Kra 07] presents a fast and hybrid simulation framework which allows switching between native code execution and ISS-based simulation. In this approach, the platform-independent parts of the software stack are executed directly on the host machine, while the platform dependent code executes upon an ISS. Thus, the framework allows debugging a complex application through executing it natively until the point where the bug is expected, and then executing on the ISS to examine the detailed software behavior.

Other research groups focus on integrating ISS within existing design flows. For instance, [Liu 98] presents a framework of ISS integration within the Ptolemy design environment that leverages the approach of time-approximate cosimulation based on source code estimation of execution time, and refines its precision by using an ISS.

[Park 07] presents the automatic generation of virtual execution platforms for the hardware architecture to analyze the run-time behavior of the application running on a real-time operating system and to estimate accurately performance data.

[Kli 06] allows generation of synthesizable communication from high level TLM communication models. The scope of this work is to reduce the gap between TLM and RTL design for automating MPSoC synthesis.

[Schir 08] proposes automatic software synthesis from TLM platforms. They support automatic generation of HdS, including code generation, communication software synthesis, multi-task synthesis and generation of the configuration and makefiles to control the cross compilation and linking of the generated code for a particular processor.

5.7.2. Research Perspectives

The virtual platform has to be available earlier than the real hardware in order to allow concurrent software and hardware design. Therefore, one research perspective regarding the

virtual prototype design relies on the automation of the generation process by using the architecture parameters annotating the system architecture model and the simulation results of the higher abstraction levels. The automatic generation of the virtual prototype model shortens the design time and permits to reduce human coding errors.

Virtual prototype uses instruction set simulators for the software execution. This implies high accuracy and low simulation speed. Moreover, the simulation time increase exponentially with the number of processor cores integrated on the same chip. Thus, finding new methodologies that speed up simulation, but still maintain accurate performance evaluation represents another important issue for future research perspectives.

Another important research perspective, related to the considered case studies, represents the simulation of the MJPEG and H.264 applications running on the multiprocessor architecture. As the target architectures include commercial off-the-shelf DSP processors and their compiler and instruction set simulator were provided as standalone applications, the integration of the DSP instruction set simulators into a hardware simulation environment, such as the one previously described in SystemC, represents an essential future perspective. Generally, the integration of ISS into an existing platform imposes development of a software simulation wrapper that interacts with the hardware model and solves the synchronization problem for the hardware and software interaction.

5.8. Conclusions

This chapter detailed the virtual prototype design. The virtual prototype design consisted of integrating the HAL component into the software stack, cross compiling it for the target processor and fixing the final memory mapping.

The validation of the software binary was performed by using Instruction Set Simulators (ISS). Thus, the Token Ring application was executed on the 1AX architecture, Motion JPEG on the Diopsis RDT architecture and H.264 Encoder on the Diopsis R2DT architecture.

The Motion JPEG application was also executed using ISS on a single processor (ARM7, ARM9 and DSP) and the H.264 Encoder was simulated using ISS running both on multiprocessor architecture with 3 ARM7 processors and single processor (ARM7 and

ARM9). The simulation of the virtual prototype model allows to validate the final software binary and the memory mapping.

Chapter 6

CONCLUSIONS AND FUTURE PERSPECTIVES

6.1. Conclusions

This thesis proposed a software design and validation flow able to efficiently use the resources of the architecture and allowing easy experimentation of several mappings of the application onto the platform resources. The thesis used Simulink environment to capture both application and architecture initial representations. The software generation and validation was performed gradually from this initial model corresponding to different software abstraction levels. Specific software development platforms (abstract models of the architecture) in SystemC were used to allow debugging the different software components with explicit hardware-software interaction.

The proposed software design flow decreases the complexity of the design process by structuring it into several layers. The different components of the software stack were generated and validated incrementally: the simulation at system architecture level validated the application's functionality; the virtual architecture level simulation allowed debugging the final application task code, the execution at transaction accurate architecture level validated the integration of the tasks code with the OS and communication library, while the virtual prototype enables the validation of the binary image.

Besides the software debug, the platforms also allowed to accurately estimate the use of the hardware resources by counting the total number of transactions exchanged during the simulation. The proposed software design flow made also possible to optimize the communication performance by using the architecture capabilities. The communication optimization relied on easy experimentation of different mappings of the communication onto the platform resources, using simple annotations of the initial Simulink model and generating the corresponding platforms.

Automatic tools were also developed to generate the hardware development platforms in SystemC for the 1AX architecture, Diopsis RDT and Diopsis R2DT architectures. Thus, the initial Simulink model is parsed and stored in an XML based format. The Simulink parser was developed using *lex/yacc* tools. Then, the virtual architecture and transaction accurate architecture platforms are generated automatically from the intermediate XML representation format. The generation makes use of a platform library at each abstraction level, which contains parameterized template hardware components. The automatic generation of the SystemC code of the hardware simulation models in case of the 1AX architecture, Diopsis

RDT and Diopsis R2DT architectures takes only few seconds from the Simulink model. More details about the automatic generation of the SystemC development platforms can be found in [Pop 07-b].

The flow is able to facilitate programming existing hardware platforms that contain heterogeneous multiprocessor architectures with specific I/O components. The design flow allows mapping sophisticated software organized into several stacks made of different layers on these platforms. The new software design flow masters the complexity of the software design process. This is achieved thanks to the incremental software layers generation and corresponding software development platforms generation. These platforms are able to abstract multimedia architectures at different abstraction levels and enable separate debug of the software components. The application of the proposed approach on the 1AX platform, the off shelf multimedia Diopsis RDT platform and Diopsis R2DT architecture allowed to demonstrate that the proposed software design flow enables efficient communication optimization in addition to efficient software debug.

