Study and design of a manycore architecture with multithreaded processors for dynamic embedded applications

Charly Bechara

UNIVERSITE PARIS-SUD 11

ÉCOLE DOCTORALE : *Informatique Paris-Sud (EDIPS)*
Laboratoire : LCE, CEA LIST

***DISCIPLINE** Informatique*

THÈSE DE DOCTORAT
THÈSE DE DOCTORAT SUR TRAVAUX

soutenue le 08/12/2011

par

Charly BECHARA

Study and design of a manycore architecture with multithreaded processors for dynamic embedded applications

Directeur de thèse :	Daniel ETIEMBLE	Professeur - Université Paris-Sud, LRI
Co-directeur de thèse :	Nicolas VENTROUX	Ingénieur chercheur (PhD) - CEA LIST

Composition du jury :

Président du jury :	Alain MERIGOT	Professeur - Université Paris-Sud
Rapporteurs :	Frédéric PETROT	Professeur - INP, Grenoble
	Olivier SENTIEYS	Professeur - Université de Rennes (ENSSAT)
Examinateurs :	Agnès FRITSCH	Chef du laboratoire (PhD) – Thales Communications France

Abstract

Embedded systems are getting more complex and require more intensive processing capabilities. They must be able to adapt to the rapid evolution of the high-end embedded applications that are characterized by their high computation-intensive workloads (order of TOPS: Tera Operations Per Second), and their high level of parallelism. Moreover, since the dynamism of the applications is becoming more significant, powerful computing solutions should be designed accordingly. By exploiting efficiently the dynamism, the load will be balanced between the computing resources, which will improve greatly the overall performance.

To tackle the challenges of these future high-end massively-parallel dynamic embedded applications, we have designed the AHDAM architecture, which stands for "Asymmetric Homogeneous with Dynamic Allocator Manycore architecture". Its architecture permits to process applications with large data sets by efficiently hiding the processors' stall time using multithreaded processors. Besides, it exploits the parallelism of the applications at multiple levels so that they would be accelerated efficiently on dedicated resources, hence improving efficiently the overall performance. AHDAM architecture tackles the dynamism of these applications by dynamically balancing the load between its computing resources using a central controller to increase their utilization rate.

The AHDAM architecture has been evaluated using a relevant embedded application from the telecommunication domain called "spectrum radio-sensing". With 136 cores running at 500 MHz, AHDAM architecture reaches a peak performance of 196 GOPS and meets the computation requirements of the application.

Keywords: Multicore, MPSoC, manycore, asymmetric, multithreaded processors, embedded systems, dynamic applications, simulation

Résumé

Les systèmes embarqués sont omniprésents dans notre vie quotidienne. Un grand nombre de produits contiennent un ou plusieurs processeurs invisibles pour l'utilisateur dans un emballage sophistiqué. Ils effectuent des traitements complexes et communiquent avec l'environnement pour satisfaire les besoins des utilisateurs.

Les systèmes embarqués couvrent tous les aspects de la vie moderne et il y a de nombreux exemples de leur utilisation. Dans le domaine des télécommunications, il existe de nombreux systèmes embarqués, des commutateurs téléphoniques pour les réseaux jusqu'aux téléphones mobiles. Dans le domaine de l'électronique grand public, les systèmes embarqués sont utilisés dans les assistants numériques personnels (PDA), les lecteurs MP3, les consoles de jeux vidéo, les appareils photo numériques, les lecteurs de DVD, les GPS et les imprimantes. Cette liste est n'est pas exhaustive et il y a beaucoup d'autres exemples dans d'autres domaines comme les systèmes de transport, les équipements médicaux, les applications militaires, etc.

Les utilisateurs finaux ne veulent pas seulement davantage de fonctionnalités et de meilleures performances, mais ils sont aussi intéressés par des dispositifs ayant le même niveau de performance, mais moins chers. Ainsi, on observe des convergences dans les marchés de l'électronique grand public. Les montres numériques et les pagers ont évolué vers les assistants numériques personnels (PDA) et les téléphones intelligents (Smartphones). De même, les ordinateurs de bureau et les portables convergent vers les netbooks qui utilisent le processeur Atom d'Intel et les processeurs ARM. Ces dispositifs demandent de plus en plus des capacités calculatoires pour un budget énergétique faible avec des contraintes thermiques strictes.

Pendant près de 40 ans, les innovations technologiques sont succédées dans le but de réduire les temps d'exécution des programmes exécutés par le processeur. Une technique s'est appuyée sur la réduction des dimensions des transistors, qui a conduit à une augmentation du nombre de composants intégrables sur une puce, permettant d'implanter des architectures plus complexes et une augmentation de la fréquence d'horloge du processeur, d'où une exécution plus rapide des instructions. Cependant, ce gain de performance est aujourd'hui limité par les problèmes énergétiques et de dissipation thermique. En plus, les systèmes embarqués fonctionnent avec un budget de puissance limitée et donc d'augmenter la fréquence du processeur pour améliorer la performance n'est plus une solution pour les concepteurs de système.

Heureusement, de nombreuses applications embarquées sont naturellement parallèles. Les applications sont parallélisées au niveau des tâches pour atteindre des performances supérieures. Il existe deux solutions pour s'attaquer au parallélisme de tâches (TLP).

La solution la plus simple pour exécuter plusieurs tâches consiste à utiliser un processeur "monothread" avec toutes les techniques déployées pour accélérer le traitement d'un seul flux d'instructions. Le système d'exploitation organise et répartit les threads sur le processeur pour donner l'impression qu'ils sont exécutés en parallèle. Cela peut être considéré comme une virtualisation des ressources d'exécution. Pour cette virtualisation, deux techniques d'accélération pour les processeurs monothreads sont largement utilisées. La première, appelée le parallélisme temporel, consiste à réduire le temps d'exécution en divisant l'exécution des instructions en plusieurs étapes successives avec un chevauchement dans l'exécution de plusieurs instructions. C'est ce qu'on

appelle une exécution pipeline. La seconde, appelée le parallélisme spatial, repose sur la multiplication des ressources d'exécution. Dans ce type d'architectures, l'expression du parallélisme peut être explicite ou implicite. Le parallélisme est explicite lorsque le compilateur gère les dépendances de données et le flux de contrôle pour garantir l'exécution correcte du programme. Le contrôle est alors relativement simple et permet d'utiliser des fréquences d'horloge plus élevées ou de réduire les besoins énergétiques à performance donnée. Les architectures VLIW en sont l'exemple type. D'autre part, lorsque l'architecture traite dynamiquement tous les aléas d'exécution, le parallélisme est exploité de manière implicite. Ces architectures sont appelés superscalaires. Cette approche simplifie la tâche du compilateur au prix d'une plus grande complexité du matériel: les mécanismes de spéculation, l'exécution dans le désordre et les prédictions de branchement ont un grand impact sur l'efficacité énergétique et l'efficacité transistor de l'architecture du processeur.

La deuxième solution consiste à multiplier le nombre de cœurs pour exécuter les tâches en parallèle. L'avancement de la technologie des semi-conducteurs a permis aux fabricants de puces d'augmenter la puissance de calcul globale en intégrant des processeurs supplémentaires ou "cœurs" sur la même puce, ce qui est la version moderne des multiprocesseurs. Ainsi, cette solution exploite le parallélisme au niveau des threads (TLP), où plusieurs threads peuvent être exécutés en parallèle sur plusieurs cœurs. Dans le domaine des systèmes embarqués, ces architectures sont connues sous le terme MPSoC, qui signifie Multi-Processor System-On-Chip.

Dans ce contexte, c'est une architecture MPSoC pour les systèmes embarqués qui est étudiée et évaluée dans cette thèse.

Les systèmes embarqués sont de plus en plus complexes et requièrent des besoins en puissance de calcul toujours plus importants. Ils doivent être capables de s'adapter à l'évolution rapide des applications qui requièrent un haut niveau de performance (ordre du TOPS: Téra-opérations par seconde) et de parallélisme. Notamment, les applications haut de gamme ont beaucoup de parallélisme au niveau de tâches (TLP) et de parallélisme au niveau des boucles (LLP). Par conséquent, les architectures MPSoC doivent cibler l'ère "manycore" afin de répondre à ces besoins de calcul élevés. Les architectures multicœurs doivent être efficace au niveau transistor et énergie, puisque la taille de la puce et le budget énergétiques sont limitées dans les systèmes embarqués. Ainsi, ils doivent être conçus avec un bon équilibre entre le nombre de cœurs et la quantité de mémoire sur la puce. Les processeurs doivent être très efficaces en transistor et énergie. Il devrait n'y avoir aucune perte injustifiée de l'énergie dans les ressources d'exécution avec des techniques telles que la spéculation. Dans de telles architectures multicœurs complexes, il y a beaucoup de sources de latences qui provoquent des arrêts temporaires d'exécution des instructions, d'où une perte d'efficacité et d'énergie puisque les circuits continuent d'être alimentés pendant ces suspensions de fonctionnement. Dans ce contexte, les processeurs multithreads sont une solution intéressante à étudier.

Une caractéristique importante des applications embarquées de calculs intensifs est le dynamisme. Alors que certains algorithmes sont indépendants des données avec un flux de contrôle régulier, d'autres algorithmes sont très dépendants des données et leur temps d'exécution varie en fonction des données d'entrée, du contrôle de flux irrégulier, et de leur auto-adaptabilité aux environnements applicatifs. Par conséquent, l'architecture MPSoC devrait être très réactive par rapport aux besoins de calcul afin d'augmenter le taux d'occupation d'exécution des ressources. Ainsi, il

devrait permettre une répartition de charge globale et dynamique entre les ressources d'exécution.

Pour répondre aux besoins de ces applications de calcul intensif massivement parallèle et dynamique, nous proposons dans cette thèse l'architecture AHDAM qui signifie architecture homogène asymétrique avec allocation dynamique ou bien *Asymmetric Homogeneous with Dynamic Allocator Manycore architecture*. Cette architecture a été conçue afin de masquer efficacement la latence d'accès à la mémoire extérieure dont de nombreux accès sont nécessaires lors de la manipulation de grands volumes de données. Pour cela, des processeurs multitâches ont été utilisés. Par ailleurs, l'architecture AHDAM imbrique plusieurs niveaux de parallélisme afin de tirer partie efficacement des différentes formes de parallélisme des applications, et ainsi atteindre un haut niveau de performance. Enfin, cette architecture utilise un contrôleur centralisé pour équilibrer la charge de calcul entre ses ressources de calcul afin d'augmenter leur taux d'utilisation et d'exécuter efficacement les applications fortement dynamiques.

Le chapitre 1 présente le contexte de notre travail en se concentrant principalement sur les exigences des applications et des solutions architecturales existantes. Le cadre de notre étude sont les applications massivement parallèles et dynamiques pour l'embarqué. Ces applications sont très parallèles. Le parallélisme peut être extrait au niveau thread (TLP) et au niveau des boucles (LLP). Ainsi, une application peut avoir de nombreux threads qui peuvent être traitées en parallèle. Par conséquent, les architectures MPSoCs de type "manycore" sont des solutions naturelles pour ces applications. En outre, le dynamisme de ces applications nécessite une solution efficace pour gérer l'utilisation des ressources et équilibrer les charges dans le MPSoC afin de maximiser la performance globale.

On identifie trois grandes familles de MPSoCs pour les systèmes embarqués dans l'état de l'art: les MPSoC symétriques, les MPSoC asymétriques homogènes et les MPSoC asymétriques hétérogènes.

Les MPSoCs symétriques sont constitués de plusieurs processeurs homogènes qui exécutent à la fois la tâche de contrôle et les tâches de calculs. Les MPSoCs asymétriques sont constitués d'un (parfois plusieurs) processeur de contrôle centralisé ou hiérarchisé, et plusieurs processeurs homogènes ou hétérogènes pour les tâches de calcul. Dans notre contexte d'étude, les architectures MPSoC asymétriques homogènes sont la meilleure solution pour les applications dynamiques, puisqu'elle permet l'équilibrage de charge rapide et réactive entre les processeurs homogènes. Ces architectures ont une grande efficacité transistor et énergétique en raison de la séparation entre les processeurs de contrôle et de calcul.

En particulier, une architecture MPSoC asymétriques homogène, appelé SCMP, qui est la propriété du laboratoire du CEA LIST, sera utilisée dans le reste de cette thèse comme l'architecture de référence pour les expérimentations. SCMP est conçue pour traiter les applications embarquées avec un comportement dynamique en faisant migrer les threads entre les cœurs de calcul en utilisant un contrôleur central.

Par ailleurs, et selon nos observations, les architectures MPSoC asymétriques homogènes ne répondent pas aux exigences des applications embarquées haut de gamme qui sont massivement parallèles. Tout d'abord, elles ne sont pas extensibles au niveau manycore parce que le contrôleur central est une source de contentions. Par exemple, SCMP peut supporter jusqu'à 32 cœurs de calcul avant de connaître une dégradation des performances. Ensuite, les puces manycore ont un

nombre limité de pattes d'E/S, d'où une bande passante limitée. Cela implique que plus le trafic hors-puce augmentera, plus les processeurs subiront de suspensions dans la puce.

Il existe deux voies possibles pour améliorer les architectures MPSoCs asymétriques homogènes: la scalabilité et les processeurs multitâches matériels. Dans cette thèse, nous allons d'abord étudier les avantages/inconvénients du multithreading matériel dans l'architecture SCMP, puis nous allons proposer une nouvelle solution qui va cibler l'ère manycore. Cette solution doit relever les défis des applications embarquées.

Le chapitre 2 explore et analyse les performances et l'efficacité des processeurs matériels multithreads dans les systèmes embarqués. Les processeurs embarqués doivent avoir une taille dans l'ordre du mm^2 et consommer de l'ordre de quelques mW. Ainsi, ils doivent utiliser une technologie simple pour exploiter l'ILP, telles que le pipeline ou l'approche VLIW. Un processeur matériel multithread fournit les ressources matérielles et des mécanismes pour exécuter plusieurs threads matériels sur un cœur de processeur, afin d'accroître son utilisation du pipeline, et donc le débit d'exécution des applications. Au sein d'un processeur multithread, les slots d'instruction inutilisés sont remplis par des instructions d'autres threads. Les threads matériels sont en compétition pour accéder aux ressources partagées et pour tolérer les aléas de pipeline avec une longue période de latence, comme c'est le cas lors d'un défaut de cache. Ces événements peuvent bloquer le pipeline jusqu'à 75% de son temps d'exécution. Ainsi, le principal avantage des processeurs multithreads sur les autres types de processeurs est leur capacité à cacher la latence d'un thread. Les futures architectures manycores ont tendance à utiliser de petits cœurs RISC comme éléments de base des traitements. Dans ce cas, plusieurs processeurs peuvent être intégrés sur une seule puce tout en gardant la consommation d'énergie globale sous un seuil tolérable. Par conséquent, nous considérons un processeur RISC (AntX) avec un pipeline à 5 étages, démarrant dans l'ordre une seule instruction par cycle. Ensuite, nous étudierons ce processeur, au niveau RTL (VHDL), avec deux techniques de multithreading: le multithreading entrelacé (IMT) et le multithreading par bloc (BMT). Nous avons synthétisé les 3 processeurs en technologie TSMC 40 nm. Les résultats de synthèse montre que le banc de registres occupe plus de 38% de la surface de base globale. Donc, il n'est pas efficace d'intégrer plus de deux threads (TC) par processeur multithread et c'est pourquoi nous nous sommes limités à deux threads par processeur. Les résultats de synthèse montrent également que les versions IMT et BMT ont 73.4% et 61.3% de surface en plus par rapport au processeur monothread. C'est le BMT qui a la plus petite surface.

Enfin, nous avons comparé les performances et l'efficacité transistor des processeurs MT en utilisant comme application le tri-bulle, tout en variant la taille du cache L1 de données et la latence de la mémoire de données. Les résultats montrent qu'il n'y a pas de conclusion définitive sur le meilleur type de processeur multithread. En fait, il y a un compromis entre la taille des données de la mémoire cache, la latence mémoire de données, et le surcoût en surface du cœur. Choisir le meilleur processeur multitâche dépend fortement du cahier des charges du concepteur du système et des exigences d'application.

D'après cette conclusion, nous allons explorer dans le prochain chapitre l'impact du processeur multithread sur les performances d'une architecture MPSoC asymétrique: l'architecture SCMP.

Le chapitre 3 étudie les avantages/inconvénients du multithreading matériel dans un contexte MPSoC asymétrique homogène (architecture SCMP). Pour cette exploration, nous présentons le simulateur SESAM, dans lequel l'architecture SCMP est modélisée. Ensuite, nous étendons SESAM pour supporter les processeurs multithreads. En particulier, nous avons développé un nouveau simulateur multithread au cycle près (ISS) en SystemC pour modéliser le processeur IMT/BMT avec deux threads (TC). Nous utilisons plusieurs benchmarks basés sur un contrôle des flux et des applications de streaming afin de choisir ce qui convient le mieux pour un processeur multithread entre les approches IMT et BMT, quel ordonnanceur de threads global pour plusieurs processeurs multithread donne la meilleure performance (VSMP ou SMTC), et quelle architecture MPSoC asymétrique est la plus performante et transistor efficace (SCMP ou MT_SCMP).

Deux applications du domaine de l'embarqué sont utilisées: l'étiquetage de composants connexes (de type contrôle de flux) et WCDMA (de type dataflow/streaming). Dans le modèle d'exécution de flux de contrôle, les tâches sont traitées jusqu'à la fin, alors que dans le modèle d'exécution de flux de données, les tâches se synchronisent entre elles sur les données, créant ainsi des aléas de pipeline en plus. Les deux modèles d'exécution couvrent un large ensemble de comportements d'applications.

Les résultats montrent que le processeur multithread bloqué (BMT) et l'ordonnanceur SMTC convient le mieux pour MT_SCMP.

Afin d'estimer la surface des deux systèmes, nous utilisons les résultats de synthèse en technologie TSMC 40 nm pour les processeurs et les réseaux d'interconnexion, et nous estimons la taille des caches en utilisant l'outil CACTI 6.5. Pour résumer les résultats, le MT_SCMP donne une meilleure performance de pointe, mais inférieure en efficacité transistor que SCMP. En fait, les performances de MT_SCMP dépendent fortement de cinq paramètres principaux: le TLP de l'application, le taux de défauts de caches, la latence d'un défaut de caches, la hiérarchie mémoire, et l'ordonnancement global des threads. Ce dernier paramètre implique que pour des applications dynamiques, un équilibrage de charge dynamique et l'ordonnancement donnent une performance optimale. C'est pourquoi SCMP est une architecture hautement efficace.

Choisir des processeurs multithreads ou non pour SCMP est la responsabilité du concepteur du système. Si les performances de pointe sont un paramètre clé, les processeurs multithreads sont une solution intéressante. Cependant, pour l'efficacité transistor, les processeurs monothreads restent une solution plus efficace.

SCMP a quelques limitations pour s'attaquer aux exigences des applications dynamiques et massivement parallèles. Il y a des limites à la scalabilité au niveau manycore et des limitations pour traiter des quantités de données qui ne rentrent pas dans la mémoire sur la puce.

Pour surmonter ces limitations, nous allons concevoir dans le chapitre suivant une nouvelle architecture manycore qui s'attaque aux défis des applications dynamiques haut de gamme de l'avenir appelée architecture AHDAM.

Le chapitre 4 présente une nouvelle architecture manycore appelée AHDAM, qui signifie *Asymmetric Homogeneous with Dynamic Allocator Manycore* architecture. Il est utilisé comme un accélérateur pour les applications massivement parallèles et dynamiques en permettant la migration des threads entre les unités d'exécution via un contrôleur central. En outre, elle est conçue pour accélérer l'exécution des codes de boucle, ce qui constitue souvent une grande partie

du temps d'exécution de l'application. L'architecture AHDAM met en œuvre des processeurs monothread et multithread pour accroître l'utilisation des processeurs si nécessaire. L'architecture AHDAM est présentée en détail dans ce chapitre. En particulier, son environnement de système, son modèle de programmation, sa description architecturale et les fonctionnalités de chaque composant matériel, son modèle d'exécution, et sa scalabilité maximale, sont expliquées. L'étude montre que la scalabilité d'AHDAM peut atteindre jusqu'à 136 processeurs (8 Tuiles x 16 LPEs + 8 MPEs) en fonction de l'exigence d'application.

Enfin, le chapitre 5 évalue les performances et l'efficacité transistor de l'architecture AHDAM en utilisant une application pertinente de l'embarquée dans le domaine des télécommunications appelé *radio-sensing*. Cette application a été conçue par Thales Communications France. La configuration "high-sensitivity" de cette application a beaucoup d'exigences de calcul (75.8 GOPS), beaucoup de parallélisme au niveau des threads et des boucles (99.8%), une grande quantité de données (432 Mo), et elle est dynamique. L'application de "radio-sensing" est parallélisée et portée en utilisant le modèle de programmation d'AHDAM. L'architecture AHDAM est simulée en utilisant une combinaison d'outils de simulation tels que SESAM et Trimaran, et en utilisant le modèle analytique du processeur BMT que nous avons développé. Avec 136 cœurs cadencés à 500 MHz, l'architecture AHDAM atteint une performance crête de 196 GOPS et répond aux exigences de l'application.

Nous évaluons l'efficacité transistor de l'architecture. Après avoir mené plusieurs expérimentations, nous concluons que la propriété asymétrique de l'architecture AHDAM est essentielle pour les applications dynamiques pour augmenter leurs performances. L'ordonnanceur dynamique donne une accélération de 2.4 par rapport à l'ordonnanceur statique de l'application de "radio-sensing". En outre, le multithreading booste les performances de l'architecture AHDAM avec un gain de 39% par rapport au monothread pour seulement 7% d'augmentation de surface totale. Donc, le multithreading augmente l'efficacité transistor de l'architecture. Enfin, l'architecture AHDAM a une accélération de 574 par rapport a 1 PE, tandis que SCMP a une accélération de 7.2. Ceci dit, l'architecture AHDAM est un progrès important par rapport à SCMP et peut répondre aux exigences des futures applications haut de gamme, massivement parallèles, et dynamiques. La surface d'AHDAM sans les interconnections est estimée d'être $53\ mm^2$ en technologie 40 nm.

Il reste beaucoup de concepts proposés dans l'architecture AHDAM qui ont encore besoin d'études, de développements et d'améliorations.

Les perspectives à court terme peuvent être divisées en trois étapes principales: le développement d'un simulateur, la construction d'un prototype, et la comparaison avec les autres architectures multicœurs.

Dans la première étape, nous avons besoin de développer un simulateur pour AHDAM, principalement une extension de l'environnement de simulation SESAM. De nouveaux composants doivent être élaborés en SystemC qui n'existaient pas auparavant pour l'architecture SCMP, telles que la mémoire cache L2 et ses protocoles, le *Thread Context Pool* et la mémoire scratchpad TCP state. En outre, un simulateur multithread VLIW doit être développé. Ensuite, nous avons besoin d'encapsuler chaque ensemble de processeurs (tuile) dans un module afin que le contrôleur

matériel la voie comme un PE, et de valider toutes les fonctionnalités d'une tuile. En particulier, l'architecture du réseau sur puce doit être étudiée. Enfin, la mémoire L2 d'instructions et de données devrait être divisée en deux mémoires séparées.

Après avoir construit l'environnement de simulation AHDAM, l'environnement d'exécution proposé pour "Fork-Join" devrait être développé. Ce "runtime" est un élément essentiel de la fonctionnalité de l'architecture AHDAM et la gestion des transferts internes à une tuile ou entre tuiles. L'objectif est de trouver le nombre optimal de threads à lancer (fork) car c'est un paramètre important pour l'accélération de l'ensemble des régions de boucles. En outre, le modèle d'exécution "farming" doit être validé.

Avec un simulateur AHDAM et l'environnement d'exécution utilisable, il serait intéressant de poursuivre le développement de la chaîne de programmation automatique que nous avons commencé dans le chapitre 4. Il pourrait être basé sur l'outil PAR4ALL. Cela nous permettrait de porter facilement toutes les applications 'legacy' sur l'architecture AHDAM.

La deuxième étape principale consiste à construire un prototype de l'architecture AHDAM sur une carte d'émulation matérielle. Ce prototype sera la preuve de concept de l'architecture. Ayant un tel prototype, on peut estimer précisément l'efficacité transistor et énergétique de l'architecture AHDAM ainsi que des processeurs multithreads. En particulier, nous pouvons rendre la puce AHDAM plus économe en énergie, en explorant de nouvelles stratégies d'équilibrage de charges à l'intérieur de chaque tuile et entre les tuiles, et d'intégrer les stratégies dans l'environnement d'exécution. D'autres techniques énergétiques efficaces comme DVFS peuvent être implémentés sur FPGA. Nous pouvons imaginer que chaque tuile fonctionne avec une fréquence différente contrôlée par le contrôleur matériel selon les besoins de l'application.

Enfin, la troisième étape principale consiste à comparer l'architecture AHDAM avec d'autres solutions manycore pertinentes tels que TILE64 de Tilera, P2012, de ST Microelectronics et MPPA de Kalray. Pour cette raison, nous avons besoin de porter plusieurs applications de l'embarqué qui sont dynamiques et pertinentes et qui ont beaucoup de parallélisme et des exigences de calcul. Ces applications doivent fonctionner sur toutes ces architectures afin qu'on puisse mener une comparaison appropriée.