Apart from the case studies presented in this document, the proposed programming environment has been applied successfully for the following multimedia applications running on the corresponding MPSoC architectures:

- ✓ Token Ring application targeting the 2A1X (2 ARM processors and 1 XTENSA processor interconnected through the AMBA bus) and Diopsis RDT architectures
- ✓ MP3 audio decoder running on the Diopsis RDT architecture with AMBA bus
- ✓ Vocoder audio encoder executed on the Diopsis RDT architecture with AMBA bus
- ✓ Motion JPEG image decoder running on the following architectures: 1AX, 2A1X, Diopsis RDT with NoC interconnect, Diopsis RDT with AMBA bus in normal mode without burst data transfers
- ✓ H.264 video encoder, main profile, running on the following architectures: Diopsis RDT with NoC, Diopsis RDT with AMBA bus with and without burst transfer and Diopsis R2DT with AMBA bus
- ✓ H.264 video decoder application, base profile, running on 1AX and 2A1X architectures

Moreover, the Motion JPEG application was also loaded and executed on the ARM 9 processor of the FPGA emulation platform of the Diopsis RDT architecture. Thus, in order to validate the correctness and efficiency of the proposed software design flow, the generated software stack of the MJPEG application was executed on a Diopsis emulation platform using Xilinx Virtex-II XC2V8000 FPGA provided by Atmel Inc. Since the HAL library to access the resources of the emulation platform was not yet fully available, the 4 tasks of the Motion JPEG application were mapped onto the ARM9 processor. Then, the software stack was designed and validated incrementally. The hardware simulation models were automatically generated from the input system architecture Simulink model. For the tasks management, the DwarfOS tiny in-house OS was used to implement basic OS services, such as tasks scheduling and software FIFO channels for the communication between the tasks. This OS was enriched to support specific context switch for the ARM9 processor. A Multi-Ice GDB debugger server was used in order to load the final software binary image on the local SDRAM memory. The FPGA platform based emulation ensured the reliability of the software code's functionality.

6.2. Future Perspectives

This thesis presented the complexity of software design and validation for heterogeneous MPSoC architectures with an initial formalization of the programming process.

Future research perspectives tackle the following described items:

i) *Automation of the software design and validation flow*

The automation of the software design flow concerns two aspects:

- automatic tools for the software stack construction
- automatic tools for the software simulation models generation

On the validation side, the automatic SystemC development platforms generator tools assume only a subset of subsystems and interconnect schemes. Extending these tools to support general architectures and different software components remains as future work.

The automatic generation of the different MPSoC abstraction levels could be made possible by applying a service-based modeling of the hardware-software interface as

described in [Ger 07]. The composition of the services allows the automatic generation tools to build easily the different software and hardware simulation models.

The automatic generation of the hardware and software architectures at the different abstraction levels shortens the design time and permits to reduce human coding errors.

ii) Automatic generation of the RTL hardware architectures

This thesis assumed examples of existing fixed hardware architecture. It made use of abstraction models of the target architecture at the different TLM abstraction levels to allow the software validation.

The automatic generation of RTL to allow synthesis of the target hardware architectures represents future work. This would make possible hardware design in parallel with the software design for a specific application, allowing hardware implementation of some functions for a target application.

iii) Formalization of the hardware-software partitioning process

The proposed software design and validation flow uses system architecture model, which represents the partitioned model of the application onto the target architecture. Thus, formalizing the partitioning process represents another future perspective to allow early design space exploration. Design space exploration represents an essential issue to analyze the impact on performances by using different application partitioning, mapping and communication schemes. Design space exploration allows finding the best combination of the application/architecture configurations to achieve the required communication and computation constraints. Future works focus on better parallelization of the applications and exploration of the different partitioning and mapping combinations.

Automatic tools for application partitioning, mapping, and evaluation metrics such as performance, power, and cost are necessary to fully explore the design space and help designer's choices. Therefore, future work should address estimation tools such as power estimation to meet the tight power constraints on MPSoCs. For instance, power estimation can be implemented by embedding cycle-accurate power model into each hardware component of the development platform.

Another aspect of future perspective for design space exploration constitutes the annotation of the intermediate abstraction platforms with execution delays to provide more accurate performance estimation at the design steps earlier than the virtual prototype design.

iv) Support of multiple applications

This thesis presented the problems met in programming MPSoC that runs single application. The support of the multiple applications running on the same MPSoC architecture, i.e. an audio encoder combined with a video encoder application is also envisioned for the future. During the parallel execution of the multiple applications, the main difficulties that need to be overcome are related to the global scheduling and hardware resource sharing of the different applications.

v) Hardware-software co-design flow

A completely automated hardware-software co-design flow represents another future research perspective. The flow involves a seamless refinement at the four abstraction levels (system architecture, virtual architecture, transaction accurate architecture, virtual prototype). It requires automatic code generators for the software design, platform based generators for the hardware design and automatic hardware-software interfaces refinement.

References

[Ambric] AM2000 Processor Array Family, http://www.ambric.com

[Arm] Technical documentation of ARM7 and ARM9 processors, AMBA Bus Technical Specification, RealView Compilation Tools Linker and Utilities Guide, Embedded Software Development with ADS v1.2, http://www.arm.com

[Arm-b] Technical documentation of ARM MaxSim http://www.arm.com

[Asc 05] G. Ascia, V. Catania, M. Palesi "Mapping Cores on Network-on-Chip", *International Journal of Computation Intelligence Research*, Vol. 1, No. 2, 2005, pp. 109-126

[Atm] mAgicV VLIW DSP and Diopsis http://www.atmelroma.it

[Bac 05] I. Bacivarov, A. Bouchhima, S. Yoo, A.A. Jerraya "ChronoSym: a new approach for fast and accurate SoC cosimulation", *International Journal on Embedded Systems (IJES)*, Volume 1, Issue 1, 2005, pp. 103-111

[Bac 06] I. Bacivarov "Evaluation des performances pour les systèmes embarques heterogenes, multiprocesseurs monopuces", *Thèse de Doctorat INPG*, TIMA Laboratory, 2006

[Bal 06] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, S. Neema "Developing Applications Using Model-Driven Design Environments", *IEEE Computer Society* 39(2), 2006, pp. 33-40

[Bel 06] G. Beltrame, D. Sciuto, C. Silvano, P. Paulin, E. Bensoudane "An Application Mapping Methodology and Case Study for Multi-Processor On-Chip Architectures", Proceeding of VLSI-SoC 2006, 16-18 October 2006, Nice, France, pp. 146-151

[Ben 02] L. Benini, G. De Micheli "Networks on Chips: a new SoC paradigm", IEEE Computer, Vol. 35(1), 2002, pp. 70-78

[Bert 02] C. Berthet "Going Mobile: The Next Horizon for Multi-million Gate Designs in the Semi-Conductor Industry", *Proceeding of DAC 2002*, 10-14 June 2002, New Orleans, USA, pp. 375-378

[Ber 04] Friedbert Berens "Algorithm to System-on-Chip Design Flow that Leverages System Studio and SystemC 2.0.1", *The Synopsys Verification Avenue Technical Bulletin,* Volume 4, Issue 2, May 2004

[Ble 03] S. Bleuler, M. Laumanns, L. Thiele, E. Zitzler "PISA- A Platform and Programming Language Independent Interface for Search Algorithms",

Evolutionary Multi-Criterion Optimization (EMO 2003), Volume 2632/2003 of LNCS, Springer, pp. 494-508

[Bon 06] M. Bonaciu "Plateforme flexible pour l'exploitation d'algorithmes et d'architectures en vue de réalisation d'application vidéo haute définition sur des architectures multiprocesseurs mono puces", *Thèse de Doctorat INPG*, TIMA Laboratory, 2006

[Bou 04] A. Bouchhima, S. Yoo, A.A. Jerraya "Fast and Accurate Timed Execution of High Level Embedded Software Using HW/SW Interface Simulation Model", *Proceeding of ASP-DAC 2004*, January 2004, Yokohama, Japan, pp. 469-474

[Bou 05] A. Bouchhima, X. Chen, F. Petrot, W.O. Cesario, A.A. Jerraya "A Unified HW/SW interface model to remove discontinuities between HW and SW design", *Proceeding of EMSOFT'05*, 18-22 September 2005, New Jersey, USA, pp. 159-163

[Bou 05-b] A. Bouchhima, I. Bacivarov, W. Youssef, M. Bonaciu, A.A. Jerraya "Using Abstract CPU Subsystem Simulation Model for High Level HW/SW Architecture Exploration", *Proceeding of ASP-DAC 2005*, 18-21 January 2005, Shanghai, China, pp. 969-972

[Buc 92] J. Buck, S. Ha, E. Lee, D. Messerschmitt. "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems", *International Journal of Computer Simulation*, v. 4, pp. 155-182.

[Bus 06] G. Busonera, S. Carta, A. Marongiu, L. Raffo "Automatic Application Partitioning on FPGA/CPU Systems Based on Detailed Low-Level Information", *Proceeding of the 9th EUROMICRO Conference on Digital System Design*, 30 August – 1 September 2006, Croatia, pp. 265-268

[But 97] D.R. Butenhof "Programming with POSIX Threads", Addison Wesley, May, 1997, ISBN 0201633922

[Car] Open Systems Glossary of Software Engineering Institute, Carnegie Mellon, http://www.sei.cmu.edu/opensystems/glossary.html

[Ces 02] W.O. Cesario, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, L. Gauthier, M. Diaz-Nava, A.A. Jerraya "Multiprocessor SoC Platforms: A Component-Based Design Approach", IEEE Design & Test of Computers, Volume 19, Nr. 6, November-December 2002, pp. 52-63

[Cha 00] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, J. McDonald "Parallel Programming in OpenMP", Morgan Kaufmann, October 2000, ISBN 9781558606715

[Chak 03] S. Chakraborty, S. Kunzli, L. Thiele, A. Herkersdorf, P. Sagmeister "Performance evaluation of network processor architectures: combining simulation with analytical estimation", *Computer Networks: The International Journal of Computer and Telecommunications Networking*, Volume 41, Issue 5, April 2003, pp. 641-665

[Chen 05] K. Chen, J. Sztipanovits, S. Neema "Toward A Semantic Anchoring Infrastructure for Domain-Specific Modeling Languages", Proceeding of EMSOFT 2005, 19-22 September 2005, New Jersey, USA, pp. 35-43

[ChenJ 06] Jian-Wen Chen, Chao-Yang Kao, Youn-Long Lin "Introduction to H.264 Advanced Video Coding", *Proceeding of ASP-DAC 2006*, 24-27 January 2006, Yokohama, Japan, pp. 736-741

[Cho 05] Y. Cho, S. Yoo, K. Choi, N.E. Zergainoh, A.A. Jerraya "Scheduler implementation in MPSoC Design", *Proceeding of ASP-DAC 2005*, 18-21 January 2005, Shanghai, China, pp. 151-156

[Coo 69] J. Cooley, P. Lewis, and P. Welch (1969). "The finite Fourier transform", *IEEE Trans. Audio Electroacoustics* 17 (2), pp. 77-85