A ce stade, nous sommes prêts à mener un transfert technologique de la puce d'AHDAM pour des projets nationaux/européens. En particulier, nous pouvons développer deux versions d'AHDAM: bas de gamme et haut de gamme. La première version vise le marché de l'embarqué, tandis que la dernière cible le marché des serveurs, et le calcul dans le nuage en particulier. Ce qui différencie les deux puces est le nombre de tuiles, le nombre de processeurs par tuile, et les stratégies d'équilibrage de charge utilisée dans la puce qui dépendent de la performance et de l'efficacité énergétique.

Sur le long terme, il y a plusieurs améliorations architecturales imaginables pour l'architecture AHDAM.

Comme la technologie du procédé s'améliore, il y a plus de préoccupations au sujet de la fiabilité de l'architecture AHDAM. AHDAM pourrait être utilisé dans des domaines critiques tels que les applications militaires, nucléaires et spatiales, où la tolérance de panne est une décision architecturale non négligeable. Nous pouvons imaginer que la puce AHDAM appliquera la tolérance de pannes au niveau Tuile, MPE et LPE par l'intégration de composants de rechange.

En outre, il y a un écart énorme entre la vitesse du processeur et de la mémoire. Cela ne semble pas devoir changer à l'avenir sauf si une nouvelle percée technologique est trouvée pour les technologies de mémoire. En supposant que ce n'est pas le cas, il devrait y avoir une solution architecturale pour éviter de bloquer les processeurs multithread LPE. Une solution serait d'augmenter le nombre de threads matériels par LPE. Mais comme nous l'avons vu précédemment dans le chapitre 2, ce n'est pas une solution efficace en transistor pour les processeurs faible coût. Une nouvelle technique serait d'utiliser une architecture de multithreading avec entrelacement statique N sur M. Cette technique implique qu'un processeur multithread a N threads de premier plan (thread matériel) et M threads virtuels stockés dans une spéciale mémoire scratchpad à proximité du processeur multithread. De cette façon, nous augmentons le nombre de threads pris en charge par LPE.

La puce AHDAM est une architecture manycore. Mais comme nous l'avons vu dans le chapitre 4, il y a aussi des limites à l'extensibilité de l'architecture. Une solution serait d'intégrer plusieurs contrôleurs DDR3 sur puce, ce qui signifie qu'on puisse augmenter ainsi le nombre de tuiles. Une autre solution au problème d'extensibilité est de considérer l'architecture AHDAM comme un cluster optimisé dans un environnement de multi-clusters. Puis, en utilisant une solution hiérarchique, nous pouvons augmenter le nombre de cœurs (plus de 1000 cœurs). A ce stade, on pourrait imaginer que le modèle de programmation AHDAM est étendu pour supporter la communication MPI entre les différents clusters AHDAM. Ainsi, AHDAM supporterait le modèle OpenMP + MPI.

Enfin, la mémoire SRAM et les mémoires caches peuvent être empilés sur les processeurs en utilisant une technologie d'empilage 3D. Ce serait une amélioration spectaculaire de la taille de la puce, puisque 73% de la surface estimé de la puce est occupée par les caches et la mémoire SRAM. Ainsi, plus de cœurs pourraient être intégrées et les temps d'accès mémoire seraient plus rapides. Cela permettrait d'améliorer les performances de la puce AHDAM et peut-être de nouvelles améliorations architecturales pourraient être proposées avec une technologie d'empilage 3D.

Mots clés: Multicœur, MPSoC, manycore, asymétrique, processeur multitâche, systèmes embarqués, applications dynamiques, simulation

Acknowledgments

You can dream, create, design and build the most wonderful place in the world, but it requires people to make the dream a reality – Walt Disney, film producer

I am greatly indebted to many individuals for their advice and assistant in this PhD thesis journey.

I would like to thank Thierry COLLETTE (head of CEA LIST/DACLE department), Cécile MORILLON (assistant head of CEA LIST/DACLE department), Laurent LETELLIER (ex-head of LCE laboratory), and Raphaël DAVID (head of LCE laboratory), for warmly hosting me in the DACLE department and the LCE laboratory, in addition for providing me all the resources for the success of this work.

A special warm thank for Nicolas VENTROUX (PhD thesis supervisor, R&D engineer at CEA LIST), for trusting my capabilities and giving me this opportunity to work on this PhD topic. His motivation and dynamism for this work has infected me all along this successful journey. I thank you for your daily professional and personal advices. I really enjoyed working with you.

I am also grateful to Daniel ETIEMBLE (PhD thesis director, professor at Université Paris-Sud), for his great supervision and scientific guidance. I thank you for your kind advices, and especially for your fast responsiveness that unblocked me in several issues. I was honored to be supervised by a professor from your quality and I have learned a lot from your experience.

I would like also to express my gratitude to all my PhD defense jury members. In particular, I thank Olivier SENTIEYS (reviewer, professor at Université de Rennes) and Frédéric PETROT (reviewer, professor at Grenoble-INP), for taking the time to read my thesis report and for your interesting comments about my work. In addition, I thank Alain MERIGOT (president of the jury, professor at Université Paris-Sud) and Agnès FRITSCH (examiner, head of laboratory at Thales Communications France) for accepting to be in my PhD defense jury.

These 3 years would never pass so fast and so smooth without the great support of my laboratory colleagues. I thank you all, one by one, for the great time we shared during and after the work. In particular, I would like to thank all the people who helped me to overcome all the technical problems related to my work, and also for our great discussions about my work. It is of great pleasure to continue this journey with you after the PhD.

A special thank to the department secretaries, in particular Annie, Sabrina, and Frédérique, for taking care with an ultimate speed and lovely smile all my administrative issues.

In these 3 years, I have met lot of friends at CEA from outside my laboratory to whom I am grateful. In particular, I am thinking of all the Lebanese colleagues from all the CEA departments. I thank you all for the beautiful moments we shared together. You were real quality friends.

Last but not least, a very special acknowledgment of love and gratitude to my family, the RIM's. Thank you my parents Elie and Denise, my brother Anthony and my sister Melissa, for your continuous love and support. It was the key for my success throughout the challenging years of my education. I also thank my wife's family, the KHATTAR's, for showing their ultimate care and sharing this joy with me.

Finally, there are no words that can express my feelings of gratitude to my wife Manar. Your big

support, your great tolerance, your daily encouragements, and your shining love made this tough journey very special. I love you.

Contents

List of Figures

List of Tables

Introduction

One thing is sure. We have to do something. We have to do the best we know how at the moment...If it doesn't turn out right, we can modify it as we go along. – Franklin D.Roosevelt, president

Contents

Context of study

Embedded systems exist everywhere in our quotidian life. Almost every single product contains one or multiple processors hidden from the end user in a fascinating package. They are performing the computation and communication with the environment to bring the intelligence and satisfy the end user needs.

Embedded systems span all aspects of modern life and there are many examples of their use. In the telecommunication domain, there are numerous embedded systems from telephone switches for the network to mobile phones at the end-user. Computer networking uses dedicated routers and network bridges to route data. In the consumer electronics domain, embedded systems are employed in personal digital assistants (PDAs), MP3 players, videogame consoles, digital cameras, DVD players, GPS receivers, and printers. This list is exhaustive and there are much more examples that exist in other domains such as the transportation systems, medical equipments, military applications, etc...

The end users no longer only wants more features and better performance, but are increasingly interested in devices with the same performance level at a lower price. Thus, consumer electronic markets, and therefore industries, started to converge. Digital watches and pagers evolved into powerful personal digital assistants (PDA) and smartphones. Similarly, desktop and laptop computers were recently reduced to netbooks that use Intel Atom and ARM processors. The resulting devices demand ever more computational capabilities at decreasing power budgets and within stricter thermal constraints [42].

During nearly 40 years, the technological innovations followed one another with the goal of reducing the processor execution times. One technique relied on shrinking the physical transistor integration, which led to an increase in the processor clock frequency, hence a faster instruction execution. However, this performance gain is limited today by the physical integration barriers. In addition, embedded systems function on a limited power budget and thus increasing the processor frequency to improve the performance is no longer a solution for system designers.

Fortunately, many embedded applications are parallel by nature. Applications are parallelized on the task level to reach higher performances. There exist 2 solutions to tackle to the task level parallelism (TLP).

The simplest solution to execute multiple tasks consists in using a monothreaded processor with all the techniques deployed to accelerate the processing of a single instruction flow. The operating system schedule and allocate the threads concurrently on the processor giving the impression that they are running in parallel. This can be thought of as virtualization of the execution resources. For this virtualization, two acceleration techniques for the monothreaded processors are largely used. The first one, called *temporal parallelism*, consists in reducing the execution time by dividing the instruction execution into several successive stages. This is referred to a *pipeline* execution. The second one, called *space parallelism*, relies on the multiplication of the execution resources. Expressing the parallelism in this type of architecture can be *explicit* or *implicit*. The parallelism is *explicit* when the compiler manages the data dependencies and the control flow in order to guarantee the availability of the resources. The control is then relatively simple and makes it possible to use higher clock frequencies. For instance, this is the case for the VLIW [48, 115] architectures. On the other hand, when the architecture deals dynamically with all these execution hazards, the parallelism is exploited in an *implicit* way. These architectures are called superscalar. The advantages of this approach are the simplicity of the parallelism description and its dynamic management during the execution. Nevertheless, the complexity of the speculation mechanisms, out-of-order executions and branch predictions has great impact on the energy efficiency and transistor efficiency of the processor architecture.

The second solution consists of multiplying the number of cores and executing the tasks in parallel. The advancement in semiconductor processing technology allowed chip manufacturers to increase the overall processing power by adding additional CPUs or "cores" to the microprocessor chip. Hence, this solution exploits the parallelism at the thread level (TLP), where multiple threads can be executed in parallel on multiple cores. These architectures are known as *MPSoC*, which stands for *Multi-Processor System-On-Chip*.

The context of study of this thesis will be the design of MPSoC architectures for the embedded systems.

Problematic

Embedded systems require more intensive processing capabilities and must be able to adapt to the rapid evolution of the high-end embedded applications. These embedded applications are getting more and more complex. Their computation requirements have reached the order of TOPS (Tera Operations Per Second) and they have a large data set. These applications have lot of thread level parallelisms and loop level parallelisms. Therefore, MPSoC architectures must target the manycore era in order to meet this high computation demands. The manycore architecture must be transistor and energy efficient, since the chip size and the power budget are limited in the embedded systems. Thus, they must be designed with a good balance between the number of cores and the on-chip memory. The cores should be highly transistor and energy efficient, where specialization is a key element. There should be no unjustified waste of energy in execution resources with techniques such as speculation. In such complex manycore architectures, there are lots of sources of latencies

that cause the cores to be stalled, thus wasting energy. In this context, multithreaded processors are an interesting solution to investigate.

An important feature of these embedded computation-intensive applications is the dynamism. While some algorithms are data independent with a regular control flow, other algorithms are highly data-dependent and their execution time vary with respect to their input data, their irregular control flow, and their auto-adaptability to the application environments. Therefore, the MPSoC architecture should be highly reactive with respect to the computation needs in order to increase the execution resources occupation rate. Thus, it should support global and dynamic load-balancing of threads between the execution resources.

Based on these observations, we will design a new manycore architecture that tackles the challenges of future high-end massively parallel dynamic applications. The manycore architecture is called AHDAM, which stands for *Asymmetric Homogeneous with Dynamic Allocator Manycore architecture.*

Outline of this report

Chapter 1 presents the context of our work by focusing mainly on the applications requirements and the existing architectural solutions. First, it highlights the performance requirements of future high-end massively-parallel dynamic embedded applications. Then, it presents a state of the art of the *MPSoCs* for embedded systems by providing a classification of the overall architectures' space that currently exist in the literature. Three big families are identified: *Symmetric MPSoCs*, *Asymmetric Homogeneous MPSoCs*, and *Asymmetric Heterogeneous MPSoCs*. The *Asymmetric Homogeneous MPSoCs* will be exploited since its characteristics can meet the future embedded applications constraints. An *asymmetric homogeneous architecture* consists of one (sometimes several) centralized or hierarchized control core, and several homogeneous cores for computing tasks. In particular, an asymmetric homogeneous MPSoC, called SCMP [151], which is proprietary to CEA LIST laboratory, will be retained for the rest of this thesis as the architecture of reference for experimentations. SCMP is designed to process embedded applications with a dynamic behavior by migrating the threads between the cores using the central controller. Finally, and based on our observations, we will analyze why the currently existing *asymmetric homogeneous MPSoC architectures* do not meet the requirements of future high-end massively-parallel dynamic embedded applications, and what are the possible solutions. In particular, we will be interested in hardware multithreading as an efficient solution to increase the performance of the *asymmetric homogeneous MPSoC* architectures.

At the beginning, chapter 2 explores and analyzes the performance and efficiency of hardware multithreaded processors in embedded systems. First of all, it provides a classification of the different types of multithreaded processors that exist in the literature. In particular, two multithreading techniques for single-issue cores will be retained: *Interleaved multithreading* (IMT) and *Blocked multithreading* (BMT). These multithreaded architectures should meet the embedded systems requirements and are suitable for manycore architectures. Then, we apply the two multithreading techniques on a small footprint monothreaded core at the RTL level (VHDL), and synthesize the 3 cores in 40 nm TSMC technology. In this way, we compare the area overhead of each multithreaded processor type (IMT and BMT) with respect to the monothreaded core. Finally, we compare the

performance of the monothreaded, IMT, and BMT cores in a typical processor system configuration, and we show the characteristics of each processor type and under which conditions it should be used.

Then, chapter 3 explores the advantages/disadvantages of hardware multithreading in an asymmetric homogeneous MPSoC context (SCMP architecture). In order to conduct this exploration, we present the SESAM simulation framework, where the SCMP architecture is modeled. Then, we extend SESAM to support multithreaded processors. In particular, we have developed a new cycle-accurate multithreaded Instruction Set Simulator (ISS) in SystemC to model the IMT processor with 2 thread contexts (TC). After replacing the monothreaded processor by an IMT/BMT processor with 2 TCs, we used several benchmarks in order to know which multithreaded processor type suits best the SCMP architecture and to measure the transistor efficiency of the new SCMP architecture with multithreaded processors. For this reason, two types of applications are used from the embedded domain: connected component labeling (control-flow) and WCDMA (dataflow/streaming). In the control-flow execution model, the tasks are processed until completion, while in the dataflow execution model, the tasks synchronize between each other on data, thus creating more processor stalls. Both execution models cover a large set of applications behavior. The benchmarking results show that multithreading boosts the performance of SCMP, however it does not reach the desired level to make it a transistor efficient solution. Chapter 3 concludes that SCMP has some limitations for tackling the requirements of the future massively-parallel dynamic applications. In particular, it was due to the scalability limitations to the manycore level and the lack of support for large data set sizes of applications that does not fit in the on-chip memory.

To overcome these limitations, chapter 4 presents a new manycore architecture called AHDAM [17]. AHDAM stands for *Asymmetric Homogeneous with Dynamic Allocator Manycore architecture*. It is used as an accelerator for massively parallel dynamic applications by migrating the threads between the execution units using a central controller. In addition, it is designed to accelerate the execution of the loop codes, which often constitutes a large part of the overall application execution time. AHDAM architecture implements monothreaded and multithreaded cores to increase the cores utilization when necessary. AHDAM architecture is presented in details in this chapter. In particular, its application system environment, its programming model, its architectural description and the functionalities of each hardware component, its execution model, and its maximum scalability, are presented in this chapter.

Finally, chapter 5 evaluates the performance and transistor efficiency of AHDAM architecture by using a relevant embedded application from Thales Communications France in the telecommunication domain called *radio-sensing*. This application has lots of computation requirements, lots of parallelism at the thread and loop levels, a large data set, and is dynamic. The radio-sensing application is parallelized and ported using the AHDAM programming model flow. We evaluate the transistor efficiency of the architecture. In particular, we estimate the overall chip area in 40 nm technology for multiple chip configurations and we evaluate its performance by running the radio-sensing application on different chip configurations. The AHDAM architecture shows excellent results compared to the SCMP architecture.

Chapter 1

MPSoC architectures for dynamic applications in embedded systems

A problem well stated is a problem half solved. – Charles F. Kettering, inventor

Contents

During the last decades, computing systems were designed according to the CMOS technology push resulting from Moore's Law, as well as the application pull from ever more demanding applications [42]. The emergence of new embedded applications for mobile, telecom, automotive, digital television, mobile communication, medical and multimedia domains, has fueled the demand for architectures with higher performances, more chip area and power efficiency. These complex applications are usually characterized by their computation-intensive workloads, their high-level of parallelism, and their dynamism. The latter implies that the total application execution time can highly vary with respect to the input data, irregular control flow, and auto-adaptive applications.

Traditional high-performance superscalar general-purpose processors implement several architectural enhancement techniques such as out-of-order execution, branch prediction, and speculation, in order to exploit the instruction-level parallelism (ILP) of a sequential program. However, the ILP has reached its limits [156] and cannot be more exploited. To compensate the ILP limitation, chip manufacturers relied on increasing the clock frequency to provide free performance gain. Yet, a higher clock frequency implies more power dissipation. Therefore, superscalar processors have low transistor/energy efficiency that render them not suitable for embedded systems applications. On the other hand, the advancement in semiconductor processing technology allowed chip manufacturers to increase the overall processing power by adding additional CPUs or "cores" to the microprocessor package. Hence, this coarse-grained solution consists of exploiting the parallelism

at the thread level (TLP), where multiple threads can be executed in parallel on multiple cores or concurrently on hardware multithreaded cores. According to the authors in [158], all the microprocessor chip architectures for the embedded systems world are called *MPSoC*, which stands for *Multi-Processor System-On-Chip*.

MPSoCs consist of any number of processing cores greater than 1 connected to any number of IPs (Intellectual Property) through an interconnection network, all integrated in one microprocessor package. The interconnection network can be a simple bus or a complex Network-On-Chip (NoC). MPSoCs provide the necessary execution resources to exploit the Thread-Level Parallelism (TLP) of an application.

The embedded world has been the pioneer to realize the need of novel SoC architectures in order to meet the embedded systems applications requirements: high performance, low-power, small die area, real-time and multithreading execution. In the year 2000, the first MPSoC was released for wireless base stations applications. It is the Lucent Daytona [3] with a chip size of 200 mm^2 in a 0.25 μm CMOS process. After that date, lots of MPSoC architectures have been designed for the embedded systems domain. This happened thanks to Moore's Law advances and the continuous demand for more performance and transistor/energy efficient solutions for embedded applications such as multimedia and graphics, telecommunication, embedded computer vision, networking, and military applications. On the other hand, general-purpose processors have followed the pace 1 year later with IBM POWER4 processor [140], the first commercial 2 cores chip launched in 2001.

In this chapter, we will first highlight the performance requirements of future dynamic embedded applications. Then, we will present a state of the art of the *MPSoCs* for embedded systems. At the beginning, we will explain the different MPSoC characteristics that identify each architecture. After, and based on these characteristics, we will provide a classification of the overall architectures' space that currently exist in the literature. We identify 3 big families: *Symmetric MPSoCs*, *Asymmetric Homogeneous MPSoCs*, and *Asymmetric Heterogeneous MPSoCs*. We will analyze and compare different architectural solutions in order to identify which characteristics can meet the future high-end massively-parallel embedded applications constraints. In particular, an asymmetric homogeneous MPSoC, called SCMP [151], will be retained for the rest of this thesis as the architecture of reference for experimentations. SCMP is designed to process embedded applications with a dynamic behavior by migrating the threads between the cores using the central controller. Finally, and based on our observations, we will analyze why the currently existing *asymmetric homogeneous MPSoC architectures* do not meet the requirements of future embedded applications, and what are the possible solutions.

1.1 Dynamic applications in embedded systems

Throughout the history of computing systems, applications have been developed that demanded ever more performance, and this will not change in the foreseeable future. Recent innovative embedded systems applications such as domestic robots, future cars, telepresence, Human++ [112], smart camera networks, realistic games, virtual reality, and cognitive radio systems, require extremely large amounts of processing power in the order of TOPS (Tera Operations Per Second). Figure 1.1 shows the computation requirements for several embedded application domains.

Most of these embedded applications are parallel by nature. Therefore, an application can be

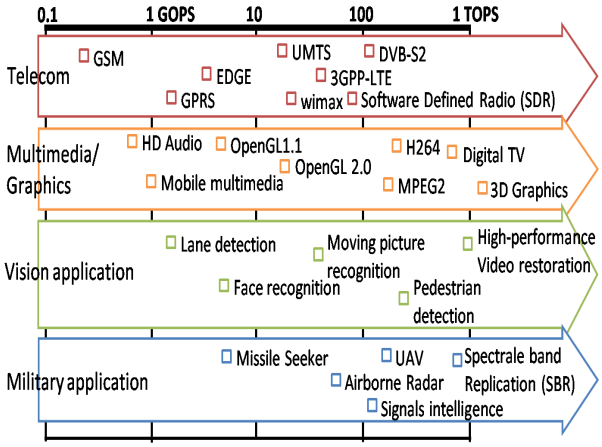


Figure 1.1: Different embedded applications performance requirements grouped by application domain (GOPS: Giga Operations Per Second; TOPS: Tera Operations Per Second) [56].

decomposed into multiple concurrent threads, where each thread or task is composed of a group of instructions. This coarse-grain parallelism is called *Thread-Level Parallelism (TLP)*. A concurrent algorithm can perfectly well execute on a single core, but in that case will not exploit any parallelism. Thus, to exploit the Thread-Level Parallelism and reach higher performances, a multi-threaded application should execute on a platform with multiple cores. In addition to ILP and TLP, there is the data-level parallelism (DLP). The DLP is expressed using special instruction set extensions (SSE, SSE2, SSE3, SSE4, AVX, Altivec, Neon) and executed by SIMD-like microprocessor architectures [139, 64]. The DLP is expressed inside each thread.

A thread is composed of a group of instructions. Part of these instructions is executed one-time sequentially, and the other part is executed multiple times iteratively. The latter refers to program loop codes. In fact, most of the thread execution time is spent in loop codes. One technique of loop optimization is called *loop parallelization* [5]. It can be done automatically by compilers or manually by inserting parallel directives like OpenMP [106]. OpenMP is a method of parallelization (mainly for loops) whereby the master thread forks a specified number of slave threads, and a task is divided among them. Then, the threads run in parallel, with the runtime environment allocating threads to different cores. In this case, the loop region is considered as the parallel region. According to Amdahl's law [7], a program or thread is composed of a serial region S and a parallel region P. Let s be the execution time of the S region, p the execution time of the P region and n the number of cores, then the maximum possible acceleration obtained by parallelizing the loops of a thread is equal to:

$$A = \frac{(s+p)}{(s+p/n)} \tag{1.1}$$

In other words, whatever the number of cores is, the execution time of a program is always limited by its serial region. Thus, the parallel architecture must execute effectively the sequential operations. Nevertheless, Gustafson's law [59] restrains the conclusions of Amdahl's law. He noticed that the parallel region is composed of loops that process the data. Thus, if the number of data to be processed increases, the contribution of the serial region reduces with the n number of cores

used. Let a be the size of the parallel region allotted to each processor, the acceleration becomes:

$$A = \frac{(s + a \cdot n)}{(s + a)} \tag{1.2}$$

Consequently, the more the size a is important, the more the acceleration tends towards the number of cores n, which is the maximum acceleration. Thus, the maximal parallelism can exceed the limit established by Amdahl's law if the quantity of data to be processed by each core is increased.

If the size of input data is known in advance, then optimal static thread partitioning can exist on a given MPSoC architecture. However, an important feature of future embedded computation-intensive applications is the dynamism. Algorithms become highly data-dependent and their execution time depends on their input data, irregular control flow, and auto-adaptivity of the applications. Typical examples of dynamic algorithms are 3D rendering [97], high definition (HD) H.264 video decoder [130], and connected component labeling [45, 77]. The computation time of the connected component labeling algorithm depends on the size and the number of handled objects. Figure 1.2 shows the execution time taken to analyze a complete video sequence with consecutive frame images. It is clear that this algorithm is highly data-dependent and the execution time varies up to 300% depending on the image content. Hence, the loop parallelism also varies according to Gustafson's law [59].

For this type of applications, an optimal static partitioning on an MPSoC cannot exist since all the tasks processing times depend on the input data and cannot be known offline. [22] shows that the solution consists in dynamically allocating tasks according to the availability of computing resources. Global scheduling maintains the system load-balanced and supports workload variations that cannot be known offline. Moreover, the preemption and migration of tasks balance the computation power between concurrent real-time processes. If a task has a higher priority level than another one, it must preempt the current task to guarantee its deadline. Besides, the preempted task must be able to migrate on another free computing resource to increase the performance of the architecture.

In summary, future high-end embedded applications are:

- Massively parallel with a high-degree of Thread-Level parallelism (TLP), generated from application decomposition and automatic/manual loop parallelization
- Dynamic with variable threads execution time

Thus, MPSoC architectures should respond to the embedded applications requirements, as well as meet the embedded systems constraints (power, die area, reliability, etc...).

1.2 MPSoC architectures: state of the art

MPSoC design is the art of choosing the best system components that generate together the best performance with the best transistor/energy efficiency. Thus, designers must take several design parameters into account, such as the number of cores and their types (monothreaded, multithreaded, VLIW, ISA, etc...), the interconnection networks type (bus, mesh, NoC, etc...), the memory system hierarchy, the choice of HW IP accelerators, the programmability, and even the integration technology (2D, 3D), that will all best meet the requirements of a specific embedded application.