[Coware] ConvergenSC http://www.coware.com

[Cul 98] D.Culler, J.P. Singh, A. Gupta "Parallel Computer Architecture: A Hardware/Software Approach", Morgan Kaufmann, August 1998, ISBN 1558603433

[Des 02] G. Desoli et al. "A New Facility for Dynamic Control of Program Execution: DELI", *Proceeding of EMSOFT 2002*, Grenoble, France

[Dio] Diopsis D940, http://www.atmel.com

[Erb 07] Cagkan Erbes, Andy D. Pimentel, Mark Thompson, Simon Polstra "A Framework for System-Level Modeling and Simulation of Embedded Systems Architecture", *EURASIP Journal on Embedded Systems*, Volume 2007, Article ID 82123, June 2007

[Fei 02] Y. Fei, N.K. Ha "Functional Partitioning for Low Power Distributed Systems of Systems-on-a-Chip", *Proceeding of ASP-DAC 2002*, 7-11 January 2002, Bangalore, India

[Fla 07] P. Flake, F. Schirrmeister "MPSoC demands system level design automation", *EDA Tech Forum*, Volume 4, Issue 1, March 2007, pp. 10-11

[Gaj 00] D.D. Gajski "SpecC: Specification Language and Design Methodology", 2000, Kluwer

[Ger 07] P. Gerin, H. Shen, A. Chureau, A. Bouchhima, A.A. Jerraya "Flexible and Executable Hardware/Software Interface Modeling for Multiprocessor SoC Design Using SystemC", Proceeding of ASP-DAC 2007, pp. 390-395

[Gerst 05] A. Gerstlauer, D. Shin, R. Domer, D.D. Gajski "System-Level Communication Modelling for Network-on-Chip Synthesis", *Proceeding of ASP-DAC 2005*, 18-21 January 2005, Shanghai, China, pp. 45-48

[Gil 04] F. Gilliers, F. Kordon, D. Regep "A Model Based Development Approach for Distributed Embedded Systems", *Proceeding of RISSEF 2002*, pp. 137-151, 2002

[Gla 94] C. Glass, L. Ni "The turn model for adaptive routing", *Journal of ACM*, 41(5), 1994, pp. 278- 287

[Gnu] GNU tools and documentation, http://www.gnu.org

[Gro 02] T. Grotker, S. Liao, G. Martin, S. Swan "System Design with SystemC", Kluwer, 2002, ISBN 1402070721

[Gue 07] X. Guerin, K. Popovici, W. Youssef, F. Rousseau, A. Jerraya "Flexible Application Software Generation for Heterogeneous Multi-Processor System-on-Chip", *Proceeding of COMPSAC 2007*, 23-27 July 2007, Beijing, China

[Ha 06] S. Ha, C. Lee, Y. Yi, S. Kwon, Y.P. Joo "Hardware-Software Codesign of Multimedia Embedded Systems: the PeaCE", *Proceeding of 12th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*, 2006, pp. 207-214

[Ha 07] S. Ha "Model-based Programming Environment of Embedded Software for MPSoC", *Proceeding of ASP-DAC'07*, 23-26 January 2007, Yokohama, Japan, pp. 330-335

[Han 06] S.I. Han et al. "Buffer memory optimization for video codec application modeled in Simulink", *Proceeding of DAC 2006*, San Francisco, USA, pp. 689-694

[Has 05] M. AbdElSalam Hassan, Keishi Sakanushi, Yoshinori Takeuchi, Masaharu Imai "RTK-Spec TRON: A Simulation Model of an ITRON Based RTOS Kernel in SystemC", *Proceeding of DATE 2005*, 7-11 March 2005, Munich, Germany, pp. 554-559

[Has 06] M. AbdElSalam Hassan, Masaharu Imai "A System Level Modeling Methodology for RTOS Centric Embedded Systems", Proceeding of 14th IFIP VLSI-SoC, PhD Forum Digest of Papers, 16-18 October 2006, Nice, France, pp. 62-67

[Hen 03] J.L. Hennessy, D.A. Patterson "Computer Architecture: A Quantitative Approach", Third Edition, 2003, ISBN 1558605967, Printed by Elsevier Science Pte Ltd.

[Hong 06] Sungpack Hong, Sungjoo Yoo, Sheayun Lee, Sangwoo Lee, Hye Jeong Nam, Bum-Seok Yoo, Jaehyung Hwang, Donghyun Song, Janghwan Kim, Jeongeun Kim, HoonSang Jin, Kyu-Myung Choi, Jeong-Taek Kong, Sookwan Eo "Creation and Utilization of a Virtual Platform for Embedded Software Optimization: An Industrial Case Study", *Proceeding of CODES+ISSS 2006*, Seoul, Korea, pp. 235-240

[Hwang 06] Hyeyoung Hwang, Taewook Oh, Hyunuk Jung, Soonhoi Ha "Conversion of Reference C Code to Dataflow Model: H.264 Encoder Case Study", *Proceeding of ASP-DAC 2006*, 24-27 January 2006, Yokohama, Japan, pp. 152-157

[Kan 06] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hannikainen, T.D. Hamalainen, J. Riihimaki, K. Kuusilinna "UML-based multiprocessor SoC

design framework", *ACM Transactions on Embedded Computing Systems (TECS)*, Volume 5, Issue 2, 2006, pp. 281-320

[Kempf 05] T. Kempf, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, B. Vanthournout "A Modular Simulation Framework for Spatial and Temporal Task Mapping onto Multi-Processor SoC Platforms", *Proceeding of DATE 2005*, 7-11 March 2005, Munich, Germany

[Kie 02] Bart Kienhuis, Ed F. Deprettere, P. van der Wolf, Kees A. Vissers "A methodology to design programmable embedded systems- the Y-Chart approach", *Lectures Notes in Computer Science, Volume 2268, Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation–SAMOS 2002*, Springer, pp. 18-37

[Kli 06] W. Klingauf, H. Gadke, R. Gunzel "TRAIN: A Virtual Transaction Layer Architecture for TLM-based HW/SW Codesign of Synthesizable MPSoC", *Proceeding of DATE 2006*, 6-10 March 2006, Munich, Germany, pp. 1318-1323

[Kli 07] W. Klingauf, R. Gunzel, C. Schroder "Embedded Software Development on Top of Transaction-Level Models", *Proceeding of CODES+ISSS 2007*, 30 September-3 October 2007, Salzburg, Austria, pp. 27-32

[Koc 00] E.A. de Kock et al. "Yapi: Application modeling for signal processing systems", *Proceeding of DAC 2000*, USA, pp.402-405

[Kog 01] T. Kogel, A. Wieferink, H. Meyr, A. Kroll "SystemC based architecture exploration of a 3D graphic processor", *Proceeding of IEEE Workshop on Signal Processing Systems*, 26-28 September 2001, Antwerp, Belgium, pp. 169-176

[Kra 07] S. Kramer, L. Gao, J. Weinstock, R. Leupers, G. Ascheid, H. Meyr "HySim: A Fast Simulation Framework for Embedded Software Development", *Proceeding of CODES+ISSS 2007*, 30 September – 5 October 2007, Salzburg, Austria

[Kun 06] S. Kunzli, F. Poletti, L. Benini, L. Thiele "Combining simulation and formal methods for system-level performance analysis", *Proceeding of DATE 2006*, 6-10 March 2006, Munich, Germany, pp. 236-241

[Kwo 04] S. Kwon, H. Jung, S. Ha "H.264 decoder algorithm specification and simulation in Simulink and PeaCE", *Proceeding of ISOCC 2004*, Seoul, Korea

[Jer 05] A. Jerraya, W. Wolf "Hardware-Software Interface Codesign for Embedded Systems", *Computer*, Volume 38, No.2, February 2005, pp.63-69

[Jer 06] A. Jerraya, A. Bouchhima, F. Petrot "Programming models and HW-SW Interfaces abstraction for Multi-Processor SoC", *Proceeding of DAC 2006*, San Francisco, USA, pp. 280-285

[Lav 06] D. Lavenier, M. Daumas "Architectures des Ordinateurs", *Technique et Science Informatique*, 25(6), 2006

[Lie 01] P. Lieverse, T. Stefanov, P. van der Wolf, E. Deprettere "System level design with SPADE: an M-JPEG case study", *Proceeding of ICCAD 2001*, 4-8 November 2001, San Jose, USA, pp. 31-38

[Liu 98] J. Liu, M. Lajolo, A. Sangiovanni-Vincentelli "Software timing analysis using HW/SW cosimulation and instruction set simulator", *Proceeding of the 6th International Workshop on Hardware/Software Co-design CODES/CASHE'98*, 15-18 March 1998, Seattle, Washington, pp. 65-69

[Matlab] The MathWorks Inc., http://www.mathworks.com

[Mag 05] D.P. Magee "Matlab Extensions for the Development, Testing and Verification of Real-Time DSP Software", *Proceeding of DAC 2005*, Anaheim, USA

[Mar 06] Grant Martin "Overview of the MPSoC Design Challenge", *Proceeding of DAC 2006*, 24-28 July 2006, San Francisco, USA, pp. 274-279

[Mat 01] J. Mattioli, N. Museux, J. Jourdan, P. Saveant, S. de Givry "A Constraint Optimization Framework for Mapping a Digital Signal Processing Application onto a Parallel Architecture", *Proceeding of the 7th International Conference on Principles and Practice of Constraint Programming*, 2001, pp. 701-715

[Mei 07] S. Meijer, J. Walters, D. Snuijf, B. Kienhuis "Automatic partitioning and mapping of stream-based applications onto the Intel IXP Network Processor", *Proceeding of Workshop on Software & Compilers for Embedded Systems (SCOPES'07)*, Nice, 20 April 2007

[Metrowerks] CodeWarrior Development tools, http://www.metrowerks.com

[Mey 06] H. Meyr "Application Specific Processors (ASIP): On design and implementation Efficiency", *Proceeding of SASIM 2006*, Nagoya, Japan

[Mic 02] G. de Micheli, R. Ernst, W. Wolf "Readings in Hardware/Software Co-design", Morgan Kaufmann, 2002, ISBN 1558607021

[Mod] Model driven architecture http://www.omg.org/mda/

[Moh 98] P. Mohapatra et al. "Wormhole routing techniques for directly connected multicomputer systems", *ACM Computing Survey*, Vol. 30(3), 1998, pp. 374-410

[Mor 04] F. Moraes et al. "HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks-on-Chip Integration", *VLSI Journal*, v38(1), 2004, pp. 69-93

[MPI] MPI http://www-unix.mcs.anl.gov/mpi

[Nic 02] G. Nicolescu, "Specification et validation des systemes heterogenes embarques", *PhD Thesis*, TIMA Laboratory, 2002

[Nik 06] H. Nikolov, T. Stefanov, E. Deprettere "Multi-processor System Design with ESPAM", *Proceeding of CODES+ISSS'06*, 22-25 October 2006, Seoul, Korea, pp. 211-216