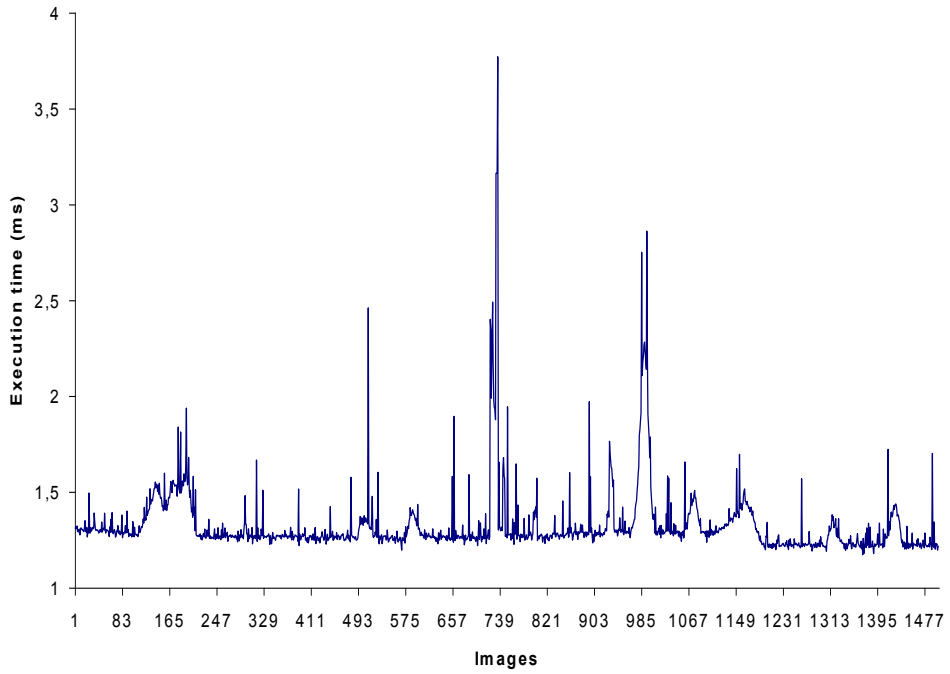


Figure 1.2: Execution time of the connected component labeling algorithm on a video sequence with an Intel Pentium 4 Xeon processor (2.99 GHz) [151].

In fact, future embedded applications necessitate more and more computing power in the order of TOPS (Tera Operations Per Second) as was shown in Figure 1.1. The main reason for this high performance demand is to meet the high expectations of the end-user from his embedded device. This device runs several applications at the same time, and since most of the applications are real-time, this renders their behavior more dynamic and data-dependent.

So it is clear that depending on the application domain and requirements, there is no general-purpose solution that can be efficient. This is why new optimized MPSoC architectures should be designed for each specific application domain. To have a grasp of the variety of architectures, it is sufficient to have a look at surveys done in [158, 23, 47, 131], which list and compare most of the MPSoC architectures that exist in the literature. In the next sections, we will compare different MPSoC architectures' characteristics, then we will provide a classification for the overall MPSoC architectures' space based on the embedded systems requirements discussed in section 1.1, and finally we will conclude which solutions are relevant for future dynamic embedded applications.

1.2.1 Characteristics

MPSoC architectures have different properties that characterize them. We identify four properties: cores' organization (symmetric/asymmetric), cores' similarity (homogeneous/heterogeneous), cores' type (monothreaded/multithreaded), and memory organization (shared/distributed memory). Each property leads to different performances depending on its utilization context and em-

bedded application requirements. In this part, we will explore the characteristics of each property.

1.2.1.1 Symmetric v/s Asymmetric

Symmetric architectures consist of homogeneous processing cores that execute both control and computing tasks. Thus, the same core allocates the next tasks to execute. The control is normally done through the intervention of the OS scheduler, whether periodically or in response to events. This implies that the computing tasks are always preempted by control tasks, which induces some noise in the execution behavior. In addition, the control tasks do not require lot of computing resources as the computing tasks. This means that the homogeneous cores are over-dimensioned for the control tasks, hence not transistor/energy efficient. However, symmetric architectures are scalable since there is no sources of centralization in the architecture that cause contention.

On the other hand, asymmetric architectures consist of one (sometimes several) centralized or hierarchized control core, and several homogeneous or heterogeneous cores for computing tasks. The control core handles the tasks scheduling and migration on the computing cores. Asymmetric architectures have usually an optimized architecture for control. This distinction between control and computing cores renders the asymmetric architecture more transistor/energy efficient than symmetric architecture. However, one main drawback of asymmetric architectures is their scalability. The centralized core is not able to handle more than a specific threshold number of computing cores due to reactivity reasons.

1.2.1.2 Homogeneous v/s Heterogeneous

If all computing cores are the same, then the MPSoC architecture is homogeneous, otherwise it is heterogeneous. By default, all symmetrical architectures are homogeneous. Most designs targeting desktops, laptops and servers are homogeneous, but in the embedded sphere, heterogeneity is more common. A homogeneous architecture has the advantage of being flexible, and simpler to program, analyze and allocate resource than a heterogeneous architecture. In case of dynamic embedded applications, load balancing is easily done between the cores. In addition, it is application-independent since the core can execute any types of tasks. From a hardware point of view, it is simpler to design since it is build out of just one kind of component duplicated across the chip. This has a direct impact on increasing the MPSoC architecture's reliability and its chip's yield rate.

On the other hand, heterogeneity is the key factor for meeting the exact transistor and energy efficiency constraints for a chip in a specific embedded application domain. This is why we tend to see more diverse MPSoC solutions in embedded systems area. However, software challenges (development time, portability, programming tools) for heterogeneous architectures are enormous. For instance, load balancing between heterogeneous core architectures through tasks migration is still a hot research topic.

1.2.1.3 Monothreaded v/s Multithreaded

A monothreaded core executes only one thread. Thus, the only parallelism it can exploit is the instruction-level parallelism (ILP) if it can exploit the so-called vectorization with SIMD instructions. ILP is extracted from a sequential program, offline by the compiler or online by the hardware. However, ILP found in a conventional instruction stream is limited. ILP studies that allow branch

speculation for a single control flow have reported parallelism of around 7 instructions per cycle (IPC) with infinite resources [78, 156], and around 4 IPC with large sets of resources (e.g. 8 to 16 execution units) [29]. Usually in embedded systems, the number of execution units is less than 8, which means the IPC is also lower. Furthermore, a monothreaded core is stalled for a significant amount of time, up to 75% of time [74], due to long latency events such as cache misses that cause access to the DDR2 memory. Therefore, monothreaded cores cannot exploit the totality of their execution resources on each execution cycle.

A possible solution to the low IPC is to exploit the thread-level parallelism (TLP) at the hardware level by using multithreaded cores [147]. A multithreaded core provides the necessary hardware resources to execute several threads concurrently within a single pipeline. Thus, all the threads share the core resources in order to maximize its utilization, hence the IPC rate. In addition, during a long latency event, a multithreaded core can hide this latency by executing instructions from another thread context. Therefore, sufficient instructions need to be obtained to mask the long latencies and increase the pipeline utilization. All these advantages come at the expense of increasing the die area, which can range from 5% till 60%, depending on the core original complexity. More detailed explanations on the importance of multithreaded processors are described in chapter 2.

1.2.1.4 Shared memory v/s Distributed Memory

When comparing the memory organization of MPSoC architectures, we mainly refer to the L2 cache memory (to our knowledge, L3 cache memory is not used for embedded MPSoCs). The L1 cache memory is usually private for each core, and the L2 cache memory can be private per core or 'logically' shared between all the cores. The former refers to a distributed memory architecture, and the latter to a shared memory architecture. A shared memory eases the software programming of the architecture, since all the instructions/data are shared by all the cores. Inter-core communications and synchronizations are performed implicitly through the shared L2 cache memory. A special cache coherency unit guarantees the coherency between the private L1 cache memories and the shared L2 cache memory. However, the L2 cache memory access can be penalizing when there is a concurrency between all the cores while accessing the same memory region. A shared L2 memory improves the memory utilization, since each task can use all the available memory space. This means that the L2 cache memory size should be big enough in order to avoid frequent off-chip memory accesses. But, the fact that all the tasks are sharing the same memory space makes the execution time of the tasks not predictable. Finally, a shared memory facilitates load balancing and task migration between the cores.

On the other hand, a distributed memory with a private L1 and L2 cache memory per core is an interesting solution since it is simpler to implement. Communication between the cores is done explicitly through message passing. This solution is penalizing because of the time taken to build a message and the communication latency with a distant core that should pass by all the memory levels. Furthermore, the memory utilization is not optimal, because the memory resources demand of each task is different. Finally, in case of dynamic applications, load balancing is not a good option for distributed memory architectures, since all the task context should be moved from one private memory to the other.

1.2.2 Classification

In this section, we provide a classification of the MPSoC architectures for embedded systems that currently exist in the literature, which is shown in Figure 1.3.

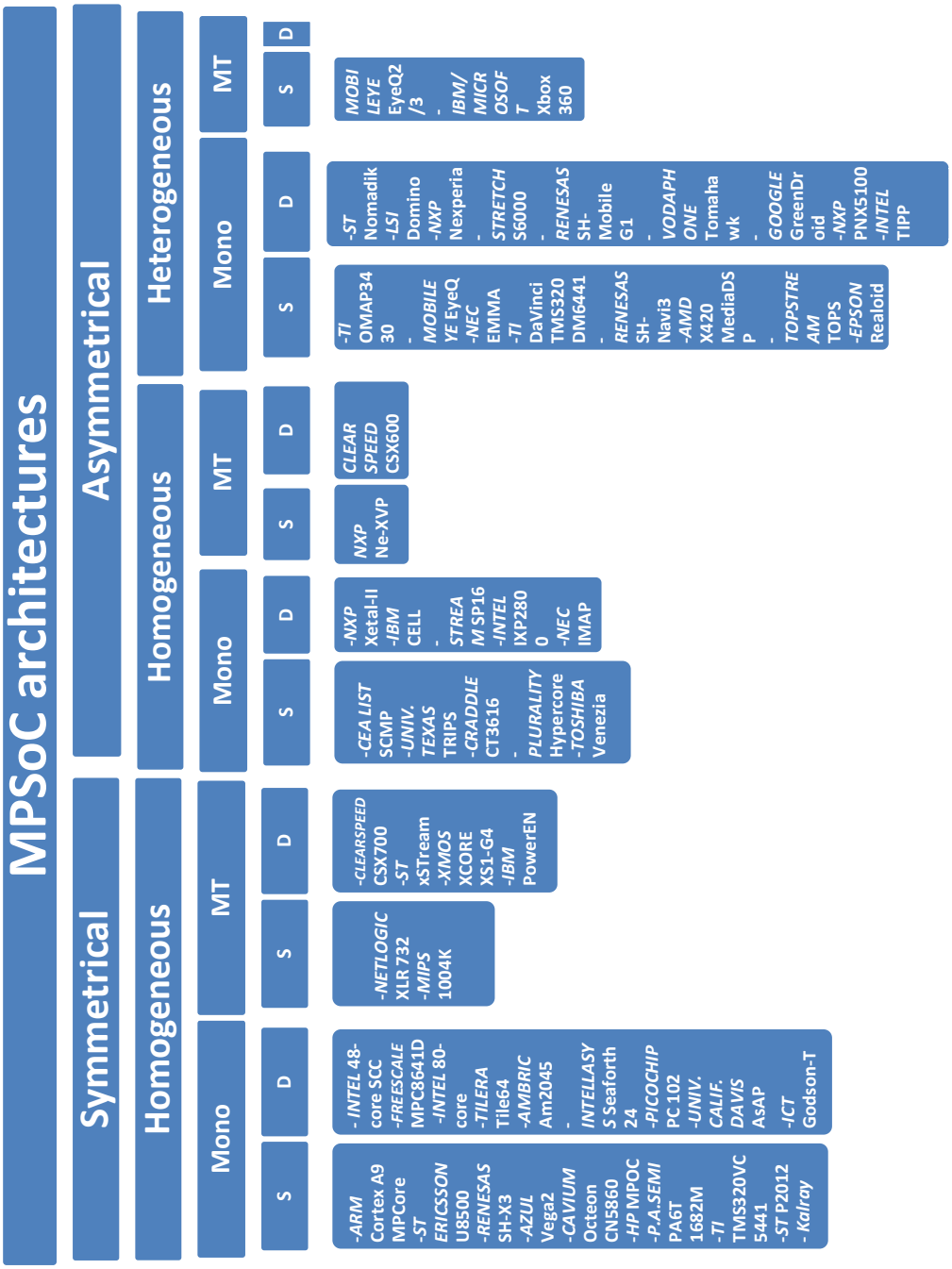


Figure 1.3: MPSoC classification into 3 big families: Symmetric, Asymmetric Homogeneous, Asymmetric Heterogeneous.

MPSoC architectures are classified based on their characteristics discussed in Section 1.2.1. First, they are divided into 2 main architectural families: *Symmetric* and *Asymmetric*. Then, each family is categorized by the similarity of the cores: *Homogeneous* and *Heterogeneous*. Afterward, two categories are distinguished depending on the computing core type: *Monothreaded* and *Multithreaded*. And finally, MPSoC architectures are classified by their memory architecture, which can be a *distributed memory* or a *shared memory*.

In the next parts, we will explore in more details the characteristics of these 3 categories, and then we conclude why the asymmetric homogeneous MPSoC category is an interesting solution for future dynamic embedded applications.

1.2.2.1 Symmetric MPSoC architectures

Symmetric MPSoC architectures are homogeneous. Therefore, their main advantage is the smaller investment in HW and SW development times. First, HW designers can take advantage of the chip homogeneity and reuse the validation scripts for one core to revalidate all the chip functionality. This implies faster chip design and validation. Second, a symmetric MPSoC architecture is a well-known architecture for the SW community, since it resembles to a SMP system. Software developers have developed lot of programs (applications, OSes, libraries) for SMP systems, which are rapidly portable to any SMP architecture. For example, the Linux SMP OS [82] can execute on ARM Cortex A9 [9] [83], Tilera Tile64 [142, 19], Intel 40-core SCC (Single-chip Cloud Computer) [65], and MIPS 1004K [90], just by porting the low-level processor dependent code to the corresponding architecture. All the others OS modules, such as communication, scheduler, file handling, memory management, and others, are reused for all the architectures. This implies that legacy codes are easily portable to new SMP architectures with little investment in software development. For instance, a SW program running in a distributed network with MPI (Message Passing Interface) communication can be easily ported to a symmetric MPSoC with distributed memory. A typical example is the Intel SCC [69] and Tilera Tile64 [126]. In addition, programs with OpenMP support can be easily ported to a symmetric MPSoC with shared memory. A typical example is the ARM Cortex A9 MPCore [24, 40], and MIPS 1004K [34]. Moreover, the homogeneity of the architecture facilitates the thread migration between the cores, which is suitable for dynamic embedded applications. In summary, the symmetric MPSoC architecture is flexible and can be easily adapted to any embedded application domain just by modifying the software program, which means faster time-to-market.

On the other hand, these advantages come at the expanse of the chip's transistor and energy efficiency. The symmetric MPSoC chip is used in a specific embedded environment. The cores' architecture is normally over-dimensioned for control and computing, so that they meet the performance requirement of all the applications. In reality, some tasks need only a small portion of processing power where in fact they will be running on the over-dimensioned core, hence adding more die area and consuming more energy. For instance, the Intel SCC [69] integrates 48 Pentium class IA-32 cores [123], which are highly inefficient for control tasks and most of the computing tasks. Another disadvantage is the architecture's reactivity. A SW OS takes lot of time to process a preemption event, since it must pass by several SW layers before a response can be sent. In addition, tasks scheduling is not deterministic, since it depends on the number of tasks to schedule and the number of cores.

The majority of the symmetric MPSoC architectures have monothreaded cores as can be seen

clearly in Figure 1.3. In some situations, long latencies might occur that can stall the cores' pipeline up to 75% of its execution time [74]. Typical sources of long latencies are cache misses that require access to off-chip main memory, tasks accessing shared resources such as I/O, tasks waiting for a HW IP to finish execution, and others. Some techniques might be implemented, such as thread level speculation (TLS) [52] and data prefetching [30] in order to lower the stall time. However, those techniques are not yet mature for embedded systems [116, 58] because they are complex to implement and they consume lot of energy for possible useless computation. Therefore, HW architects tend to use multithreaded (MT) processors in order to mask those long latencies with instructions execution of other thread contexts, hence increasing the symmetric MPSoC architecture's efficiency. For instance, the MIPS 1004K [90], NetLogic Microsystems XLR 732 [101], ClearSpeed CSX700 [33], and IBM PowerEN [28] are some commercial examples that were recently released. These architectures are suitable for applications that have sufficient multithreaded workloads. For example, the ClearSpeed CSX700 is used as an accelerator for HPC applications, and MIPS 1004K is used in the EyeQ3 [96] chip for embedded vision applications, where both applications domain are multithreaded.

1.2.2.2 Asymmetric heterogeneous MPSoC architectures

Asymmetric heterogeneous MPSoC architectures consist of one or several central control core that handles multiple heterogeneous computing cores. In a specific embedded application domain, each computing core is designed to perform dedicated functions. Therefore, the computing cores are not over-dimensioned, which implies better transistor and energy efficiencies. For instance, the Seiko-Epson inkjet printer Realoid SoC [124] incorporates 6 customized Tensilica Xtensa LX cores [141] and 1 NEC V850 control core, where each Tensilica core is customized for a unique step in the inkjet image processing chain. For mobile applications, energy efficiency is a major issue because it determines the mobile terminal's autonomy. Several MPSoC architectures exist such as ST Nomadik [107], TI OMAP3430 [86], and Google Greendroid [53]. The latter is designed for an Android platform and incorporates several Conservative cores or C-cores [149], where each core is designed to accelerate a specific hotspot function in an application. Another embedded application example is the Advanced Driver Assistance Systems (ADAS), which are systems to help the car driver in its driving process. For these systems, Mobileye fabricates vision MPSoCs such as EyeQ [134], EyeQ2 and EyeQ3 [96] families. Those architectures consist of one controller core and several custom HW IPs called Vision Computing Engines (VCEs). Each VCE has a specific role in the vision process application such as filtering, tracking, video codec, and others. All these MPSoC examples show the heterogeneity of the architectures, which render them more transistor/energy efficient for a particular application domain. For example, Intel's TCP/IP processor is two orders of magnitude more power-efficient when running a TCP/IP stack at the same performance as a Pentium-based processor [26].

However, the main drawback of the asymmetric heterogeneous MPSoCs is their programmability and resources management. Since each new architecture consists of different types of cores and IPs, a significant time on software portability and chip programming has to be invested. In addition, the heterogeneity of the computing cores makes it very difficult (until this date) for the control core to perform load balancing between the computing cores through task migrations.

There is not a big number of MPSoC architectures that utilizes MT processors. MT processors

have been recently investigated, to increase the cores' efficiency for asymmetric architectures. For example, the first generation EyeQ chip used 2 monothreaded ARM946E cores. The cores suffered from low pipeline utilization of only 0.32 IPC due to cache miss rates and bus contention bottlenecks. In their next generation EyeQ2 chip, Mobileye HW designers decided to use the multithreaded MIPS 34K [92] cores with 4 HW thread contexts instead of the monothreaded ARM946E core. The core's IPC increased to 0.9, and the overall chip performance increased 6 times compared to EyeQ. This is because the multithreaded cores were able to overlap the long stall latencies by executing instructions from other thread contexts. Also, the high performance gain is due to increasing the processors' clock frequency from 110 MHz to 330 MHz.

1.2.2.3 Asymmetric homogeneous MPSoC

Asymmetric homogeneous MPSoC architectures consist of one or several central control core that handles multiple homogeneous computing cores. This solution combines the advantages of symmetric and asymmetric heterogeneous MPSoCs. A specialized core for the control renders the architecture more transistor/energy efficient than symmetric MPSoC. Furthermore, the various hardware abstraction layers, between the OS and the hardware, penalize performance and overall system reactivity. This generates critical sections during hardware/software communication. Use of dedicated hardware components for control is thus vital to the MPSoC architecture. For instance, the HW task scheduler of the Ne-XVP [63] chip from NXP Semiconductors takes around 15 cycles overhead. In addition, due to the increasing diversity of applications that an embedded system should be able to process, fixed hardware solutions are more and more replaced by programmable solutions, pushing the flexibility to software. Thus, asymmetric homogeneous MPSoCs are easily programmable and can function in a multi-application domain. For example, the IBM Cell chip [120], which is composed of one *Power Processor Element* (PPE) and multiple *Synergistic Processing Elements* (SPE) [55], is used in application domains such as gaming (PlayStation 3), video processing, blade server, home cinema, distributed computing and others. It is programmed using C language with multithreading support for dispatching software threads to the SPEs. The Plurality Hypercore chip [110] is used for real-time video processing, image rendering, software-defined radio, and packet processing application domains. Its programming model relies on a simple, Task Oriented Programming model, which is directly supported by hardware as opposed to the intermediation of an operating system layer. Moreover, the support for dynamic load-balancing depends on the memory architecture. The Cell has a distributed memory architecture, where each SPE has its local memory implemented as a software cache. In general, dynamic load-balancing and thread migration is inefficient in a distributed memory architecture. On the other hand, an asymmetric homogeneous MPSoC with a shared memory is highly efficient for load-balancing. For instance, Plurality Hypercore [110], TOSHIBA Venezia [95] and CEA LIST SCMP [151], support dynamic load-balancing between all the homogeneous cores, which make them suitable for dynamic embedded applications.

One main disadvantage of asymmetric homogeneous MPSoC architectures is their scalability. The centralized control core suffers from contentions when the number of computing cores increases, hence the scheduling overhead of the central core also increases. This means that the computing cores are stalled while waiting the scheduling decision of the control core. For instance, SCMP [151] can support up to 32 computing cores before starting to suffer from scheduling performances

degradation. Plurality claims that Hypercore processor can support up to 256 cores [111], however there are no publicly available benchmarks that show the linearity of the performance gain. Ne-XVP [63] from NXP Semiconductors can support up to 16 computing cores.

Multithreaded processors are still rarely used in these types of architectures. We could identify 2 architectures that use MT computing cores: NXP Semiconductors Ne-XVP [63] and the ClearSpeed CSX600 [102]. The Ne-XVP architecture is 16 times more efficient in silicon area and power than off-the-shelf TriMedia TM3270, for applications such as H.264 decoding.

1.2.3 Synthesis

Each MPSoC family has its advantages and disadvantages. However, we should never dissociate an MPSoC architecture from the targeted embedded application domain. In this thesis, we are targeting future dynamic embedded applications as explained in section 1.1. These applications are *massively parallel* with a high-degree of Thread-Level parallelism (TLP) and *dynamic* with variable data-dependent thread execution time. In Table 1.1, we compare the 3 MPSoC families.

	Symmetric	Asymmetric Heterogeneous	Asymmetric Homogeneous
Programmability / Flexibility	++	-	+
Transistor / Energy efficiency	-	++	+
Scalability	++	-	-
Dynamic load-balancing	+	-	++

Table 1.1: Characteristics comparison of the 3 MPSoC families: Symmetric, Asymmetric Homogeneous, Asymmetric Heterogeneous.

As can be observed from Table 1.1, the asymmetric homogeneous architecture bridges the gap between the symmetric and asymmetric heterogeneous MPSoCs families. It combines the ease of programmability and flexibility from symmetric MPSoCs. Probably the most difficult thing for hardware designers as they move to MPSoC design is that they must worry about software design from the beginning. The hardware architect cannot simply create a machine that is hard to program, since the time gained to design the MPSoC will be rapidly lost by the software application development. In addition, asymmetric homogeneous MPSoC are transistor/energy efficient because of the separation between control and computing cores. And finally, the implementation of a central control core allows for fast dynamic load-balancing between the computing cores with a very small overhead.

All these characteristics lead us to choose the *asymmetric homogeneous MPSoC* as the best architecture design adapted for future dynamic embedded applications despite its scalability limitation. In particular, an asymmetric homogeneous MPSoC, called SCMP [151], will be retained for the rest of this thesis as the architecture of reference for experimentations and will be explained in

more details in section 1.3. SCMP is designed to process embedded applications with a dynamic behavior. In addition, a solution to SCMP scalability limitation will be proposed in chapter 4.

1.3 SCMP: an asymmetric MPSoC

The SCMP architecture is an asymmetric homogeneous MPSoC architecture, which is proprietary of CEA LIST [150]. SCMP stands for Scalable Chip MultiProcessor. It is designed as a compute-intensive accelerator for tasks that are inefficiently processed by the host cores such as dynamic applications. It is seen by the host CPU as a coprocessor, as shown in Figure 1.4. The software operating system (OS) running on the host CPU is commonly used for general-purpose processing or interface management. All control of intensive parallel processes with dynamic behavior must nevertheless be performed by an efficient control system. Therefore, in this context, an asymmetric architecture is the most adequate solution as was previously explained in section 1.2. In this section, we will present an overview of the SCMP architecture and its components, then we will explore in more details the programming and execution models of SCMP, and finally we will show a typical functionality processing example.

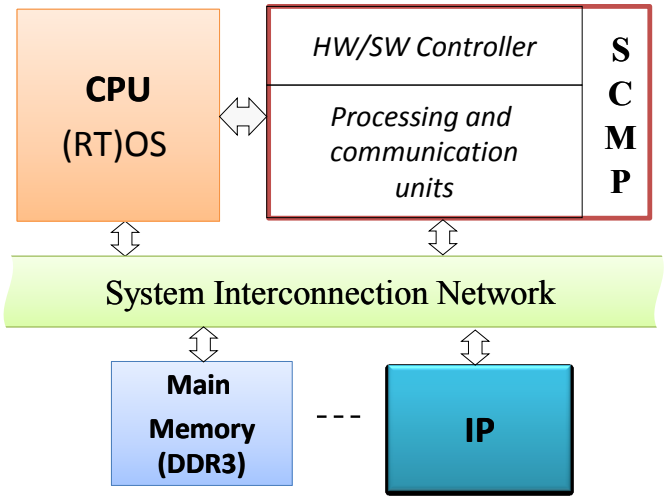


Figure 1.4: SCMP system architecture: the host CPU dispatches massively parallel dynamic applications to SCMP for optimal transistor/energy efficient acceleration.

1.3.1 Architecture overview

SCMP is composed of three main parts as shown in Figure 1.5: a *central controller*, *processing elements*, and a *memory system*.

Central Controller : The *central controller* is an efficient control component for the overall SCMP architecture. It holds all the information of the application tasks in a special local memory. It dynamically determines the list of eligible tasks to be executed on multiple *processing elements*,

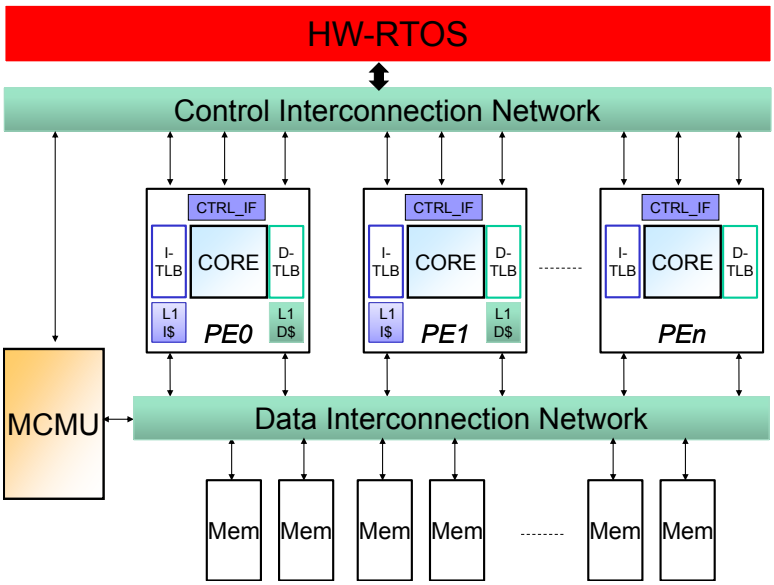


Figure 1.5: SCMP architecture.

based on control and data dependencies. Then, task allocation follows online global scheduling, which selects real-time tasks according to their dynamic priority, and minimizes overall execution time for non-real-time tasks. In addition, the *central controller* manages memory allocations and the exclusive sharing of physically distributed and logically shared *memory system*. It also prefetches tasks' code in these memories so that the beginning of the task execution does not suffer from memory latencies. There exist two implementations for the *central controller*: hardware and programmable RTOS (real-time operating system). The HW-RTOS is called OSoC (Figure 1.6(a)), which stands for Operating System accelerator On Chip. It has RTOS specialized components implemented in hardware such as a scheduler unit. It is described in more details in [150]. However, besides its high reactivity, the main disadvantage of OSoC is its programmability. In fact, each time we need to implement a new scheduling protocol, the hardware IPs should be modified in VHDL. Thus, a more flexible implementation is the programmable RTOS, which is called CCP (Figure 1.6(b)). It consists of an optimized processor for control, such as AntX, which is a small RISC 5-stage pipeline core optimized for control and is proprietary of CEA LIST. Most of the RTOS functionalities are implemented in software. In this thesis, we will use the programmable central controller for our experimentations.

Processing Elements : A typical *processing element* or PE in SCMP is composed of 4 components: a Control Interface unit (CTRL_IF), an execution core, Translation Lookaside Buffers (TLBs), and cache memories.

The CTRL_IF is the interface between the PE and the *central controller*. Through this interface, it receives tasks execution/preemption/stop demands from the central controller and updates its execution status for proper global scheduling.

The homogeneity and heterogeneity of the PEs is identified by its *execution core*. Heterogeneous cores have better chip area and power efficiency for the SCMP system. Heterogeneity may

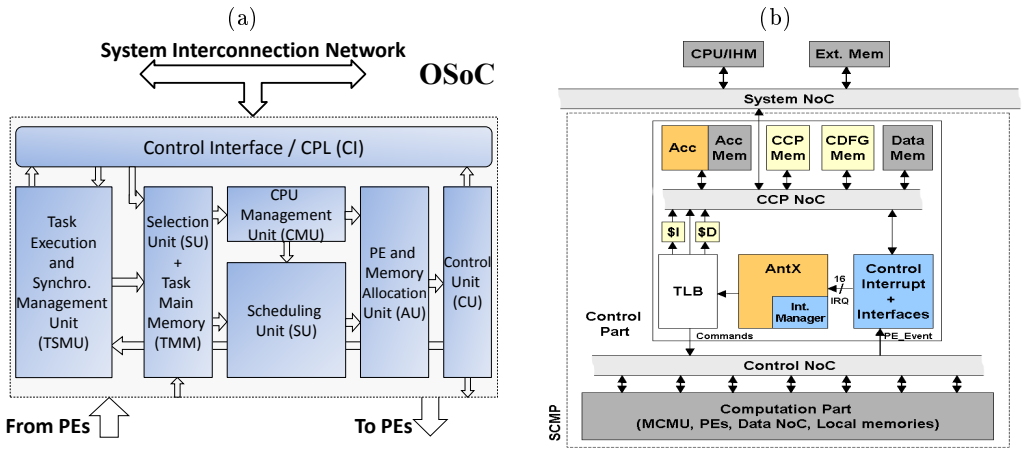


Figure 1.6: a) HW RTOS OSoC [150] b) Programmable RTOS CCP.

be implemented either in software or hardware. It is possible to implement general-purpose or dedicated processors (DSP, VLIW, etc.), coarse-grained reconfigurable resources (ADRES [87], DART [38], XPP [13], etc.), time-critical I/O controllers (video sensor, etc.), and dedicated or accelerated hardware components (DMA, IP). These dedicated units can take part in critical processes, for which no programmable or reconfigurable solution with sufficient computing performances exists. As mentioned earlier, each task is executed by a predefined execution core. This implies there is **no support for task migration and load-balancing between heterogeneous PEs**. Thus, in this thesis, we will use a SCMP system with only homogeneous programmable execution cores as was discussed in section 1.2.3. Typical core architectures are MIPS1, MIPS32, LEON3, PowerPC, SPARC V8, and others.

In SCMP, each task has its own virtual address space in the memory. Therefore, each memory access from the execution core maps in the virtual address space of the task. To map the correct physical memory region, each PE has a TLB for instruction (I-TLB) and data (D-TLB) for virtual to physical memory translation. The TLBs are then connected to private L1 Instruction cache memory (I$) and Data cache memory (D$).

Memory system : The on-chip *memory system* is a 2-level memory hierarchy: a private L1 I$ and D$ for each PE, and a logically shared physically distributed L2 memory. They are connected by a data interconnection network that transfers the memory requests of all the PEs to the multibanked shared memory. The data interconnection network is a multibus. Initially, all the task codes are prefetched into the L2 memory. Data is also prefetched via DMA engines. Thus, there is no off-chip memory access during task execution on a PE. A special unit called MCMU (Memory Configuration and Management Unit) handles the memory configuration for the tasks. It divides the memory into pages. In addition, MCMU is responsible of managing the tasks' creation and deletion of dynamic data buffers at runtime, and synchronizing their access with other tasks. There is one allocated memory space per data buffer. A data buffer identifier is used by tasks to address them. Each task has a write exclusive access to a data buffer. Since all tasks have an exclusive access to data buffers, data coherency problems is eliminated without the need for specific coherency mechanisms. A data buffer access request is a blocking demand, and another task can

read the data buffer when the owner task releases its right. Multiple readers are possible even if the memory latency will increase with the number of simultaneous accesses.

In section 1.3.2, we will describe in more details the functionality and interaction of these units.

1.3.2 Programming models

The programming model for the SCMP architecture is specifically adapted to dynamic applications and global scheduling methods. The proposed programming model is based on the explicit separation of the control and the computing parts. As depicted in Figure 1.7, each application must be manually (the tool chain is still under development) parallelized and cut into different tasks, from which explicit execution dependencies are extracted. Thus, computing tasks and the control task are extracted from the application, so as each task is a standalone program. The greater the number of independent and parallel tasks that are extracted, the more the application can be accelerated at runtime.

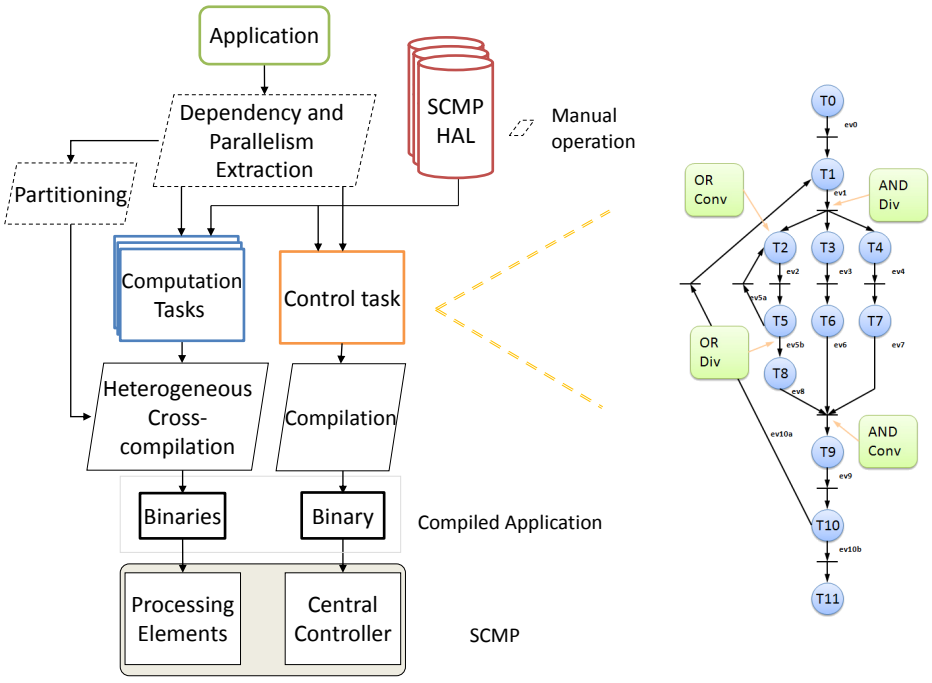


Figure 1.7: SCMP programming model and an example of a typical CDFG control graph.

The control task is a Control Data Flow Graph (CDFG) extracted from the application (Petri Net representation), which represents all control and data dependencies between the computing tasks. The control task handles the computing task scheduling and other control functionalities, like synchronizations and shared resource management for instance. A specific and simple assembly language is used to describe this CDFG and must be manually written. In addition, a specific compilation tool is used for the binary generation from the CDFG. Once each application and thread has been divided into independent tasks, the code is cross-compiled for each task. For

heterogeneous computing resources, the generated code depends on execution core type.

For the computing cores, a specific Hardware Abstraction Layer (HAL) is provided to manage all memory accesses and local synchronizations, as well as dynamic memory allocation and management capabilities. With these functions, it is possible to carry out local control synchronizations or to let the control manager taking all control decisions. Concurrent tasks can share data buffers through local synchronizations handled by the MCMU (streaming execution model), or wait for the central controller decision before reading the input data (control flow execution model). Each task is defined by a task identifier, which is used to communicate between the control and the computing parts.

In SCMP, two programming models are supported: a *control-flow* and a *streaming* programming model.

1.3.2.1 Control-flow programming model

The control-flow programming model allows the execution of a task when all the previous dependent tasks have finished their execution, and therefore have produced their intermediate results (Figure 1.8). The task execution order is described in the CDFG that is handled by the control unit. The task execution follows the run-to-completion model. Therefore, they cannot be preempted by data or control dependencies. During its execution, a task cannot access data buffers not selected at the extraction step. Consequently, the control-flow programming model eliminates data coherency problems, thus there is no need for specific coherency mechanisms. This constitutes an important feature for embedded systems, since the MPSoC architecture is accordingly simplified.

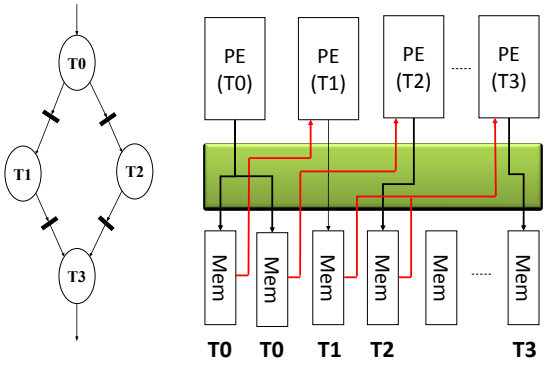


Figure 1.8: Control-flow execution model. Reads and writes are non-blocking. Each arrow represents an interconnection through the data network between a Processing Element (PE) and a data buffer stored in local memories (Mem). Dark arrows are read/write accesses, whereas gray arrows represent read-only accesses. Task execution requires only data produced by tasks that have finished their execution on a PE.

1.3.2.2 Streaming programming model

In the streaming programming model, a task suspends/resumes its execution based on data availability from other tasks. It follows the streaming/dataflow execution model as shown in Figure 1.9. When a data is produced by Task A, then Task B resumes its execution. When data is consumed by Task B, then it suspends its execution. Each task has the possibility to dynamically allocate or

deallocate buffers (or double buffers) in the shared memory space through specific HAL functions. An allocated buffer is released when a task asks for it and is the last consumer. A buffer cannot be released at the end of the execution of the owner task. Dynamic right management of buffers enables a dataflow execution between the tasks and is handled by the MCMU.

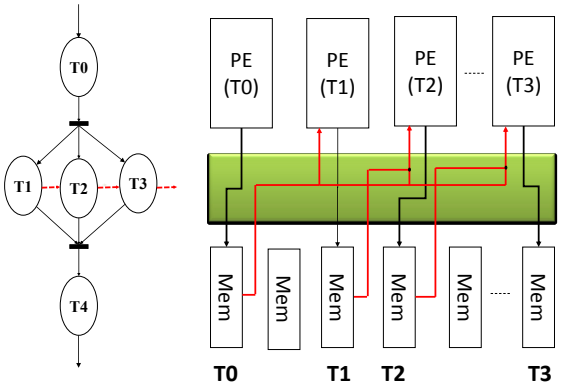


Figure 1.9: Streaming execution model. Each arrow represents an interconnection through the data network between a Processing Element (PE) and a data stored in local memories (Mem). Dark arrows are read/write accesses, whereas gray arrows represent read-only accesses. Task execution begins as soon as intermediate data are ready. Local synchronization is afforded through a memory management unit.

1.3.3 SCMP processing example

As mentioned earlier, SCMP is used in the context of a co-processor for a host CPU. The host CPU can accelerate applications at runtime. It communicates with SCMP via the *System Interconnection Network* used in the system (Figure 1.4). It can ask for execution of a new application, stop or suspend it, or wake up a suspended application. Multiple applications and multiple instances of the same application can be executed and managed concurrently by SCMP. To execute a new application, the host CPU must load all necessary instructions or data for the application into SCMP local memory. When the transfer is completed, the CPU informs the *central controller* and sends an execution order. After execution of the application, an acknowledgement is sent to the CPU.

When the *central controller* receives an execution order of an application, its Petri Net CDFG representation is built into the *central controller*. Then, the central controller proceeds with the execution and configuration demands according to the application status. They contain all identifiers of active tasks that can be executed and of coming active tasks that can be prefetched. Scheduling of all active tasks must then incorporate the tasks for the newly loaded application. If a non-configured task is ready and waiting for its execution, or a free resource is available, the *central controller* sends a configuration primitive to the Memory Management and Configuration Unit (MCMU).

Based on the task identifier, the MCMU allocates a memory space for the context, the code and the stack of the task. Then, it loads the instruction code related to that task from the off-chip main instruction memory and initializes the context. Configuration of these local memories is sequential

and takes place only once before execution of the task. Once the transfer is finished, the address of the selected memory along with the task identifier is written into the MCMU.

After its configuration, the task is ready to be scheduled and dispatched by the *central controller*. If there is a free PE, the *central controller* sends an execution demand to the selected PE. If the selected task has a higher priority over another task that uses the same type of PE, or a task running on a PE is stalled waiting for producer data (streaming execution model), the *central controller* sends a preemption demand to the selected PE. Then the task execution context is saved. Because all memories are shared, this execution context can be accessed by another PE, thus enabling easy task migration from one PE to another.

When a PE receives an execution request, it asks the MCMU for the translation address table of the task memory through the TLB. This table contains all translations of allocated pages for the context, the code and the stack. With these addresses the PE can begin executing the task. The distribution of such data management units among the PEs allows concurrent communications and data transfers between tasks. I/O controllers are used by the OSoC as other PEs. For example, a data transfer from a DMA implies moving external off-chip data to the task memory. Local transfers can take place, where necessary, via another PE to distribute large amounts of data among other local memories, thereby improving access memory parallelism. Because all data required to execute the task are ready and all synchronization has been completed at the task selection level, the execution then simply consists of processing the data from local memories and storing the result in a memory open to the other tasks.

1.4 Why these MPSoC architectures are not suitable?

In this chapter, we defined the context of our study: *massively-parallel dynamic embedded applications*. These applications are highly parallel. The parallelism can be extracted at the thread level (TLP) and at the loop level. So an application might have more than 1000 parallel threads to be processed in parallel. Therefore, *manycore* architectures are natural solutions for these applications. In addition, the dynamism of those applications requires an efficient MPSoC solution to manage the resources occupation and balance the loads in order to maximize the overall throughput. In this case, we saw in section 1.2.2 that *asymmetric MPSoC* architectures are the best solution for fast and reactive load-balancing. They are also highly transistor and energy efficient because of the separation between control and computing cores. Finally, we concluded that *asymmetric homogeneous MPSoCs* are the best choice for these applications, since load-balancing cannot be done between heterogeneous cores. Thus, we have chosen SCMP as the architecture of reference for experimentations, and it was presented in section 1.3.

Besides, currently existing *asymmetric homogeneous MPSoCs* are not suitable for future massively-parallel dynamic applications. First of all, they are not scalable to the manycore level because the central controller is a source of resources contentions. For instance, SCMP can support up to 32 processing cores before experiencing performances degradation. So, they are not designed for the manycore era and this will be performance limiting. However, manycore chips have limited I/O pins in their chip package, hence limited bandwidth [12]. This implies that the more traffic will be exercised off-chip, the more the cores will be stalled on-chip, hence lower aggregate IPC. Thus, it will be advantageous to explore the benefits of hardware multithreading for future manycore chips,

in order to keep the core as busy as possible and increase the aggregate IPC. In particular, the homogeneous cores that constitute a manycore chip should be as small and efficient as possible [10].

In summary, there exist two suggestions for improvements of the currently existing *asymmetric homogeneous MPSoCs*: scalability and hardware multithreading. In this thesis, we will first investigate the advantages/disadvantages of hardware multithreading in SCMP architecture, and then we will propose a novel solution that will target the manycore era. This solution should tackle the challenges of future massively-parallel dynamic embedded applications.

Chapter 2

Multithreaded processors in embedded systems

Multitasking? I can't do two things at once. I can't even do one thing at once. – Helena Bonham Carter, actress

Contents

Traditional high-performance superscalar processors implement several architectural enhancement techniques such as out-of-order execution, branch prediction, and speculation, in order to exploit the instruction-level parallelism (ILP) of a single-thread sequential program. However, due to the limits of ILP [156], a more coarse-grained solution consists of exploiting the parallelism at the thread level (TLP), where multiple threads can be executed in parallel on multicore processors or concurrently on hardware multithreaded processors.

Embedded processors must have a die size in the order of few mm^2 and most consume in the order of few mW. Thus, they should support simple technology for exploiting ILP, such as pipelining or VLIW. Non-deterministic ILP boosting mechanisms, such as speculative scheduling, should be avoided. In this context, processing a single thread stream often leaves many functional units of the embedded processor underutilized, which wastes leakage power. To compensate the loss in single-thread performance and to increase the transistor/energy efficiency of the embedded processor,

designers are exploiting the parallelism at the thread level (TLP) through the implementation of embedded multithreaded processors [93, 68, 41].

A hardware multithreaded processor [147] provides the hardware resources and mechanisms to execute several hardware threads on one processor core in order to increase its pipeline utilization, hence the application throughput. Unused instruction slots, which arise from pipelined execution of single-threaded programs by a monothreaded core, are filled by instructions of other threads within a multithreaded processor. The hardware threads compete for the shared resources and tolerate pipeline stalls due to long latency events, such as cache misses. These events can stall the pipeline up to 75% of its execution time [74]. Thus, the main advantage of multithreaded processors over other types of processors is their ability to hide the latency within a thread (e.g. memory or execution latency).

Future manycore architectures tend to use small footprint RISC cores [10] as basic processing elements. In this case, more processors can be integrated on a single die while keeping the aggregate cores' energy consumption under a tolerable threshold. Therefore, in our thesis study, we will consider a **5-stage pipeline, in-order, single-issue RISC core**. Then, we will support this core with hardware multithreading.

In this chapter, we will explore and analyze the performance and efficiency of multithreaded processors in embedded systems. First of all, we will provide a classification of the different types of multithreaded processors that exist in the literature. In particular, two multithreading techniques for single-issue cores will be retained: *Interleaved multithreading* (IMT) and *Blocked multithreading* (BMT). These multithreaded architectures should meet the embedded systems requirements and are suitable for manycore architectures. Then, we need to know the surface occupation of each multithreaded processor type (IMT and BMT), and its overhead with respect to the monothreaded core. Hence, we will apply the two multithreading techniques on a small footprint monothreaded core at the RTL level (VHDL), and synthesize all the 3 cores in 40 nm TSMC technology. Finally, we will compare the performance of the monothreaded, IMT, and BMT cores in a typical processor system configuration, and we show the characteristics of each processor type and under which conditions should be used.

2.1 Classification

There are lot of misconceptions when defining the term multithreaded processor. A processor can be regarded as a simple state machine. It contains a context and an execution core. The context stores the state of a process/thread in the program counter, data registers and status registers [31]. The execution core performs computation on the stored state. Thus, the state of the process/thread being executed is advanced by the execution core over the time.

Given definitions for a thread context (TC) and an execution core, it becomes possible to classify different types of architecture by relating the number of contexts to the number of execution cores. This is shown in Figure 2.1.

The simplest arrangement is the monoprocessor. It has one context and one execution core (1/1). The multithreaded processor contains multiple contexts, sharing a single execution core (N/1, where N>1). And finally, the multiprocessor has groups of contexts and cores, with one or several contexts per core (M/N, where $M \geq N$ and N>1).

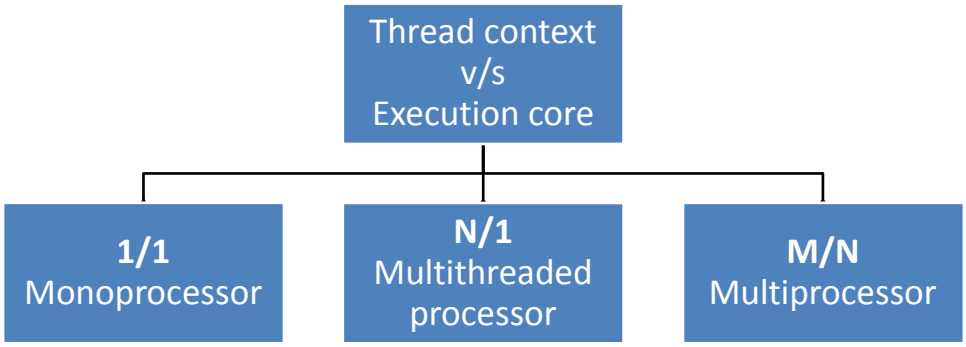


Figure 2.1: Thread context v/s execution cores: Monothreaded, Multithreaded, Multiprocessor.

In section 1.2 of chapter 1, we already discussed multiprocessor architectures or MPSoCs that have monothreaded and multithreaded execution cores.

In this section, we will focus our study on the multithreaded processor. In particular, we will explore the whole design space solution and focus mainly on characteristics that are relevant to the embedded systems requirements: explicit and scalar multithreaded cores will be retained for further analysis. Then, we will discuss two types of multithreading techniques for scalar cores: interleaved and blocked multithreading. Finally, we will present a cost-effectiveness model that will allow us to conclude which are the best multithreaded processor types that are adapted for the embedded systems.

2.1.1 Multithreaded processor design space

There exist 3 main characteristics that identify a multithreaded processor (Figure 2.2): parallelism type, execution core, and instruction issue.