[Nex] Nexperia http://www.nxp.com

[Nom] Nomadik, http://www.st.com

[OSCI] Open SystemC Initiative (OSCI) http://www.systemc.org

[Oya 07] M. Oyamada, F.R. Wagner, M. Bonaciu, W. Cesario, A. Jerraya "Software Performance Estimation in MPSoC Design", *Proceeding of ASP-DAC'07*, 23-26 January 2007, Yokohama, Japan, pp. 38-43

[Park 07] S. Park, W. Olds, K.G. Shin, S. Wang "Integrating Virtual Execution Platform for Accurate Analysis in Distributed Real-Time Control System Development", *Proceeding of RTSS 2007*, 3-6 December 2007, Tucson, Arizona, USA

[Pao 06] Pier S. Paolucci, Ahmed A. Jerraya, Rainer Leupers, Lothar Thiele, Piero Vicini "SHAPES : a tiled scalable software hardware architecture platform for embedded systems", *Proceeding of CODES+ISSS 2006*, Seoul, Korea, pp. 167-172

[Pau 06] P. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, O. Benny, B. Laviguer, D. Lo, G. Beltrame, V. Gagne, G. Nicolaescu "Parallel programming models for a multi-processor SoC platform applied to networking and multimedia", *IEEE Transactions on VLSI Journal*, 2006

[Paz 04] Nuria Pazos, Alexander Maxiaguine, Paolo Ienne, Yusuf Leblebici "Parallel modelling paradigm in multimedia applications: Mapping and scheduling onto a multi-processor system-on-chip platform", *Proceedings of the International Global Signal Processing Conference*, Santa Clara, California, USA, September 2004

[Pop 07] K. Popovici, A.A. Jerraya "Simulink based Hardware-Software Codesign Flow for Heterogeneous MPSoC", *Proceeding of Summer Computer Simulation Conference (SCSC'07)*, 15-18 July 2007, San Diego, USA, pp. 497-504

[Pop 07-b] K. Popovici, X. Guerin, F. Rousseau, P.S. Paolucci, A. Jerraya "Efficient Software Development Platforms for Multimedia Applications at Different Abstraction Levels", *Proceeding of RSP 2007*, 28-30 May, 2007, Porto Alegre, Brazil

[Pop 08] K. Popovici, X. Guerin, F. Rousseau, P.S. Paolucci, A. Jerraya "Platform based Software Design Flow for Heterogeneous MPSoC", *ACM Journal: Transactions on Embedded Computing Systems (TECS), Special Issue on Rapid System Prototyping*, Accepted 17 January 2008

[Pos 03] Frank Pospiech "Hardware dependent Software (HdS). Multiprocessor SoC Aspects. An Introduction", *MPSoC 2003*, 7-11 July 2003, Chamonix, France

[Pul 07] A. Pullini, F. Angiolini, P. Meloni, D. Atienza, S. Murali, L. Raffo, Giovanni. De Michelli, L. Benini " NoC Design and Implementation in 65nm Technology", *Proceeding of the 1st Internal Symposium on Networks-on-Chip*, 7-9 May 2007, Princeton, New Jersey, USA, pp. 273-282

[Rey 01] L.M. Reyneri, F. Cucinotta, A. Serra, L. Lavagno "A hardware-software codesign flow and IP library based on Simulink", *Proceeding of the 38th conference on Design Automation*, Las Vegas, United States, 2001, pp. 593-598

[Ric] I. Richardson, G.J. Sullivan "H264 and MPEG-4 Video Compression"

[Roa 07] Jeff Roane "Electronic system level design for embedded systems", *EDA Tech Forum*, Volume 4, Issue 1, March 2007, pp. 14-16

[Row 94] J.A. Rowson "Hardware/Software cosimulation*", Proceeding of DAC 1994,* San Diego, USA, pp. 439-440

[Qin 06] W. Qin, J. D'Errico, X. Zhu "A Multiprocessing Approach to Accelerate Retargetable and Portable Dynamic-compiled Instruction-set Simulation", *Proceeding of CODES+ISSS 2006*, 22-25 October 2006, Seoul, Korea, pp. 193-198

[Schir 07] Gunar Schirner, Andreas Gertslauer, Rainer Domer "Abstract, Multifaced Modeling of Embedded Processors for System Level Design", *Proceeding of ASP-DAC 2007*, 23-26 January 2007, Yokohama, Japan, pp. 384-389

[Schir 08] Gunar Schirner, Andreas Gertslauer, Rainer Domer "Automatic Generation of Hardware dependent Software for MPSoCs from Abstract System Specifications", *Proceeding of ASP-DAC 2008*, 21-24 January 2008, Seoul, Korea

[Sem 00] L. Semeria, A. Ghosh "Methodology for hardware/software co-verification in C/C++", *Proceeding of ASP-DAC 2000,* Yokohama, Japan, pp. 405-408

[Shapes] Shapes (Scalable Software Hardware Architecture Platform for Embedded Systems) European Project, http://shapes-p.org

[Shin 04] Dongwan Shin, Samar Abdi, D.D. Gajski "Automatic Generation of Bus Functional Models from Transaction Level Models", *Proceeding of ASP-DAC 2004*, 27-30 January 2004, Yokohama, Japan, pp. 756-758

[Shin 06] Dongwan Shin, Andreas Gertslauer, Junyu Peng, Rainer Domer, D.D. Gajski "Automatic Generation of Transaction-Level Models for Rapid Design Space Exploration", *Proceeding of CODES+ISSS 2006*, Seoul, Korea, pp. 64-69

[Sin 07] S. Singhai, M.Y. Ko, S. Jinturkar, M. Moudgill, J. Glossner "An Integrated ARM and Multi-core DSP Simulator", *Proceeding of CASES'07,* 30 September-3 October 2007, Salzburg, Austria, pp. 33-37