2.1.1.1 Parallelism type

The parallelism type can be explicit or implicit. Explicit multithreading exploits the TLP (thread level parallelism) that are user-defined or OS-defined threads. In other words, threads should be explicitly identified and created in order to be executed by the multithreaded processor. On the other hand, implicit multithreading exploits the TLS (Thread level speculation) of a single-threaded program. Threads are dynamically generated by the processor from single-threaded programs using speculation such as the dynamic multithreading processor [6], trace processor [122], and the speculative multithreaded processors [133], or statically using compiler support such as the multiscalar processor [132] and superthreaded processor [145].

Speculative execution is proposed to provide sufficient instructions even before their control dependencies are resolved. It necessitates register renaming mechanism to allocate a virtual register space to each speculative instruction. The result is higher utilization and statistically better performance for single-threaded programs. Nevertheless, mis-speculations need to be discarded and all effects of the speculatively executed instructions must be disposed. This wastes both energy and execution time on the mis-speculated path. In terms of hardware cost, the number of functional

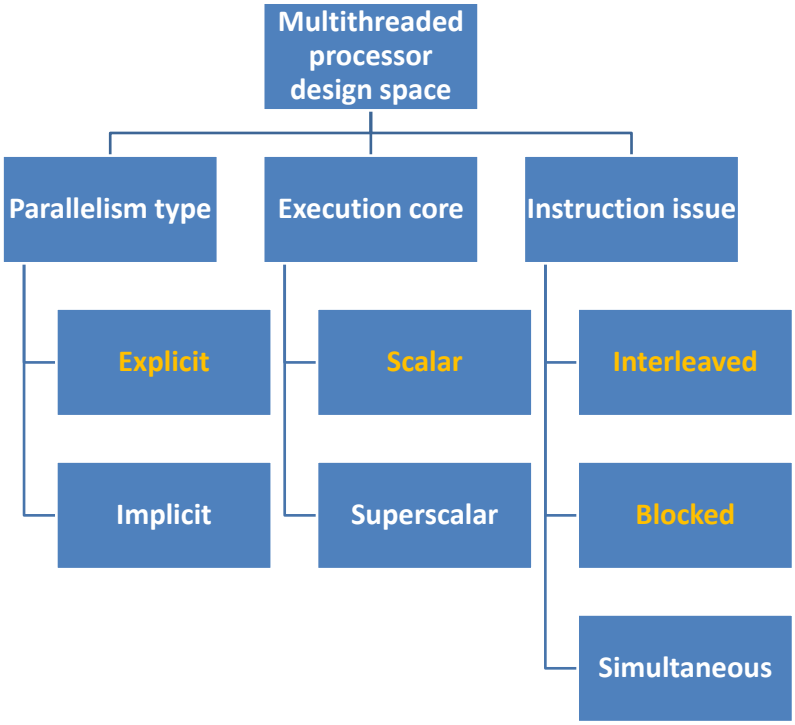


Figure 2.2: Multithreaded processor design space.

units goes up linearly with the required degree of parallelism. Therefore, thread speculation is an interesting solution for general-purpose processors. However, it should be avoided for embedded systems for energy and die area constraints.

In embedded systems, simple techniques should be used to increase the performance. Thus, performance gained from thread level parallelism (explicit multithreading) should compensate the need for instruction level parallelism extraction using speculation (implicit multithreading).

2.1.1.2 Exection core

The execution core can be scalar or superscalar (Figure 2.3). The simplest processors are scalar processors. A scalar processor has one ALU (Arithmetic Logic Unit) and maybe one FPU (Floating-Point Unit). Thus, the maximum theoretical instruction issue is 1 instruction per cycle and the maximum ILP exploited is 1.

A superscalar execution core has multiple redundant functional units (ALU, FPU, multipliers, SIMD, etc...). A typical example is the PowerPC 970 [121], which has four ALUs, two FPUs, and two SIMD units. Multiple instructions from the same thread are issued to multiple functional units. In this case, the maximum theoretical IPC is equal to the maximal number of fetched instructions per clock cycle. However, since the ILP of a single thread is limited, functional units are under-utilized. To compensate this limitation and increase the pipeline utilization, superscalar processors tend to issue multiple instructions from multiple threads simultaneously at every clock

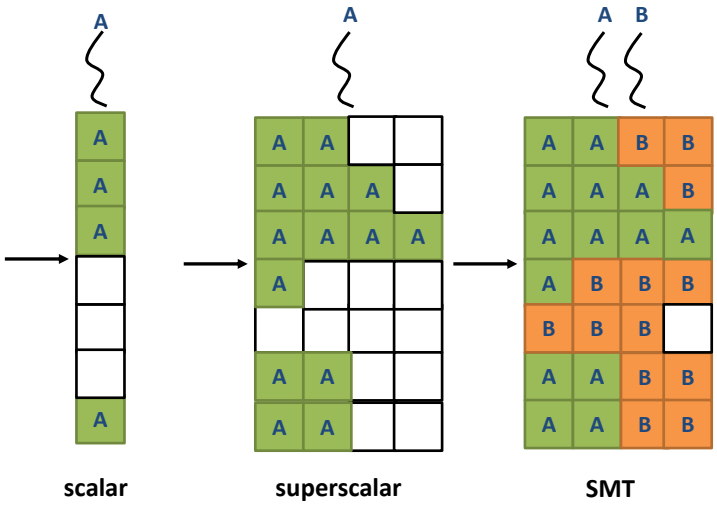


Figure 2.3: Multithreaded processor execution core: scalar, superscalar, SMT. A scalar execution core processes one thread and issues 1 instruction per cycle. On the other hand, a superscalar execution core processes one thread but issues n instructions per cycle depending on the number of functional units. The SMT is a superscalar execution core that processes m threads.

cycle. This type of architecture is called *simultaneous multithreading (SMT)* [146]. Most of the SMT architectures are found in the general-purpose domains such as IBM POWER5 [129], Intel Pentium4 HT [75], Intel Atom [136], and Sun Microsystems UltraSPARC T1 [80]. They tend to use large number of redundant functional units (around 8) and hardware threads, which increase the IPC rate but make the instruction issue/dispatcher unit and thread scheduler more complex [43]. For instance, the IBM POWER5 [129] is a dual core 2-way SMT, 8-way superscalar processor, with a die area of 389 mm^2 in 130 nm technology. It is typically used in server architectures. For embedded systems, there exist some more optimized SMT architectures such as simultaneous thin-thread [155] and Responsive multithreaded architecture [159]. They are 4-way superscalar processors with small dispatch queues (32).

SMT processors are not suitable for embedded systems domain for 3 main reasons: 1) It is impossible to determine the WCET since instructions are scheduled dynamically 2) Large die area because of multiple functional units and registers 3) High power consumption.

Thus, scalar processors are more attractive for embedded systems integration and will be adapted for our further analysis. In the next section, we will show what the multithreading techniques for explicit scalar processors are.

2.1.1.3 Instruction issue

Finally, two types of instruction issue exist (Figure 2.4): interleaved and blocked.

Interleaved multithreading (IMT), also called switch-on-cycle or fine-grain multithreading, is a multithreading technique that issues an instruction from a different thread at every clock cycle using a round-robin scheduler, with zero context-switching overhead. The first well-known architecture which uses IMT is the Denelcor HEP [76]. It supports up to 50 threads in hardware. Tera MTA [66] is a derivative of the HEP with similar properties that supports 128 TCs. These architectures

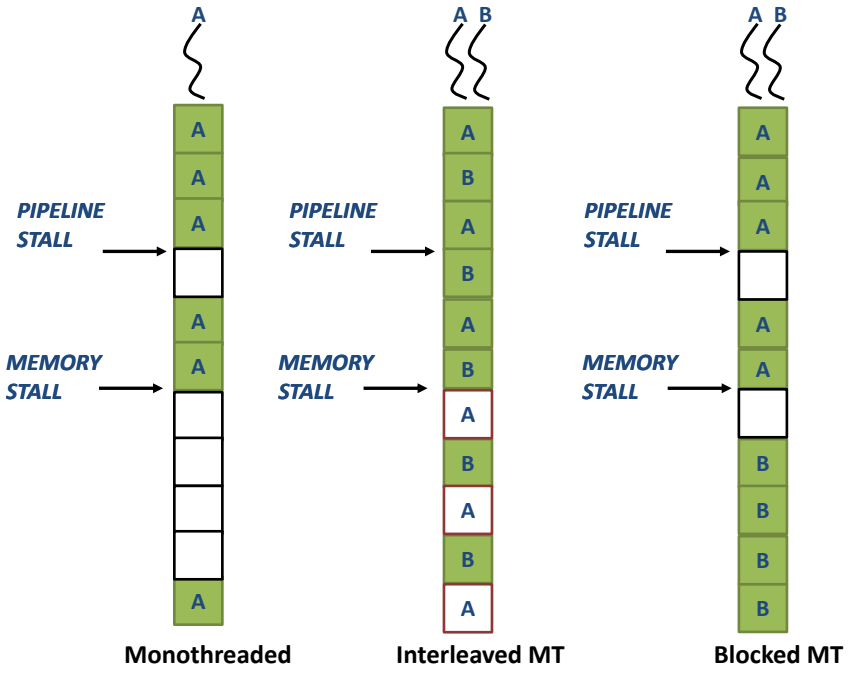


Figure 2.4: Multithreaded instruction issue types compared with monothreaded: interleaved multithreading (IMT), blocked multithreading (BMT). IMT issues an instruction from a different thread at every clock cycle with zero context-switching overhead. On the other hand, BMT allows a thread to run normally as in sequential mode before being switched out for long latency events. However, the context-switching has some penalty cycles.

do not use caches, and rely on having a large number of threads to hide the memory latency between successive instructions of a thread. At any point in time, each pipeline stage will contain an instruction from a different thread. Therefore, there is no need for a complex circuitry that handles pipeline interlocks (instruction, data, and control dependencies) since each thread can have just one instruction in the pipeline. Nevertheless, to support sufficient parallelism, the number of active threads should be equal or greater to the number of pipeline stages, thus more hardware resources. For instance, MIPS 34K [93], a recent IP for MPSoC integration, has a 9-stage pipeline and supports 9 TCs. SUN UltraSPARC T2 [127], a CMT processor used for server architectures, has a 6-stage pipeline for each core and supports 8 TCs. In the IMT, the performance of a single thread is degraded by 1/n, where n is the number of TCs. Thus, IMT architectures are useful for throughput oriented architectures. For example, in embedded systems, Eleven Engineering XInc [44, 79] and Ubicom MASI [2, 49] IMT processors are used in the wireless communication domain. Another researcher has developed an IMT MicroBlaze soft-IP for FPGAs [99].

On the other hand, blocked multithreading (BMT), also called switch-on-event or coarse-grain multithreading, allows a thread to run normally as in sequential mode before being switched out for long latency events such as cache misses, failed synchronization [4], or wait for producer data in a streaming execution model. These events normally represent points in execution at which the processor would become idle for a long period of time. In such a case, it is useful to perform a context switch and execute instructions from another thread to fill the otherwise idle cycles. This is only effective when the context switch time is significantly less than the idle period of the

event causing the switch [25]. The main advantage of BMT is that it requires a smaller number of TCs for multithreading to mask the long latency stalls, which means lower hardware cost than IMT. For instance, Infineon TriCore2 [68, 103] supports 2 TCs for a 6-stage pipeline, PRESTOR-1 [138] supports 4 TCs for a 10-stage pipeline, and MulTEP [157], which is based on Anaconda multithreaded processor [98], supports 2 TCs for a 5-stage pipeline. In addition, each thread can execute at full processor speed as in single-threaded mode. However, careful processor design choices must be taken to avoid starving other waiting thread contexts. For instance, if the BMT processor is well-dimensioned and the cache misses are almost negligible, this means there will be no context switches, hence other thread contexts will never execute and advance. Thus, for real-time embedded applications, TCs should have priorities to guarantee the response time. For instance, in TriCore 2, TC0 is the main thread and TC1 is a helper thread. Another drawback is the context switch penalty, which is dependent of the number of pipeline stages. In fact, for each thread context switch, the pipeline should be totally flushed and reset.

Examples of recent IMT and BMT processors that exist in embedded systems are shown in Figure 2.5.

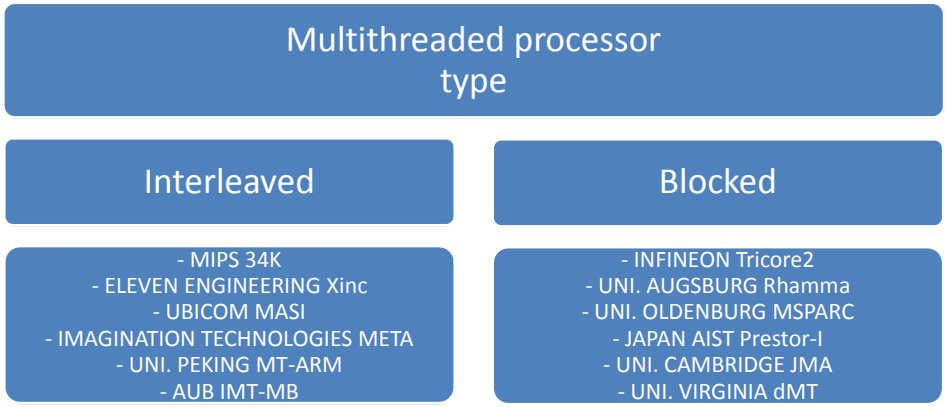


Figure 2.5: Example of industrial and research interleaved and blocked multithreading processors.

In the next section, we will present a cost-effectiveness model that will give us a relationship between the performance and the implementation cost of a multithreaded processor with respect to the number of TCs.

2.1.2 Cost-effectiveness model

A multithreaded processor is characterized by its number of TCs. The cost-effectiveness model is the relationship between the performance efficiency and the total implementation cost (transistor count, power, design complexity, etc...) each TC adds to the multithreaded processor. The cost-effectiveness model (CE) is proposed by Culler [37] and it is given by the following formula:

$$CE(n) = \frac{E(n)}{C(n)} \tag{2.1}$$

where E(n) is the processor efficiency distribution given by:

$$E(n) = 1 - \frac{1}{\sum_{i=0}^{n} (\frac{r(n)}{l(n)})^i \cdot \frac{n!}{(n-i)!}} \tag{2.2}$$

n is the degree of multithreading, r(n) is the mean service time distribution, and l(n) is the mean latency penalty distribution.

and C(n) is the total implementation cost given by:

$$C(n) = \frac{C_s + n.C_t + C_x}{C_b} \tag{2.3}$$

n is the degree of multithreading, C_s is the cost for a single threaded mechanism, C_t is the incremental cost per thread, C_x is the incremental cost of thread interactions, C_b is the base cost of an equivalent single thread processor.

The processor efficiency distribution E(n) of a multithreaded processor is proposed by Agarwal [4], and it is an extension of [67]. The analytical model relies more on a dynamic execution model (scheduling) of the threads, hence service/workload distribution information is injected in the model. The service time intervals between context switches are distributed geometrically. A latency penalty is distributed exponentially. The processor efficiency distribution is presented in equation 2.2.

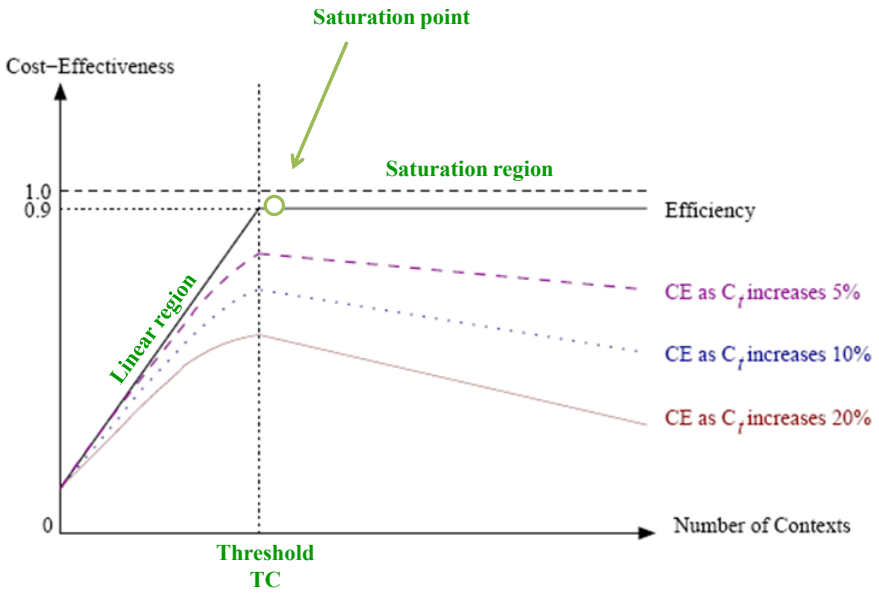


Figure 2.6: Cost-effectiveness of a multithreaded processor when varying C_t [4].

In Figure 2.6, we plot the theoretical processor efficiency model and the cost-effectiveness model versus the number of thread contexts n. There are two regions on this processor efficiency graph: a linear region on the left and a saturation region on the right. The saturation point is reached when the service time of the processor completely conceals the latency. However, for the same number of TCs n, the cost-effectiveness model shows that when the cost per thread C_t increases beyond

a certain threshold number of TCs, the cost-effectiveness decreases. Therefore, it is necessary to support the multithreaded processor with a maximum number of TCs not exceeding a certain threshold in order to obtain the peak cost-effectiveness result.

2.1.3 Synthesis

In this section, we investigated the different types of multithreaded processors that exist in the literature. Based on our classification, explicit and scalar multithreaded processors are retained due to their simplicity for embedded systems requirements. Then, we saw that there are 2 types of multithreading techniques that can be adapted for scalar multithreaded processors: IMT and BMT. Both techniques have their advantages and disadvantages, but it is not yet clear for us which one has the best transistor efficiency and which degree of multithreading is the best. Therefore, in the next section 2.2, we will develop an RTL model of a IMT and BMT core using a small 5-stage pipeline RISC. Based on the synthesis results, we will be able to choose the optimal number of thread contexts that fit a very small multithreaded core for embedded systems.

2.2 Implementation of a small footprint multithreaded processor for embedded systems

We have seen so far (section 2.1) that explicit and scalar multithreaded processors fit the embedded systems requirements. In addition, the degree of multithreading, or in other words the number of hardware TCs, should not exceed a certain threshold according to the cost-effectiveness model (section 2.1.2).

In this section, we will provide the answer on the degree of multithreading that is optimal for a small 5-stage pipeline RISC multithreaded core. First, we will present briefly a 5-stage RISC monothreaded core called AntX. Then, we will extend this core to support IMT and BMT. And finally, we will compare the transistor efficiency of the IMT and BMT cores using the synthesis results in the 40 nm TSMC technology and the performance results of a simple bubble-sort application.

2.2.1 Monothreaded AntX

AntX is a scalar, in-order, 5-stage pipeline (IF,ID,EX,MEM,WB), monothreaded RISC core (Figure 2.7), developed by the Embedded Computing Laboratory at CEA LIST. It is a 32-bit architecture designed specifically to be used as a low-cost control core in a MPSoC environment. Therefore, there are no complex units such as a branch predictor, FPUs, and multipliers. Its register file has 16 32-bit registers.

AntX has a GNU toolchain (antx-elf) that supports its ISA. The ISA has a variable instruction size (16/32 bit) in order to reduce the instruction memory footprint. So, some basic arithmetic/-logic/comparison/jump instructions are coded in 16-bit, while other more complex instructions are coded in 32-bit. The Instruction Fetch (IF) unit fetches a 32-bit instruction from the memory and handles the aligned/unaligned instructions in a finite state machine (FSM).

The instruction flow in the pipeline resembles that of the MIPS-I R3000 described in [62]. One exception is that the jump/branch instructions are executed in the EX-stage instead of the ID-stage. They use the ALU in order to calculate the new PC address, so the hardware cost of a

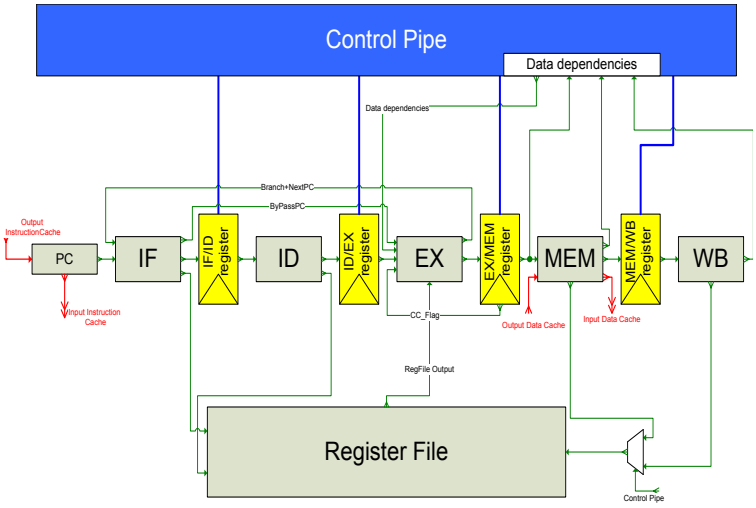


Figure 2.7: Monothreaded AntX.

dedicated adder is avoided. Another difference is that we disabled the delay slot instruction after a jump/branch instruction using the gcc compiler option '-fno-delayed'. The compiler inserts always a 'nop' instruction.

The *control pipe* unit is responsible for handling the data dependencies between the instructions in the different pipeline stages. For example, when an instruction in the ID-stage wants to read a data from a specific register, and that data is already calculated but not yet committed by the WB-stage, the *control pipe* will stall the pipeline until the data has been committed. To solve this problem, 'data forwarding' techniques between the pipeline stages are supported by AntX. Data forwarding eliminates most of the pipeline hazards (WAR, WAW, WAW). However, 1-cycle pipeline stall latency can still occur due to 2 reasons: branch instructions penalty (if taken-branch) and pipeline interlocks due to load/store instructions in the MEM-stage. The latter is due to memory access latency during a L1 cache hit when load/store instructions are in the MEM-stage. On the other hand, if the data is not present in the L1 cache (cache miss), then the waiting time is more than 1 clock cycle. In fact, those pipeline stalls will degrade the processor performance below the optimal IPC of 1.

Monothreaded AntX has been synthesized in 40 nm TSMC technology (low power, low threshold voltage, worst case) with a frequency of 300MHz. We used Design Compiler tool from Synopsys. The surface repartition of each module is shown in Figure 2.8. The overall core area is 11417 μm^2, which is about 8.05 kilogates.

One clear observation is that the register file occupies a significant portion of the low-cost monothreaded core, which is 38% of the total core area. In multithreaded processors, each TC has its own register file. Therefore, for a multithreaded AntX with 4 TCs, the new core area increase will be more than 100%. This implies there is a diminishing return advantage of implementing an embedded multithreaded processor with more than 2 TCs. This conclusion is also backed up by the design choice of MIPS 1004K [94], which is a multiple multithreaded core and is synthesized for only 2 TCs per core. Thus, for the rest of our work, we chose multithreaded processors with 2

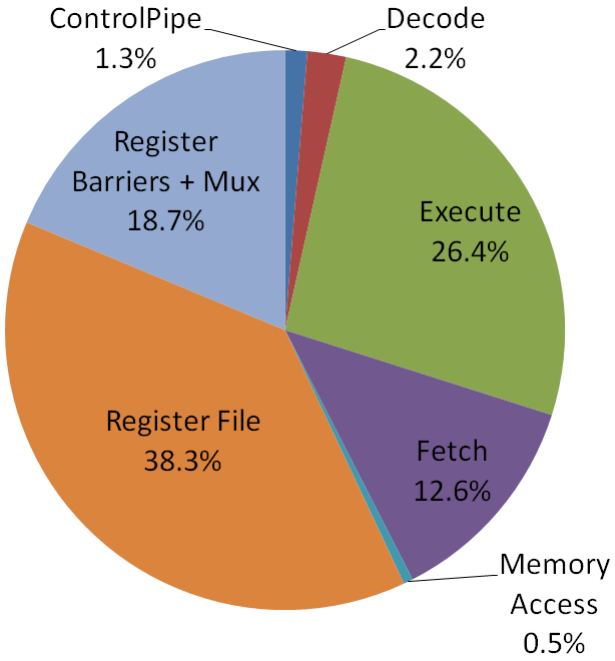


Figure 2.8: Surface repartition of different components in monothreaded AntX for 40 nm TSMC technology. Total area = 11417 μm^2, Total number of gates = 8.05 kilogates.

TCs. In the next section, we will explore in more details the design choices for IMT and BMT.

2.2.2 Interleaved MT AntX

IMT is a multithreading technique that issues an instruction from a different thread at every clock cycle using a round-robin scheduler, with zero context-switching overhead. When one TC is blocked, the IMT tries to process instructions from the active TC at half the speed (see Figure 2.4). In this section, we will modify the monothreaded AntX RTL model described in section 2.2.1 in order to support interleaved multithreading with 2 hardware TCs [14]. In fact, for a 5-stage pipeline, 2 TCs are sufficient to eliminate the stall conditions and data dependencies. AntX IMT with 2 TCs (TC1 and TC2) is shown in Figure 2.9.

The following are the main modifications for extending the monothreaded AntX to IMT:

- *Duplicating the register file and PC:* each TC should have its own register file and PC in order to store and switch the context in zero time overhead.

- *Duplicating control pipe:* the control pipe module is used to manage the instruction flow and dependencies of a TC at each pipeline stage. Therefore, it controls the pipeline and validity of each stage. In IMT, two successive pipeline stages have instructions from different TC. Therefore, to support 2 TCs, either we have to modify the original control pipe (monothreaded version) or duplicate it. According to the synthesis results of the monothreaded AntX (Figure 2.8), the surface occupation of the control pipe is only 1%. Accordingly, from development and validation time perspectives, we duplicate the control pipe. In addition, a multiplexer is

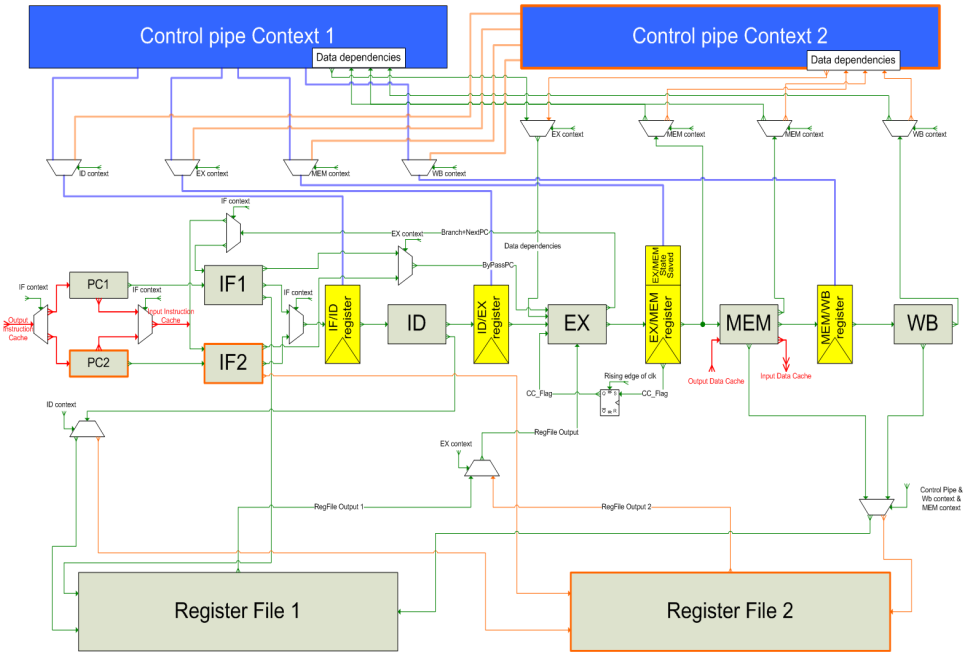


Figure 2.9: Interleaved multithreaded AntX [14].

added for each I/O signal belonging to the control pipe. This multiplexer switches between the 2 control pipes depending on the actual TC identifier in the pipeline stage.

- *Duplicating IF module:* to manage two different TCs, the IF module can be modified or duplicated. The first one involves modifying the fetch state machine and saving each TC state at each context switch, which incorporates more development and validation time. The second one is easier to implement, since the IF module is already validated. Furthermore, in terms of surface cost, the two solutions would be equivalent. Therefore, the second solution has been preferred. However, a small modification is required for each IF module to handle properly the instruction fetching: the state of the FSM should be delayed. This implies that the FSM depends on two rising edge clock cycles instead of one, since each TC is processed at half the speed.

- *Augmenting the EX/MEM inter-stage register size*: when a data cache miss occurs in the MEM stage for TC1, the pipeline is normally stalled waiting for the data, while TC2 instructions could have proceed their execution. Therefore, the EX/MEM register has been increased to save the EX/MEM state that corresponds to TC1 in order to be reloaded when TC1's data arrives. If the state is not saved, the MEM module would have the output from a wrong instruction, and the instruction that caused the data miss would be lost.

- *Delaying signals:* some signals have been delayed so they correspond to the right TC. For instance, the bypass PC signal from IF-stage to EX-stage and the execution flag signal from EX-stage are delayed by 1 cycle. Otherwise, the instruction execution flow would be incorrect.

In the next section, we will design the BMT AntX.

2.2.3 Blocked MT AntX

BMT is a multithreading technique that allows a thread to run normally as in sequential mode before being switched out for long latency events such as cache misses. In this section, we will modify the monothreaded AntX RTL model described in section 2.2.1 in order to support blocked multithreading with 2 hardware TCs. AntX BMT with 2 TCs (TC1 and TC2) is shown in Figure 2.10.

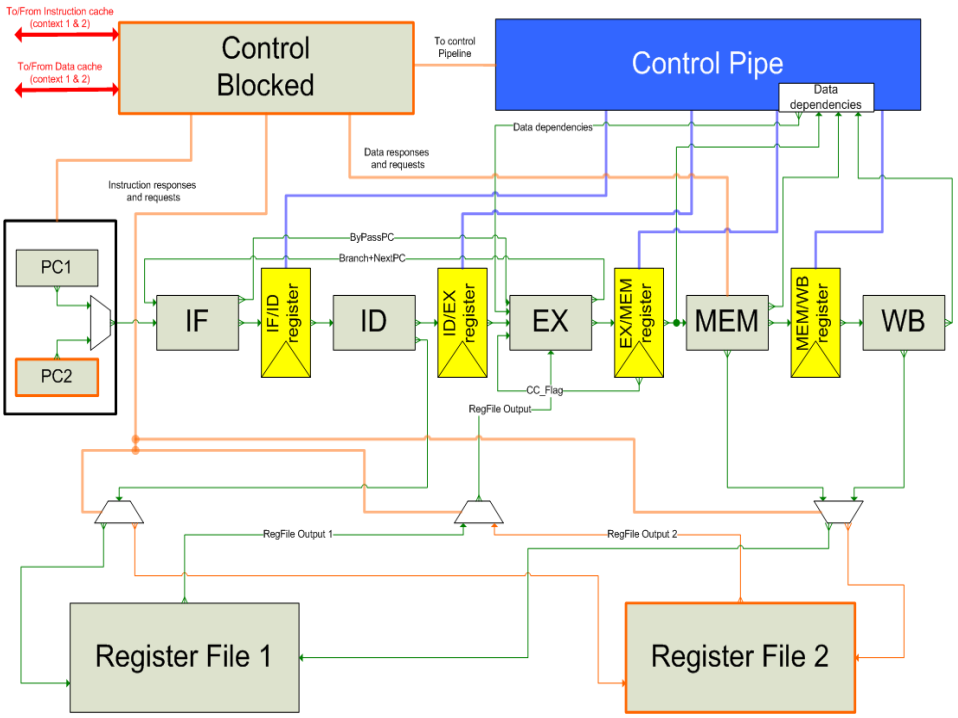


Figure 2.10: Blocked multithreaded AntX.

As we can notice, the BMT AntX resembles a lot to the monothreaded AntX. This is because one TC is processed in the pipeline at a time. Therefore, there is no need for duplicating the IF module and the control pipe, or adding extra registers and multiplexers. In fact, the main modifications are related to handling and managing I/O signals coming from external modules such as L1 caches. This is because the functionality of the BMT is dependent on these signals. The following are the main modifications for extending the monothreaded AntX to BMT:

- *Duplicating the RF and PC registers:* similarly to the IMT, each TC should have its own register file and PC in order to store and switch the context in zero time overhead. In reality, the context switch will take one cycle as we will explain in the next point.
- *Adding a 'control blocked' module*: the 'control blocked' module is the essential part of the BMT core. A cache memory access occurs at the IF-stage (I$) and MEM-stage (D$). Each request status is either a cache hit or miss that is read back by the 'control blocked' module. Internally, the 'control blocked' implements a Moore FSM that is triggered by the cache

memory request status as it is shown in Figure 2.11. Initially, and after the reset signal is low, it executes TC1 as long as there are no cache misses. When TC1 generates a cache miss, it goes to the *context switch* state that stores the appropriate PC value and re-initializes the internal register and FSM states of each pipeline stage to start TC2 processing. Then, TC2 executes as long as it hits in the cache. The BMT model implements a 'greedy' protocol, which means that TC1 has higher priority on TC2. Therefore, if the data of TC1 is returned from upper-level memories and TC2 is still executing, the latter is switched and TC1 resumes. On the other hand, if TC2 misses in the cache while TC1 has not yet its data, then the pipeline stalls and waits for one of TC's data to be returned. Finally, the 'blocked control' sends the appropriate instruction/data responses for the right context to the IF-stage, MEM-stage and RF.

- *Synchronization between 'control blocked' and 'control pipe' modules*: this is essential for proper communication between these 2 modules, especially during a context switch. In fact, the 'control blocked' module should inform the 'control pipe' module of the currently executing status of the TC in order to send the appropriate validation signals to each pipeline stage.

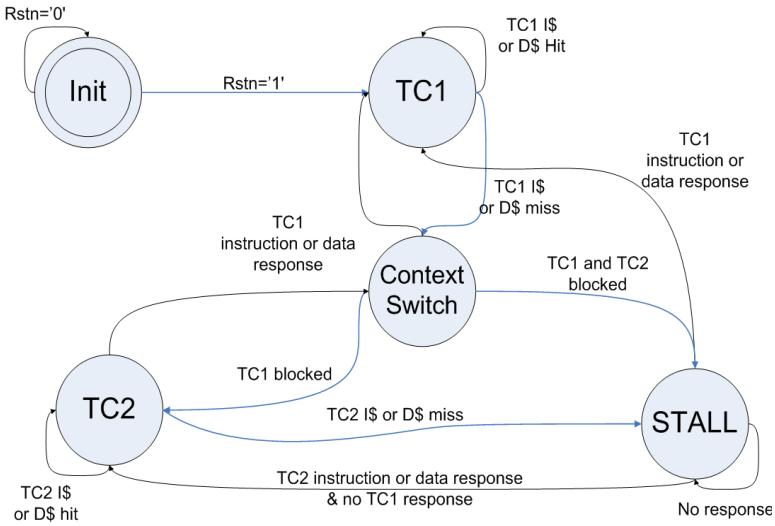


Figure 2.11: Blocked multithreaded AntX FSM for 2 thread contexts: TC1 and TC2. The FSM shows that the BMT processor has 4 execution states: executing TC1, executing TC2, context switching, and stall.

In the BMT FSM in Figure 2.11, the context switch is actually one FSM state. Therefore, the context switch overhead in BMT costs one clock cycle. But the penalty due to context switching differs if this is due to I$ miss or D$ miss. For an I$ miss, we insert a bubble in IF stage that causes another one clock cycle of penalty. On the other hand, for a D$ miss at the MEM stage of the pipeline, the already fetched instructions in the pipeline have to be invalidated before fetching instruction from the other TC. Thus, context switching penalty causes 5 cycles (1 (CS) + 4).

In the next section, we will evaluate the performance and area of the IMT/BMT AntX using 2 hardware TCs in order to understand the characteristics of each multithreaded core type and conclude which one has the best transistor efficiency.

2.3 Performance evaluation

In this section, we evaluate the transistor efficiency of the monothreaded and multithreaded processors developed in the previous section. First, we provide synthesis results of each processor type, and a comparison between the surfaces. Then, we analyze the performance of each multithreaded processor by varying several parameters such as data cache size and L2 data memory latency. These parameters will inform us under which conditions a specific multithreaded processor is an interesting solution. Finally, given the synthesis and performance results, we compare the transistor efficiency of each processor type and conclude.

2.3.1 Monothreaded v/s Multithreaded processors: area occupation

To evaluate the surface of each processor type, we use Design Compiler from Synopsys for ASIC synthesis. The IMT and BMT AntX RTL models have been developed in VHDL and synthesized in 40 nm TSMC technology (low power, low threshold voltage, worst case) with a frequency of 300MHz, similar to the monothreaded AntX. The surface repartition of IMT and BMT processor is shown in Figure 2.12(a) and 2.12(b) respectively.

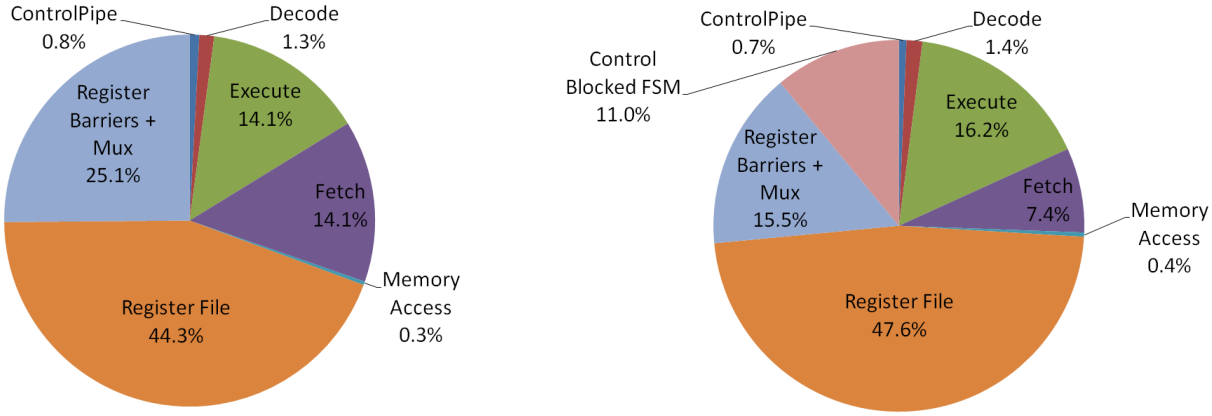


Figure 2.12: Surface repartition of different components synthesized in 40 nm TSMC technology for a) IMT AntX: Total area = 19772 μm^2, Total number of gates = 13.95 kilogates b) BMT AntX: Total area = 18418 μm^2, Total number of gates = 12.99 kilogates.

IMT AntX has an overall core area of 19772 μm^2 equivalent to 13.95 kilo gates. The IMT AntX has an augmentation of 73.4% in core area compared to the monothreaded AntX. This is mainly due to doubling the RF, PC, and IF modules, which is essential for proper IMT functioning. In addition, about 20 multiplexers (64 bits to 32 bits) have been added for IMT.

As for BMT, the overall core area is 18418 μm^2 equivalent to 12.99 kilo gates. BMT AntX has an augmentation of 61.3% in core area compared to the monothreaded AntX. This implies less surface occupation than IMT AntX.

On the other hand, by considering the area overhead for a complete processor system with L1 I$ and D$ memories, then the area overhead of a multithreaded processor with respect to a

monothreaded processor is reduced. The area of a processor system in μm^2 is given by equation 2.4:

$$surface(PE\_system) = surface(mono|MT) + surface(L1\_I\$) + surface(L1\_D\$) \quad (2.4)$$

The L1 cache memories area are estimated using CACTI 6.5 tool [100] from HP in 40 nm technology. The technology used by CACTI tool is based on ITRS roadmap [125], but it is not similar to TSMC technology. Therefore, the processor system is not synthesized with the same technology, but this gives us an idea of the relation between cache size and processor size. We estimate a direct-mapped cache memory ranging from 512-B to 4-KB. The estimated cache memory area and the corresponding area overhead of each MT system with respect to the same monothreaded system configuration are shown in Figure 2.13:

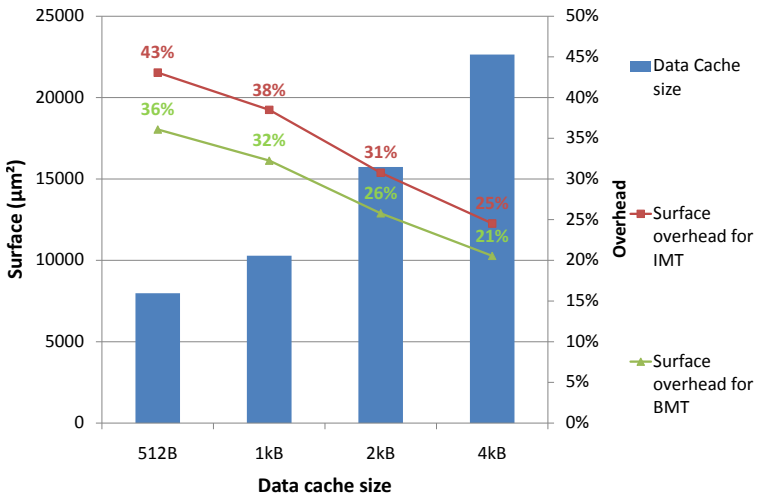


Figure 2.13: MT processor area overhead with respect to the monothreaded processor. Each MT system has one L1 I$ and one D$ memory. We show 4 MT systems with different L1 cache sizes. The L1 cache areas are estimated using CACTI 6.5 tool in 40 nm technology and the processors are synthesized in 40 nm TSMC technology.

We can notice that a 4-KB direct mapped cache memory has a bigger size than all the multithreaded processors. This shows how small the processors' size we are using. For instance, for the IMT processor, its overhead ranges from 43% to 25% depending on the size of the cache memories. It is also clear that the overhead of the BMT system is smaller than the IMT system.

As a conclusion, BMT AntX processor has a less core area overhead than IMT AntX processor according to our synthesis results. The main reason is that the surface of the FSM blocked is smaller than the surface of all the multiplexers added for IMT. In addition, the instruction fetch module is doubled in IMT. In the next section, we will see the performances of each processor type in order to conclude on the transistor efficiency of the IMT and BMT processors.

2.3.2 Monothreaded v/s Multithreaded processors: performance and transistor efficiency

In this section, we use a typical processor system environment described in Figure 2.14. The processor-memory architecture is based on a Harvard architecture with separate L1 instruction cache (I$) and data cache (D$) busses. It implements a 2-level memory hierarchy with L1 I$ and D$ memories, connected with an AHB bus to an on-chip L2 instruction and data memories. The L2 memories contain all the instruction and data codes of the applications.

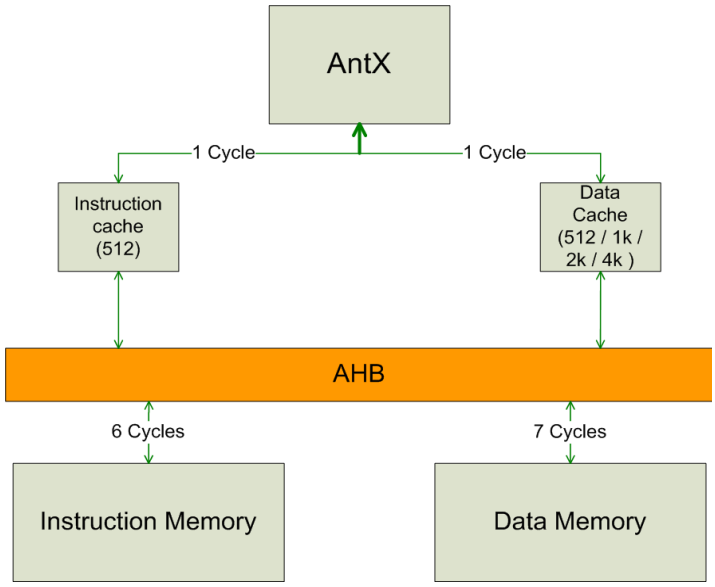


Figure 2.14: AntX hierarchical memory system: Only the data cache size parameter is varied from 512-B to 4-KB. The average memory access latency for the instruction and data memories are observed during application execution and depends on the AHB arbiter. If AntX is multithreaded, then the L1 I$ and D$ memories are segmented per TC, which means each TC has half the L1$ size compared to the monothreaded AntX.

The processor can be either monothreaded or IMT/BMT AntX with 2 TCs. For the IMT/BMT AntX, the L1$ memory is segmented per TC in order to limit cache interferences. Therefore, each TC has half the L1$ size compared to the monothreaded AntX. For this experiment, we consider a basic bubble-sort application for 600 elements. The application has lot of jump/branch instructions and data dependencies between instructions. We run 2 instances of the application with different elements sequentially on the monothreaded processor, and concurrently on the IMT/BMT processor. In this experiment, we vary 2 platform parameters for a better architecture exploration. First, the processor type can be chosen to be monothreaded, IMT or BMT. Second, the L1 D$ memory size can be set to 512-B, 1-KB, 2-KB, and 4-KB. This will generate different data cache miss rates as shown in Figure 2.15. The L1 I$ size is fixed to 512 Byte, which is sufficient for the bubble-sort application and generates only 0.07% of L1 I$ miss. A L1 cache hit takes 1 clock cycle, an access to L2 instruction memory due to L1 I$ miss takes 6 cycles, and an access to L2 data memory due to L1 D$ miss takes 7 cycles on average. L2 memory access time might vary few cycles (1-2 cycles) depending on the AHB arbiter.

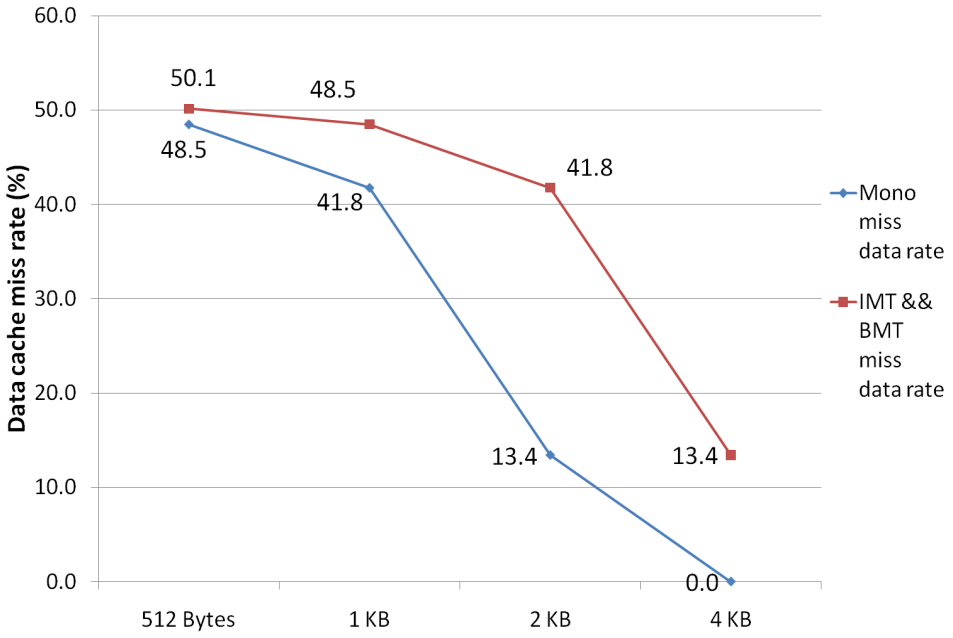
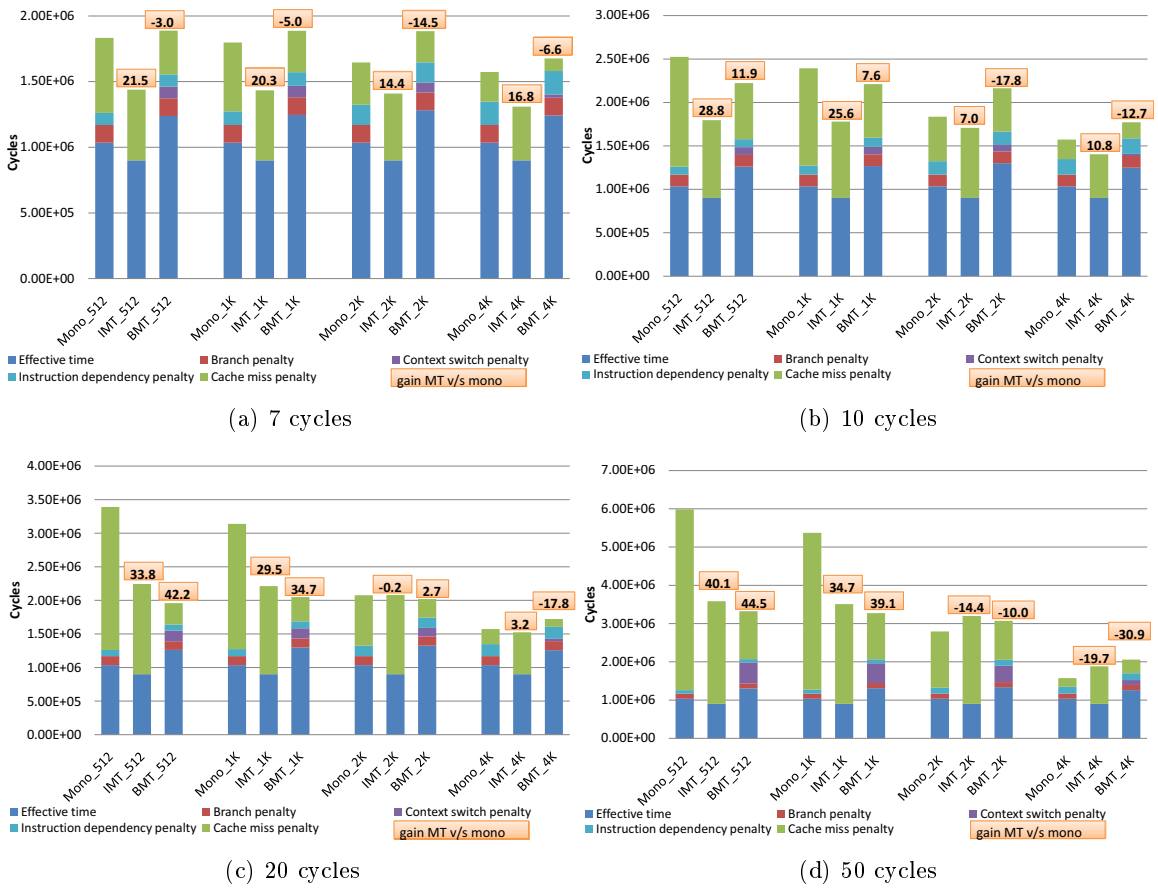


Figure 2.15: Data cache miss rates for monothreaded and IMT/BMT AntX while varying the cache size: 512-B, 1-KB, 2-KB, and 4-KB.