[Spirit] Spirit IP-XACT, http://www.spiritconsortium.com

[Synopsys] Synopsys System Studio http://www.synopsys.com

[Tan 95] Andrew S. Tanenbaum "Distributed operating systems", 1995, Prentice-Hall, ISBN 0132199084

[Tan 97] Andrew S. Tanenbaum, Albert S. Woodhull "Operating Systems: Design and Implementation", 1997, Prentice-Hall, ISBN 0136386776

[Tan 99] Andrew S. Tanenbaum "Structured Computer Organization", 1999, Prentice-Hall, ISBN 013219901

[Ten] Xtensa processor architecture, XPRES Compiler http://www.tensilica.com

[Thi 07] L. Thiele, I. Bacivarov, W. Haid, K. Huang "Mapping Applications to Tiled Multiprocessor Systems", *Proceeding of Seventh International Conference on Application of Concurrency to System Design (ACSD 2007)*, 10-13 July 2007, Bratislava, Slovak Republic, pp. 29-40

[Tho 07] M. Thompson, H. Nikolov, T. Stefanov, A.D. Pimentel, C. Erbas, S. Polstra, E.F. Deprettere "A Framework for Rapid System-level Exploration, Synthesis, and Programming of Multimedia MP-SoCs", *Proceeding of CODES+ISSS 2007*, 30 September-3 October 2007, Salzburg, Austria, pp. 9-14

[TI] TI OMAP, http://www.omap.com

[Tilera] Tile64 Processor Family, http://www.tilera.com

[Tur 05] J. Turley,"Survey says: Software tools more important than Chips", *Embedded Systems Design Journal*, 4-11-2005

[Van 04] P. Van der Wolf et al. "Design and programming of embedded multiprocessors: an interface-centric approach", *Proceeding of CODES+ISSS 2004*, Stockholm, Sweden, pp. 206-217

[Vand 06] Y. Vanderperren and W. Dehaene, "From UML/SysML to Matlab/Simulink: Current State and Future Perspectives", *Proceeding of Design Automation and Test in Europe, DATE 2006*, 6-10 March, Munich, Germany, pp. 93-93

[Ven 05] N. Ventroux, F. Blanc, D. Lavenier "A Low Complex Scheduling Algorithm for Multi-processor System-on-Chip", *Proceeding of Parallel and Distributed Computing and Networks*, 15-17 February 2005, Innsbruck, Austria

[Ver 07] S. Verdoolaege, H. Nikolov, T. Stefanov "PN: A Tool for Improved Derivation of Process Networks", *EURASIP Journal on Embedded Systems*, Volume 2007, Article ID 75947

[Verg 05] T. Vergnaud, L. Pautet, F. Kordon "Using the AADL to Describe Distributed Applications from Middleware to Software Components", *Proceeding of Ada-Europe 2005*, York, UK, 20-24 June 2005, pp. 67-78

[Vin 01] A. Sangiovanni-Vincetelli, G. Martin "Platform-Based Design and Software Design Methodology for Embedded Systems" *IEEE Design and Test* v.18, n.6, pp. 23-33, 2001

[Vin 04] A. Sangiovanni-Vincetelli, et al. "Benefits and Challenges for Platform-Based Design", *Proceeding of DAC 2004*, USA

[Wal 91] G.K. Wallace "The JPEG Still Picture Compression Standard", *Communications of the ACM, Special Issue on Digital Multimedia Systems*, Vol.34(4), April 1991, pp. 30-44

[Wol 06] W. Wolf "High-Performance Embedded Computing", Morgan Kaufmann, 2006

[X264] h264 open source code, http://www.videolan.org/developers/x264.html

[Xue 06] L. Xue, O. Ozturk, F. Li, M. Kandemir, I. Kolcu "Dynamic Partitioning of Processing and Memory Resources in Embedded MPSoC Architectures", *Proceeding of DATE 2006*, 6-10 March 2006, Munich, Germany, pp. 690-695

[Yoo 03] Sungjoo Yoo, A.A. Jerrara "Introduction to Hardware Abstraction Layers for SoC", *Proceeding of DATE 2003*, 3-7 March 2003, Munich, Germany, pp. 336-337

[You 04] M.W. Youssef, S. Yoo, A. Sasongko, Y. Paviot, A. Jerraya "Debugging HW/SW Interface for MPSoC: Video Encoder System Design Case Study", *Proceeding of DAC 2004*, 7-11 June 2004, San Diego, USA, pp. 908-913

Publications

Book Chapters:

K. POPOVICI, W. O. CESARIO, F. WAGNER, A. JERRAYA "Hardware-Software Interfaces Design for SoC", *Chapter in "Networked Embedded Systems",* Ed. CRC Press, to appear 2008

K. POPOVICI, A. JERRAYA "Programming Models for MPSoC", *Chapter 4 in "Model Based Design of Heterogeneous Embedded Systems",* Ed. CRC Press, to appear 2008

K. POPOVICI, A. JERRAYA "Hardware Abstraction Layer – Introduction and Overview", *Chapter 3 in "Hardware dependent Software, Concept, Tools and Applications",* Ed. Springer, to appear 2008

International Journals:

K. POPOVICI, X. GUERIN, F. ROUSSEAU, P.S. PAOLUCCI, A. JERRAYA "Platform based Software Design Flow for Heterogeneous MPSoC", *ACM Journal: Transactions on Embedded Computing Systems (TECS), Special Issue on Rapid System Prototyping*, Accepted 17 January 2008, to appear

International Conferences, Symposiums, Workshops:

E. MORENO, **K. POPOVICI**, N. CALAZANS, A. JERRAYA "Integrating Abstract NoC Models within MPSoC Design", *Proceeding of RSP 2008,* 2-5 June 2008, Monterey, USA, to appear

K. POPOVICI, A. JERRAYA "Multilevel Communication Modeling for Multiprocessor System-on-Chip", *Proceeding of VLSI-DAT 2008*, 23-25 April 2008, Hsinchu, Taiwan, to appear

K. POPOVICI, A. JERRAYA "Simulink based Hardware-Software Codesign Flow for Heterogeneous MPSoC", *Proceeding of SCSC 2007*, 16-19 July 2007, San Diego, USA, Invited Paper

X. GUERIN, **K. POPOVICI**, W. YOUSSEF, F. ROUSSEAU, A. JERRAYA "Flexible Application Software Generation for Heterogeneous Multi-Processor System-on-Chip", *Proceeding of COMPSAC 2007*, 23-27 July 2007, Beijing, China

K. HUANG, S.I. HAN, **K. POPOVICI**, L. BRISOLARA, X. GUERIN, L. LI, X. YAN, S.I. CHAE, L. CARRO, A. JERRAYA "Simulink based MPSoC Design Flow: Case Study of Motion JPEG and H.264", *Proceeding of DAC 2007*, 4-8 June 2007, San Diego, USA**,** Best Paper Nominee

K. POPOVICI, X. GUERIN, F. ROUSSEAU, P.S. PAOLUCCI, A. JERRAYA "Efficient Software Development Platforms for Multimedia Applications at Different Abstraction Levels", *Proceeding of RSP 2007*, 28-30 May, 2007, Porto Alegre, Brazil, Best Rated Paper

K. POPOVICI, X. GUERIN, L. BRISOLARA, A. JERRAYA "Mixed Hardware-Software Multilevel Modeling and Simulation for Multithreaded Heterogeneous MPSoC", *Proceeding of VLSI-DAT 2007*, 25-27 April 2007, Hsinchu, Taiwan

N. E. ZERGAINOH, **K. POPOVICI**, A. JERRAYA, P. URARD "IP-Block based Design Environment for High Throughput VLSI Dedicated Digital Signal Processing Systems", *Proceeding of ASP-DAC 2005*, 18-21 January 2005, Shanghai, China

N. E. ZERGAINOH, **K. POPOVICI**, A. JERRAYA, P. URARD "Matlab based Environment for designing DSP Systems using IP blocks", *Proceeding of SASIMI 2004*, October 2004, Kanazawa, Japan

RESUME

La complexité et l'hétérogènité croissante des MPSoC sont accentuées par l'émergence de nouvelles applications télécoms et multimédia avec des contraintes fonctionnelles de plus en plus sévères. Pour ce genre d'architectures MPSoC hétérogènes, les environnements de programmation classiques ne sont pas adaptés pour les raisons suivantes: (i) la programmation de haut niveau ne gère pas efficacement les entrées/sorties (I/Os) et les systèmes de communication spécifiques, tandis que (ii) la programmation de bas niveau avec la gestion explicite des entrées-sorties et la communication spécifiques est très coûteuse en termes de temps et d'erreurs.

Cette thèse propose un flot de conception et validation du logiciel pour MPSoC. L'approche présentée commence par un modèle de haut niveau de l'application et de l'architecture en Simulink, permettant la simulation fonctionnelle rapide du modèle d'application. La génération et la validation du logiciel sont effectuées graduellement en partant de ce premier modèle, correspondant à différents niveaux d'abstraction. Des plateformes spécifiques de développement du logiciel (modèles abstraits de l'architecture) sont employées pour permettre le débogage des différents composants logiciels avec une interaction matériel/logiciel explicite.

Le flot proposé a été appliqué avec succès pour la génération et validation du logiciel pour plusieurs architectures MPSoC complexes qui exécutent des applications multimédia, comme l'encodeur vidéo H.264, le décodeur d'images Motion JPEG et le décodeur audio MP3. Les architectures MPSoC considérées contiennent plusieurs processeurs (DSP, RISC) interconnectés par un bus ou un réseau sur puce (NoC).

MOTS-CLES

MPSoC, logiciel embarqué, multimédia, plateforme de développement, niveau d'abstraction

TITLE

MULTILEVEL PROGRAMMING ENVIRONMENT FOR HETEROGENEOUS MPSOC ARCHITECTURES

ABSTRACT

Current multimedia applications demand complex heterogeneous multiprocessor system on chip (MPSoC) architectures with specific communication infrastructure in order to achieve the required performances. Programming these architectures usually results in writing separate low level code for the different processors (DSP, microcontroller), implying late global validation of the overall application with the hardware platform.

This thesis proposes a software design and validation flow able to efficiently use the resources of the architecture and allowing easy experimentation of several mappings of the application onto the platform resources. The thesis uses Simulink environment to capture both application and architecture initial representations. The software generation and validation is performed gradually from this initial model corresponding to different software abstraction levels. Specific software development platforms (abstract models of the architecture) are used to allow debugging of the different software components with explicit hardware-software interaction.

The proposed approach was applied on several multimedia platforms, involving high performance DSPs and RISC processors interconnected through buses or Network on Chip (NoC), to explore communication architecture and to produce an efficient executable code for several multimedia applications (H.264 encoder, Motion JPEG decoder and MP3 decoder).

Keywords

MPSoC, embedded software, multimedia, development platform, abstraction level

INTITULE ET ADRESSE DU LABORATOIRE

Laboratoire TIMA, 46 avenue Félix Viallet, 38031 Grenoble Cedex, France.

ISBN : 978-2-84813-114-6 **ISBNE : 978-2-84813-114-6**