In this study, two platform parameters are varied: the data cache size (512-B, 1-KB, 2-KB, 4-KB) and the L2 data memory latency (7, 10, 20, 50 cycles). The first parameter has an impact on the data cache miss rate, which increases the access to the L2 data memory. The access to the L2 data memory is affected by the second parameter during a 'load' instruction. For instance, as a rule of thumb, let us assume that an application contains 30% of load/store instructions that are equally divided; this implies that approximately 15% of the instruction codes are affected by the L2 data memory latency. By varying this parameter, we are modeling different memory technologies. Both parameters are important for exploring the importance of the multithreaded processor with respect to the monothreaded processor.

In Figure 2.16, we show the execution time in cycles for all the cache sizes and data memory latency. We decompose the total execution time into 4 components: effective execution time, branch instruction penalty time due to 'taken' branches, data dependencies stall time due to pipeline interlocks, and memory stalls time due to cache misses.

For a small memory data latency of 7 cycles (Figure 2.16(a)), the IMT processor overcomes the performance of the monothreaded processor for all the cache configurations. The performance gain varies between 14.4% and 21.5%. In fact, the performance gain highly depends on the percentage of data cache misses that each segmented cache generates. Each TC in IMT processor has half the cache size, hence it generates more cache misses and more pipeline stalls due to L2 memory access. Due to its interleaving property, the IMT tolerates the pipeline stalls generated by branch penalties and data dependencies between instructions. Their stall times are hidden completely by executing instructions from another TC, if it is active. It is clear that 2 TCs are sufficient to hide all these latencies for a 5-stage pipeline processor. However, for BMT AntX, there is no gain at all.



(a) 7 cycles

(b) 10 cycles

(c) 20 cycles

(d) 50 cycles

Figure 2.16: Performance results in cycles of monothreaded v/s IMT/BMT AntX with a variable L2 data memory latency. The L1 D$ size varies from 512-B to 4-KB. L1 cache memories are segmented per TC for the IMT/BMT.

In fact, the memory stall latency is not high enough to compensate the context switching penalty, which is equal to 5 cycles for a D$ miss. Furthermore, BMT does not mask the stall latencies due to instruction dependencies and branch instructions. All these conditions make the BMT processor not an interesting solution for small memory access latencies.

However, when increasing the latency of the L2 data memory, more pipeline stalls due to data cache misses are generated. Thus, BMT processor is more performant under such conditions. For instance, for 20 cycles (Figure 2.16(c)), the BMT has a gain of 42.2% and 34.7% for 512-B and 1-KB L1 D$ memories, which overcomes the performance of the IMT for the same cache sizes. The same observations appear for 50 cycles of latency in Figure 2.16(d). In fact, the BMT processor has enough stall latencies to mask and the penalties due to context switching are minimal. But, when the sizes of the D$ memory increases, it generates less cache misses for the monothreaded processor. It reaches almost 0% for a 4-KB L1 D$. On the other hand, the cache misses 13.4% for the multithreaded processor because of its segmented cache. Therefore, for all memory latencies and a big D$ size, we see little gain for IMT because it is still able to mask the other types of stalls, and no gain at all for BMT. Thus, any type of multithreaded processor is not recommended when

the cache misses are not high enough to generate enough memory stalls latencies. In fact, 2 TCs are not enough for hiding this high stall latency.

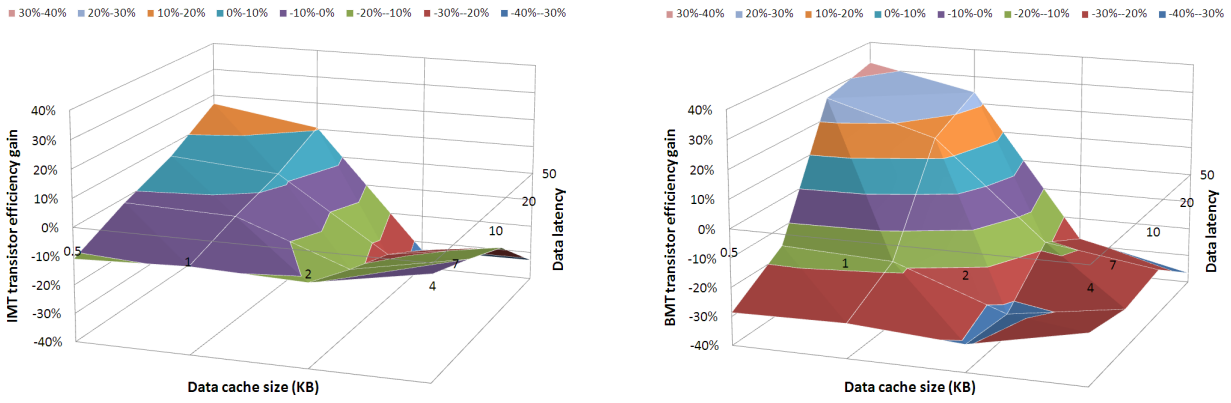
Finally, we compare the transistor efficiency of the IMT/BMT processor with respect to the monothreaded processor. The processor-system area is the sum of the processor area and its L1 cache memories given in equation 2.5:

$$TransistorEfficiency = \frac{IPC}{surface(core) \ + \ surface(I\$) \ + \ surface(D\$) \ [\mu m^2]} \tag{2.5}$$

We are mainly interested by the transistor efficiency gain of the multithreaded processor with respect to the monothreaded, which is given in equation 2.6:

$$Transistor\ Efficiency\ Gain(MT) = \frac{Transistor\ Efficiency(MT)}{Transistor\ Efficiency(Mono)} - 1 \tag{2.6}$$

Figures 2.17(a) and 2.17(b) show the transistor efficiency gain of the IMT and BMT processor respectively.



(a) IMT (b) BMT

Figure 2.17: Transistor efficiency gain of MT AntX processor with respect to monothreaded AntX processor. For the x-axis, we vary the L1 D$ size and for the y-axis we vary the data memory latency in cycles.

The transistor efficiency results show that the monothreaded processor is more efficient than any multithreaded solution when there is a small memory access latency and not enough data cache misses (i.e. big cache size) that generate pipeline stalls. In the other cases, the BMT is more transistor efficient than the IMT processor, and can reach an efficiency gain of 33% for a small cache size and high memory latencies.

2.3.3 Synthesis

In this chapter, we designed, based on a monothreaded AntX processor, two small footprint, scalar, in-order multithreaded processors for the embedded systems: Interleaved Multithreading (IMT) and Blocked Multithreading (BMT). The synthesis results in 40 nm TSMC technology showed

that the register file occupies more than 38% of the overall core area, thus it is not area efficient to integrate more than 2 thread contexts (TC) per multithreaded processor. Therefore, we have chosen to implement a multithreaded processor with 2 TCs.

Both multithreaded processors were synthesized in 40 nm TSMC technology. The results shows that the IMT and BMT processors have 73.4% and 61.3% increase in core area than the monothreaded core. Thus, the BMT has a smaller area.

Finally, we compared the performances and transistor efficiency of both MT cores using a bubble sort application, while varying the L1 data cache size and the data memory latency. The results show that there is no definitive conclusion on which type of processors is the best. In fact, there is a trade-off between the data cache memory size, the data memory latency, and the core area overhead. Choosing the best processor highly depends on the system designer specifications and the application requirements. For instance, if peak performance is the main design parameter, then the multithreaded processors offer a good increase in performance for most of the memory configurations. However, if transistor efficiency is a design constraint, then the results highly depend on the architecture parameters (processor, memory, caches, etc.). For instance, for small cache sizes that generate lot of memory accesses and for high memory latencies, the BMT processor is more performant and transistor efficient. We should note that in this experiment, we did not vary the L1 I$ size, hence there were no processor stalls due to instruction cache misses.

It is worth to note that our experiments are based on a very small-footprint processor core, which is almost the extreme case in processor design. However, if the initial processor core has more hardware blocks, hence a bigger area, then our conclusion regarding transistor efficiency might change. For instance, for 40 nm technology, the ARM Cortex A5 and MIPS 24 KE have a core area of more than 250 μm^2 and 350 μm^2 respectively. These are more than 35 times larger than AntX monothreaded! In addition, by having a deeper pipeline and branch prediction units, pipeline stalls are more severe. For instance, in MIPS 24 KE, a branch misprediction costs 5 cycles. These new types of stalls are advantageous for the multithreaded processor. Another note to take into consideration is the application type. In our experiment, we used a very simple application that runs on the processors as standalone until completion. However, in more complex SoCs, the processors might be doing different types of processing that induces new types of stalls such as task synchronization, task allocation/deallocation, memory allocation, and others, which are not directly related to the actual application execution but are necessary for proper SoC functioning. In addition, the multithreaded application can have different types of threads: computation-intensive and I/O intensive. The latter is completely masked if it is scheduled on the same multithreaded processor with a computation-intensive task. For instance, the multithreaded MIPS 34K with 2 TCs is able to achieve 200% in audio throughput applications [143] compared to the monothreaded MIPS 24KE for only 28% core area increase.

Based on this conclusion, we will explore in the next chapter the performance impact of the multithreaded processor by running more relevant benchmarks in an asymmetric MPSoC architecture: the SCMP architecture.

Chapter 3

Multithreaded processors in asymmetric homogeneous MPSoC architectures

You are not what you think you are; but what you think, you are. – Norman Vincent Peale, minister and author

Contents

Asymmetric homogeneous MPSoCs are an interesting solution for massively-parallel dynamic embedded applications due to their high reactivity and load-balancing between the homogenous cores. The separation between the control and computing cores makes the asymmetric architecture highly transistor and energy efficient. In order to tackle the requirements of future massively-parallel dynamic applications, the asymmetric homogenenous MPSoC should reach the manycore level. However, when integrating several cores on-chip, the architecture suffers from limited bandwidth [12] due to the limitation of the chip's package I/O pins. This implies that the more traffic will be exercised off-chip, the more the cores will be stalled on-chip, hence lower aggregate IPC. Thus, it will be advantageous to explore the benefits of hardware multithreading for future manycore chips, in order to keep the core as busy as possible and increase the aggregate IPC.

In chapter 2, we designed, based on a monothreaded AntX processor, two small footprint, scalar, in-order multithreaded processors for the embedded systems: Interleaved Multithreading

(IMT) and Blocked Multithreading (BMT). We have shown that there is no definitive conclusion on which type of processors is the best, and that it all depends on the system designer specifications and the application requirements. In this chapter, we use the SCMP architecture, which is an asymmetric homogenenous MPSoC, to explore the advantages/disadvantages of hardware multithreading. First of all, we present the simulation framework, called SESAM, where the SCMP architecture is modeled. Then, we extend SESAM to support multithreaded processors. In particular, we have developed a new cycle-accurate multithreaded Instruction Set Simulator (ISS) in SystemC to model the IMT processor with 2 TCs. After replacing the monothreaded processor by an IMT/BMT processor with 2 TCs, we conduct several benchmarks in order to measure the efficiency of the SCMP architecture using multithreaded processors (MT_SCMP).

3.1 MPSoC Simulation environment

Designing an MPSoC architecture requires the evaluation of many different features (effective performance, used bandwidth, system overheads...), and the architect needs to explore different solutions in order to find the best trade-off. In addition, he needs to validate specific synthesized components to tackle technological barriers. For these reasons, the whole burden lies on the MPSoC simulators, which should be parameterizable, fast and accurate, easily modifiable, support wide ranges of application specific IPs and easily integrate new ones. Simulating a whole MPSoC platform needs to find an adequate trade-off between simulation speed and timing accuracy. The Transactional Level Modeling (TLM) [51, 32] approach coupled with timed communications, is a solution that allows the exploration of MPSoCs that reflects the accurate final design [57]. Time information is necessary to evaluate performances and to study communication needs and bottlenecks.

MPSoCs' architectures can have homogeneous or heterogeneous processors, depending on the application requirements. Choosing the best processor among hundreds of available architectures, or even designing a new processor, requires the evaluation of many different features (pipeline structure, ISA description, register files, processor size...), and the architect needs to explore different solutions in order to find the best trade-off. The processor Instruction Set Simulator (ISS), which role is very important, must have the following features: it should be parameterizable, fast and accurate, and be able to be integrated easily in the MPSoC simulation environment. The ISS mimics the behavior of a processor by executing the instructions of the target processor while running on a host computer. Depending on the abstraction level, it can be modeled at the functional or cycle-accurate level.

Lot of works have been published before on single-processor, multiprocessor and full-system simulators. In [160], the authors illustrate a wide range of simulators, mainly targeting general-purpose computing. In a more recent work [35], the authors presented an interesting classification of MPSoC simulators. For our knowledge, there is no published work on a simulator that supports asymmetric MPSoC architectures and allows their exploration. In this context, we used SESAM simulation environment [152, 153], which supports asymmetrical MPSoC architectures. SESAM is developed and proprietary to CEA LIST.

In this section, we present in details the SESAM simulation environment for asymmetric MPSoC architectures, and we show how SCMP is modeled in this framework. Then, we show the

different modifications done to SESAM to support multithreaded processors, in particular the central scheduler and the processor system. Finally, we realize the need for a multithreaded ISS that will be developed in section 3.2.

3.1.1 SESAM: A Simulation Environment for Scalable Asymmetric Multiprocessing

SESAM is a tool that has been specifically built to ease up the design and the exploration of asymmetric MPSoC architectures, which includes a centralized controller core that manages the tasks for different types of computing resources. The heterogeneity can be used to accelerate specific processing, but the task migration is not supported. The best trade-off between the homogeneity, which provides the flexibility to execute dynamic applications, and the heterogeneity, which can speed-up the execution, can be defined in SESAM. Moreover, this tool enables the design of MPSoCs based on different execution models (control-flow + streaming), which can be mixed, to find the best suitable architecture according to the application. It can be used to analyze and optimize the application parallelism, as well as control management policies. In addition, SESAM can support simultaneous multiple different applications and mix different abstraction levels, and can take part in a complete MPSoC design flow.

3.1.1.1 Framework

The SESAM framework is described with the SystemC description language [105, 54], and allows the MPSoC exploration at the TLM level with fast and cycle-accurate simulations. It supports co-simulation within the ModelSim environment [88] and takes part in the MPSoC design flow, since all the components are described at different hardware abstraction levels. Besides, SESAM uses approximate-timed TLM with explicit time to provide a fast and accurate simulation of highly complex architectures that can reach up to 4 MIPS. This model, described in [57], allows the exploration of MPSoCs while reflecting the accurate final design. A 90 % accuracy is pointed up compared to a fully cycle-accurate simulator. Time information is necessary to evaluate performances and to study communication needs and bottlenecks. Thus, all provided blocks of the simulator are timed and the communications use a timed transactional protocol.

To ease the exploration of MPSoCs, all the components and system parameters are set at runtime from a parameter file without platform recompilation. It is possible to define the memory map, the name of the applications that must be loaded, the number of processors and their type, the number of local memories and their size, the parameters of the instruction and data caches, memory latencies, network types and latencies, etc. More than 120 parameters can be modified. Moreover, each simulation brings more than 200 different platform statistics, that help the architect sizing the architecture. For example, SESAM collects the miss rate of the caches, the memory allocation history, the processor occupation rate, the number of preemptions, the time spent to load or save the task contexts, the effective used bandwidth of each network, etc. As depicted in Figure 3.1, a script can be used to automatically generate several simulations by varying different parameters in the parameter file. An Excel macro imports these statistics to study their impact on performances. Thus, the cache parameters, the network bandwidths, as well as the effective performance of the architecture, are ones among many features that can be evaluated to size and explore MPSoCs.

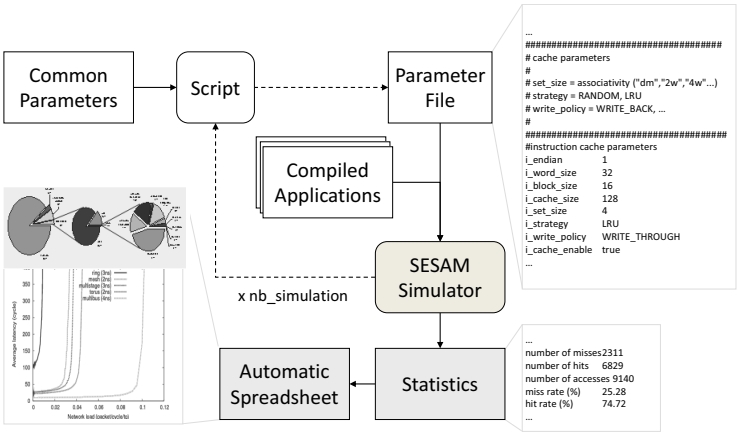


Figure 3.1: SESAM exploration tool and environment [152, 153].

Because the exploration of many parameters can take a lot of simulation time, SESAM offers the possibility to automatically dispatch all the simulations to different host PCs. Each available PC core defines an available slot, which can be used to execute one simulation. The tool is structured around a dispatcher and a NFS server. Thus, SESAM can take benefits of available PCs to automatically parallelize simulations and ease the exploration of architectures.

Debugging the architecture is possible with a specific GNU GDB [1] implementation. In the case of a dynamic task allocation modeling, it is not possible to know off-line where a task will be executed. Therefore, we built up a hierarchical GDB stub that is instantiated at the beginning of the simulation. A GDB instance, using the remote protocol, sends specific debug commands to dynamically carry out breakpoints, watchpoints, as well as step by step execution, on an MPSoC platform. This unique multiprocessor debugger allows the task debugging even with dynamic migration between the cores. Moreover, it is possible to simultaneously debug the platform and the code executed by the processing resources.

3.1.1.2 Infrastructure

As depicted in Figure 3.2, SESAM is structured as an asymmetrical MPSoC. It is based on a centralized Control Manager that manages the execution of tasks on processing elements. SESAM proposes the use of different components to design new MPSoCs. Other SystemC IPs can be designed and integrated into SESAM if they have a compatible TLM interface. The main elements are: the Memory Management Unit (MMU), the Code Loading Unit (CLU), Memories, a set of Instruction Set Simulators (ISS), a Direct Memory Access (DMA) unit, a Control Manager and Network-on-Chips (NoC).

The MMU is optional and can bring advanced capabilities to manage all the shared memory space, which is cut into pages. The whole page handler unit is physically distributed between the MMU and the local Translation Lookaside Buffers (TLB) for each processing core. All the memory functions are available through the SESAM HAL. It is possible to dynamically allocate or deallocate buffers. There is one allocated buffer per data block. An identifier is used for each data block to address them through the MMU, but it is still possible to use physical addresses. Different memory

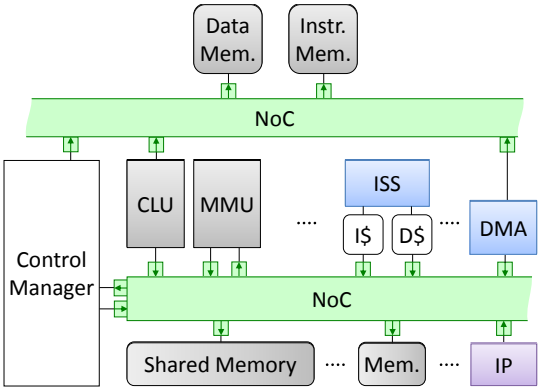


Figure 3.2: SESAM infrastructure [152, 153].

allocation strategies are available and can be implemented.

The CLU dynamically loads task codes from the external memory through a DMA access when it receives a configuration command from the Control Manager. Then, in a dynamic memory management context, it also has to update the MMU to provide the corresponding virtual to physical address translations. A context and a stack are automatically included for each task.

Different memory elements can be instantiated. The memory space can be implemented as different banks or a single memory. The former is logically private or shared, while the latter is only shared between the processors. Memory segments are protected and reserved for the Control Manager. Multiple readers are possible and all the requests are managed by the NoC.

The processors are designed with the ArchC ADL as processing resources with data and instruction cache memories, which are optional. The ArchC tool [114] generates functional or cycle-accurate monothreaded ISS in SystemC with a TLM interface [15]. A new processor is designed in approximately 2 man-weeks, but it depends on the instruction set complexity. Its simulation speed can reach tens of Millions of simulated Instructions Per Second (MIPS). Different models are available (MIPS, PowerPC, SPARC), as well as a complete MIPS32 processor (with a FPU) at the functional level. Preemption and migration of tasks are possible services that are available through an interruption mechanism. It allows to switch the context of the processing unit, to save it, and to restore the context code from the executed task memory space.

A DMA is necessary to transfer data between the external data memory and the internal memory space. A DMA is a standard processing resource and takes part in the heterogeneity of the architecture. It is a fully-programmable unit that executes a cross-compiled task for its architecture. A 3-dimensional DMA is available. Transfer parameters can afterwards be dynamically modified by other tasks, to specify source and target addresses defined at run-time. Finally, it dynamically allocates the required memory space for the transfer.

The Control Manager can be either a fully programmable ISS, a hardware component, or a mix of both. With the ISS, different algorithms can be implemented. Thanks to the SESAM HAL and an interrupt management unit, the tasks are dynamically or statically executed on heterogeneous computing resources. In addition, a multi-application execution is supported by this HAL. A set of scheduling and allocating services in hardware or software can be easily integrated, modified

and mixed. Besides, a complete hardware real-time operating system is available, named Operating System accelerator on Chip (OSoC). The OSoC supports dynamic and parallel migration, as well as preemption of tasks on multiple heterogeneous resources, under real-time and energy consumption constraints.

Many NoC topologies are supported by SESAM: a multibus, a mesh, a torus, a multistage and a ring network. These networks are detailed in [57]. All are modeled in approximate-timed TLM. Data exchanges are non-blocking and deterministic, regardless of the network load or the execution constraints. The multibus can connect all masters to all slaves, but does not allow master to master communications. In the mesh or the torus network, one master and several slaves are linked with a router. An XY routing and a wormhole technique are implemented. The multistage is an indirect fully connected network. It is divided into different stages composed of 4 input-output routers, and linked with a butterfly topology. All masters are in one side and all slaves are on the other side. It uses also a wormhole technique to transfer packets. Finally, in a ring network, a message has to cross each router when it goes through a ring. A parameter can change the number of rings. But, each master can connect itself to only one ring. A ring is bi-directional. Besides, we use a fifo with each memory to store memory accesses from computing resources. In order to accept simultaneous requests, two arbiters can be used: a FIFO or a fair round-robin policy. All communications are done at the transactional level and we can accurately estimate the time spent in every communication.

3.1.1.3 SCMP modeling

To demonstrate the SESAM's capabilities to model new asymmetric MPSoCs, we have used this framework to carry out the SCMP architecture, which is described in section 1.3. Platform parameters, such as latencies and constraints, are characterized by the Synopsys Design Compiler tool. As depicted in Figure 3.3, the architecture has three internal NoCs. The system NoC interconnects the external CPU, the external memories and the TLB dedicated to the application, with the core of the architecture. The CPU represents a host interface that allows the user to send on-line new commands to SCMP. For instance, it is possible to ask for the execution of new applications. The *TLB Appli* is used to store all the pointers of each task for each application in the external instruction memory. When the simulator starts, it automatically loads all the selected applications into this memory and update the *TLB Appli*.

The control NoC is used to connect the CCP (Central Controller Processor), which is the Control Manager, and all the processors resources through a control interface. In addition, processing resources can communicate with each other, and with the Memory Configuration and Management Unit. The MCMU aggregates the MMU and the CLU presented before. The data NoC is only used for communication between the processing resources and the local memories. It is a multi-bus network that connects all PEs and I/O controllers to all shared and banked memory resources.

The CCP prefetches tasks' code before its execution and manages all the dependencies between tasks. It determines the list of eligible tasks to be executed, based on control and data dependencies. It also manages exclusive accesses to shared resources, and non-deterministic processes. Then, task allocation follows online global scheduling, which selects real-time tasks according to their dynamic priority, and minimizes overall execution time for non real-time tasks.

SCMP supports two types of processing resources: DMA and processor. The DMA unit carries

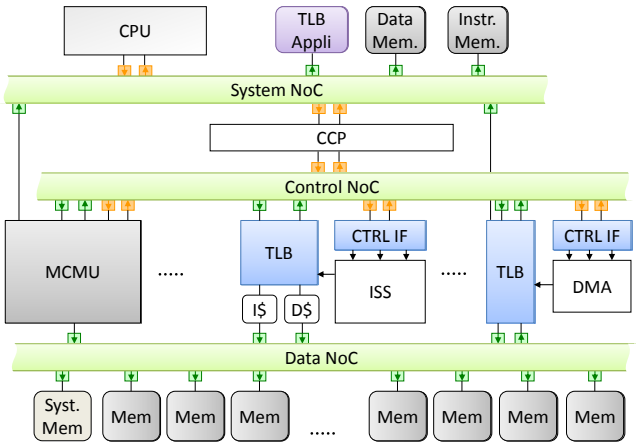


Figure 3.3: SCMP infrastructure modeled in SESAM [152, 153].

out input image transfers between the internal local memories and the external data memory. The processor executes the application C code and is modeled by a cycle-accurate or functional ISS. The ISS boots on a read-only memory, named *system memory*, that contains all the system code. When the initialization is done, it waits for the CCP requests. Currently, SESAM supports only monothreaded ISS architectures. In the next section, we will extend SESAM to support multithreaded processors.

3.1.2 Extending SESAM for multithreaded processors

Initially, SESAM is designed for handling monothreaded ISSes. When replacing the monothreaded ISS with a multithreaded ISS, some modifications to SESAM should be conducted on the processor level and control manager level.

3.1.2.1 Processor level

It consists of multiple multithreaded cores. Each core is a scalar in-order processor. It can process multiple Thread Contexts (TC) concurrently, where each TC is a virtual processor. In this thesis, we consider the case of 2 TCs per multithreaded core, which is suitable for embedded systems requirements as was proven in chapter 2. A Local Thread Scheduler (LTS) synchronizes the execution of the tasks on multiple TCs according to the PE_MT's multithreading policy. Since it is a scalar in-order processor, only one instruction is allowed to be issued from one task at a time. For instance, an IMT core issues the instructions in a round-robin manner between the available TCs, while a BMT core switches between the instructions of the available TCs whenever one is stalled on a long latency event, such as a cache miss. Each TC state is sent to the centralized controller. The TC state can be either *running* normally, *blocked* on a cache miss or I/O, or *waiting* for an execution demand. Based on these values, the controller has a more global view on all the cores' status and can perform the right scheduling decision. Each PE_MT has a shared TLB for all the TCs for proper virtual to physical address translation, and it is connected to a L1 Instruction memory cache (I$) and Data memory cache (D$). In our architecture, the L1$ is segmented per

TC in order to limit cache interferences (see Figure 3.4).

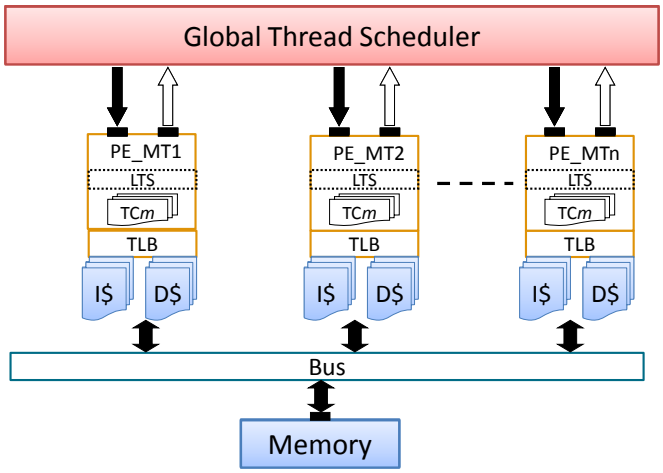


Figure 3.4: Abstraction view of SESAM with multiple multithreaded processors.

3.1.2.2 Control Manager level

The objective of a thread scheduler is to keep busy all the underlying execution resources and balance the load between them. It holds the information of all the SW threads that can be executed on the processors in a *runqueue*. For the case of a multicore system and a SMP OS such as Linux SMP, the scheduler creates a *runqueue* per each core. Tasks are migrated periodically from one runqueue to another whenever a workload imbalance occurs. This works fine with monothreaded cores. However, for multithreaded cores, it is not clear which scheduling technique fits better: whether to assign one runqueue per multithreaded core (VSMP) or one runqueue per thread context (SMTC): the objective is the same, keeping all the multithreaded cores active. VSMP and SMTC are terminologies used by MIPS Technologies.

VSMP: VSMP or Virtual SMP is an OS scheduler architecture that creates one *runqueue* per core (see figure 3.5(a)). If there is one TC per core (monothreaded processor), the scheduler converges to normal SMP. But in case of multiple TCs per core (multithreaded processor), only one *runqueue* is assigned to all the TCs. Then, it is up to the LTS to guarantee an efficient dispatching of the tasks to the free TCs. The main advantage of VSMP is its rapid deployment. Only small modifications to the SMP OS need to be done. However, the scheduler does not have a global view of the workload balance between the TCs and the cores, which might be penalizing in some cases. Consider for example 2 PE_MTs with 4 TCs each, if PE_MT1 has 3 active TCs and PE_MT2 has 1 active TC, then the VSMP scheduler will treat both multithreaded processors equally, since both of them are active.

For static VSMP, a task is allocated on a *runqueue* based on its identifier using the modulo operator. No tasks are allowed to migrate to other runqueues. As for dynamic VSMP, the scheduler scans the execution status of all the PE_MTs. If a multithreaded core is active and another one is free, it migrates a task from the active to the free *runqueue*. However, as stated earlier, the scheduling decision does not take into consideration the exact load of each PE_MT.

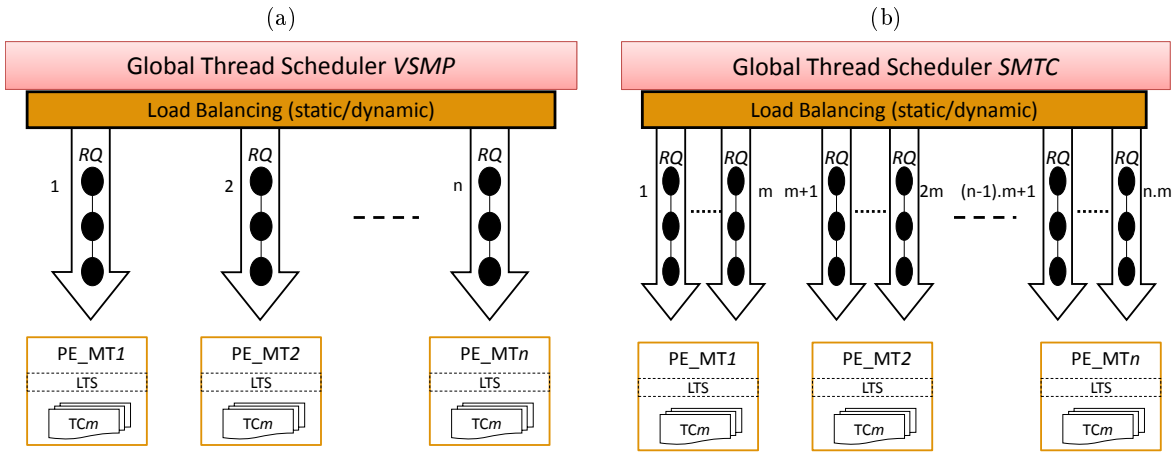


Figure 3.5: a) VSMP scheduler architecture b) SMTC scheduler architecture [18].

SMTC: SMTC or Symmetric Multi-Thread-Context is an OS scheduler architecture that creates one *runqueue* per TC (see Figure 3.5(b)). The scheduler has a more global and correct view of the real physical hardware. Depending on the TC state, the scheduler is able to know which PE_MT is active and how much tasks are scheduled, which facilitates the global workload balancing. This will relieve the LTS from doing local task allocation and concentrate only on its scheduling policy (interleaved, blocked, etc...). Since more execution state informations are available, the scheduling time might take a little longer than in VSMP as we will see later in section 3.3.3.

For static SMTC, tasks are allocated on each TC runqueue based on its identifier using the modulo operator, and no load balancing is allowed. This implies that the LTS has no local scheduling role, since the tasks are already predefined where they will execute. This can be penalizing, since all the TCs are treated equally as a virtual processor which might lead to severe load imbalance. On the other hand, for dynamic SMTC scheduler, the native SMP scheduler code needs to be modified and rethought. At the beginning of a scheduling cycle, the controller receives the execution state of all the TCs. Then, it executes the scheduling algorithm which is decomposed into 3 main parts: sorting, allocation, and verification.
The first phase creates a sorting list of the tasks that are ready to be allocated and executed. The sorting decision depends on the task priority and execution state. For example, a blocked task is put at the end of the sorting list. Then, the first NB_PE tasks are chosen to be allocated, where NB_PE is the maximum number of TCs available in the architecture. For instance, 4 PE_MTs with 2 TCs each have NB_PE equal to 8. The second phase allocates the tasks on the *runqueue* of each TC. Here, the scheduling algorithm has 2 different views of the asymmetric architecture: virtualized mode and non-virtualized mode. In the *virtualized mode*, the execution state of all the TCs of one PE_MT are grouped together in order to form a common architectural state of the PE_MT. A PE_MT is active if at least one TC is active, and an asymmetric MPSoC architecture is executing efficiently if all the PE_MTs are active. Accordingly, ready tasks are allocated on the corresponding TCs runqueue that turns a PE_MT into active. If all the PE_MTs have at least one active TC and there are still ready tasks in the sorting list, then the scheduling algorithm

switches to the *non-virtualized mode*. In this case, a ready task is allocated on a *runqueue* of a free TC. The final phase verifies if the multithreaded processors are well-balanced. For example, consider a system of 2 PE_MTs with 4 TCs each, if PE_MT1 has 3 active TCs and PE_MT2 has 1 active TC, then the dynamic SMTC scheduler will allow the migration of tasks from runqueue TC2 of PE_MT1 to runqueue TC1 of PE_MT2. This scenario is not possible for the VSMP scheduler.

Currently, SESAM supports monothreaded ISS architectures. In order to model SCMP with multiple multithreaded processors, we need to support SESAM with a multithreaded ISS. Therefore, in the next section, we will develop a cycle-accurate multithreaded ISS that can be integrated in the SESAM framework.

3.2 A Multithreaded Instruction Set Simulator

The ISS emulates the behavior of a processor by executing the instructions of the target processor while running on a host computer. Depending on the abstraction level, it can be modeled at the functional or cycle-accurate level. The functional ISS model abstracts the internal hardware architecture of the processor (pipeline structure, register files...) and simulates only the ISA. Therefore, it can be available in the early phase of the MPSoC design for the application software development, where the simulation speed and the model development time are an important factor for a fast design space exploration. Despite all these advantages, many details are hidden by the functional ISS model, such as the pipeline stalls, branch/data hazards and other parameters, which tend to be non-negligible while sizing the architecture. Those parameters evaluate the accurate performance of the processor and the surrounding hardware blocks such as caches, busses, and TLBs.

The cycle-accurate ISS model simulates the processor at an abstraction level between the RTL and the functional model. It presents most of the architectural details that are necessary for processor dimensioning, in order to evaluate in advance its performance capabilities in the MPSoC design. All these advantages come at the expense of a slower simulation speed and a longer development time.

In order to mimics the behavior of the multithreaded processor developed in RTL (see section 2.2) and to capture all the pipeline dependencies, the ISS should be *cycle-accurate*. In addition, as we saw in section 2.2, there is a diminishing return from having more than 2 hardware thread contexts per multithreaded processor from a core area point of view.

In this section, we will build a cycle-accurate multithreaded ISS [16] with 2 TCs based on multiple cycle-accurate monothreaded ISS [15]. Based on a modified ArchC ADL [15], we will build a cycle-accurate monothreaded ISS for a 5-stage RISC processor.This monothreaded ISS will be instantiated and encapsulated multiple times to build a cycle-accurate multithreaded ISS. The multithreaded ISS mimics the behavior of the IMT and BMT processors.

3.2.1 The requirements for ISS and ADL

As stated earlier, we have used SystemC as a simulation environment for MPSoC design space exploration. SystemC supports IP modeling using the Transaction-Level Modeling (TLM) protocol [51, 32]. TLM is a high-level approach to model digital systems where details of communication

among modules are separated from the details of the implementation of functional units or the communication architecture. Therefore, the multithreaded ISS should fit into the SystemC and TLM environments, while providing fast simulation speed and high-accuracy level. So in this section, we investigate the reason to develop a new multithreaded ISS based on SystemC and the choice of the ADL environment.

3.2.1.1 Why a new ISS?

A SystemC/ISS co-simulation environment provides design flexibility by being able to experiment with different types and numbers of processor architectures at the early design stages. This advantage has led researchers [27] to provide SystemC wrappers for traditional standalone ISS such as SimpleScalar [11]. Other works [21, 36] used the same technique for integrating a non-native SystemC ISS into a SystemC/ISS co-simulation environment. However, the main drawback of the SystemC wrappers approach is the slow simulation speed (order of few KIPS) with respect to a standalone ISS (order of hundreds KIPS to MIPS).

On the other hand, standalone multithreaded simulators exist in the literature, mainly targerting SMT type of processors. For example, SSMT [85], M-SIM [128] are SMT extensions on top of SimpleScalar. Other simulators, such as SESC [135] and Sam CMT Simulator kit [104], support the simulation of chip multithreaded (CMT) processors. Despite of their flexibility and parameters variability, these full-system simulators are standalone and require SystemC wrappers with TLM interfaces to be interfaced with other SystemC components.

To our knowledge, no IP-based multithreaded ISS in SystemC with TLM-based interfaces for MPSoC design space exploration exist in the literature. This is the reason why we had to develop a cycle-accurate multithreaded ISS in SystemC and TLM-based interfaces.

3.2.1.2 Which Architecture Description Language (ADL)?

The main part of an MPSoC simulator is the architecture description language (ADL), which generates an ISS at a specific level of abstraction. ADLs' modeling levels are classified into three categories: structural, behavioral, and mixed.

Structural or cycle-accurate ADLs describe the processor at a low abstraction level (RTL) with a detailed description of the hardware blocks and their interconnection. These tools, such as MIMOLA [81], are mainly targeted for synthesis and not for design space exploration due to their slow simulation speed and lack of flexibility.

On the contrary, behavioral or functional ADLs abstract the microarchitectural details of the processor and provide a model at the instruction set level. The low accuracy is compensated by the fast simulation speed. Many languages exist such as nML [46] and ISDL [60].

Mixed ADLs are a trade-off solution between structural and behavioral ADLs. They combine the advantages of both the structural (accuracy) and behavioral (simulation speed) ADLs. It is the best abstraction layer for design space exploration. EXPRESSION [61], MADL[113], LISA [109], and ArchC [114] are examples of mixed ADLs. The last two will be discussed in this literature review since they are mostly used.

LISA: LISA, which stands for Language for Instruction Set Architecture, is developed by the university of RWTH Aachen and is currently used in commercial tools for ARM and CoWare (LISATek). Processor models can be described in two main parts: resource and operation declarations (ISA). Depending on the abstraction level, the operations can be defined either as a complete instruction, or as a part of an instruction. For example, if the processor resources are modeled at the structural level (pipeline stages), then the instructions' behavior in each of the pipeline stages should be declared. Hardware synthesis is possible for structural processor models.

ArchC: A recent type of processor description language called ArchC [118] is gaining special attention from the research communities [20, 39, 73, 154]. ArchC 2.0 is an open-source Architecture Description Language (ADL), developed by the University of Campinas in Brazil. It generates a functional or cycle-accurate ISS in SystemC with its assembler, linker and debugger, from parsing two input files (see Figure 3.6): the processor architecture resource description (AC_ARCH) and the ISA description (AC_ISA) files. The ISS is ready to be integrated with no effort in a complete SoC design based on SystemC. The functional and cycle-accurate processor models are generated by a separate *simulator generator tool*. For instance, *actsim* and *acsim* tools generate the cycle-accurate and functional simulators respectively.

Figure 3.6: The ArchC simulator generator from architecture and ISA description files.

ArchC 2.0 [114] provides many advantages that lacked in its predecessor ArchC 1.6. First, it allows the simulator to be integrated and instantiated multiple times in a full SystemC platform, hence enabling a multiprocessor system simulation. And second, the simulator is wrapped by a TLM interface to allow processor interruptions and TLM communications with external modules. The main feature of ArchC is its ability to generate a cycle-accurate ISS with short development time. Only the behavioral description of the ISA requires an accurate description. The microarchitectural details are generated automatically according to the architecture resource description file. There exists also a graphical framework, called PDesigner [8], based on Eclipse and ArchC processor models, which allows the development and simulation of MPSoCs in SystemC in a friendly manner.

Since ArchC is an open-source language, we can modify the simulator generator to produce a processor with customized microarchitectural enhancements, which makes it a great tool for computer architecture research [119]. Therefore, ArchC is the ADL of choice for building our cycle-accurate ISS in SystemC. In the next subsections, we will explore in more details the generation of a monothreaded and multithreaded cycle-accurate ISS and our contributions/modifications to the ArchC ADL to fit our requirements.

3.2.2 Monothreaded cycle-accurate ISS model

ArchC supports several processor ISA models such as: MIPS-I, PowerPC, SPARC-V8 and ARM. All these models have a working functional ISS. However, the cycle-accurate ISS version generated by *actsim* is not supported for all the ISA models. In our work, we will adapt the MIPS-I R3000 [62, 148] cycle-accurate ISS model, which is described in more details in [15]. The MIPS-I R3000 architecture is almost similar to AntX. It does not have a hardware FPU, thus the FPU instructions are emulated in software by using the compiler option 'msoft-float'. First, we will start by an overview of the generated MIPS-I R3000 cycle-accurate ISS model, then we will show our modifications to the *actsim* tool in order to support ArchC 2.0 specifications.

Overview of the MIPS-I R3000:

The MIPS-I R3000 architecture is implemented as a classic 5-stage RISC processor (IF-ID-EX-MEM-WB) with 32 registers and an integer pipeline. The implemented MIPS-I ISA is similar to the optimized version described in [62]. The control instructions (jump and branch) are executed in the ID stage instead of the MEM stage, and follow the "predicted-not-taken" branch mechanism. Register forwarding is also deployed to allow instructions in the ID or EX stages to get the correct operand values from instructions that are further in the pipeline and did not commit yet. Both techniques reduce the number of pipeline stalls at the expense of adding more logics in the processor datapath.

The *actsim* tool generates the cycle-accurate ISS and the decoder shown in Figure 3.7 from AC_ARCH and AC_ISA files.

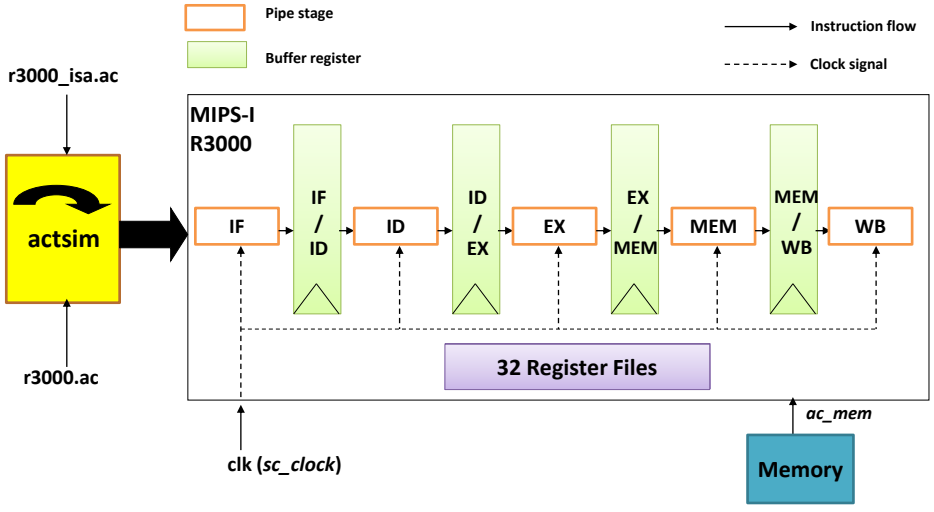


Figure 3.7: R3000 cycle-accurate model generation by actsim tool.

The cycle-accurate simulator is clearly almost similar to the actual processor architecture. The pipeline stages, inter-pipeline registers, register file, program counter (PC), and clock are all included in the simulator.

In our work, we utilize the latest available versions of *actsim* timed simulator generator

tool included in the ArchC 2.0 package, as well as the MIPS-I R3000 cycle-accurate model (r3000-v0.7.2-archc2.0beta3). Both tools are still in their beta versions as they contain some bugs. In other words, the advantages of ArchC 2.0 have not been integrated in the cycle-accurate simulator. Thus, the generated cycle-accurate ISS cannot be integrated in a multiprocessor simulation environment. Therefore, in the next section, we will modify the ArchC actsim tool in order to generate a cycle-accurate ISS compatible with ArchC 2.0 specifications.

A cycle-accurate ISS support for ArchC 2.0:

For each pipeline stage, the initial *actsim* generates a corresponding SystemC module, which is implemented as a SC_METHOD sensitive to the main clock. Implementing the stages as a SC_METHOD works fine in a standalone architecture, with one processor and cache memory. However, the multiprocessor execution will be impossible since the processor model will always own the SystemC execution context. In order to integrate the model in a SoC platform and to communicate with other SystemC IPs, we modify the stages to implement an SC_THREAD module and SystemC wait() function. This solution does not block the other IP modules from executing at the same clock cycle as the processor. A pseudo-code for the EX-stage module is shown in Figure 3.8.

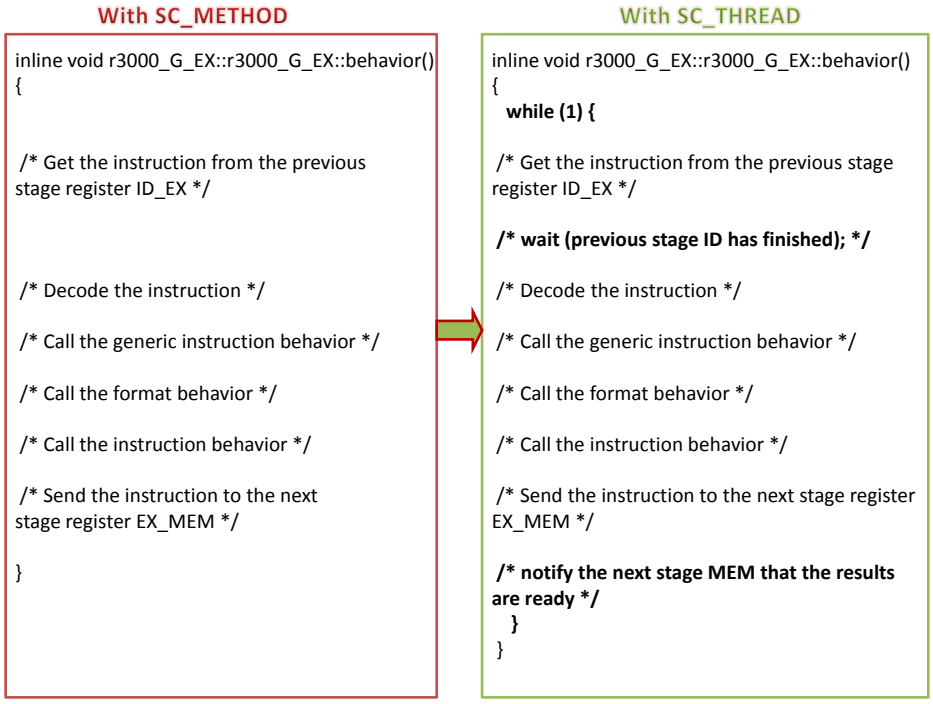


Figure 3.8: Pseudo-code for the EX-stage module in ArchC 2.0.

To model the cycle-accurate pipeline correctly, the procedure is implemented as follows: each stage module executes a while loop, and synchronizes with SystemC wait(). Only the first stage (IF) is sensitive to the main clock and to a synchronization signal (sync), while the others are sensitive

to an input sync sent from the previous stage. When a new clock signal (sc_clock) arrives, the IF-stage executes instruction i, and toggles the sync at its output. Then the ID-stage, which is sensitive to the sync from IF-stage, executes instruction i-1, and toggles its output sync. The same procedure repeats until WB-stage, which executes instruction i-4, and toggles the sync signal that is connected back to the IF-stage. Finally, the IF-stage updates the internal pipeline registers and wait() for the next clock cycle. Note that the pipeline registers are double buffered for proper instruction execution in each stage. Figure 3.9 shows the modified R3000 cycle-accurate model that is generated by 'actsim'.

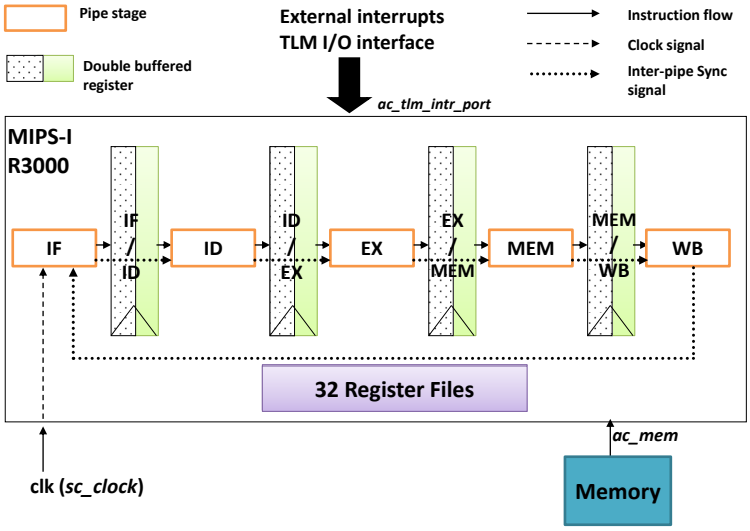


Figure 3.9: New R3000 cycle-accurate model for SoC simulator integration capabilities [15].

The second modification done to the cycle-accurate simulator is the support of a TLM interface and an interruption mechanism. Since the functional simulator already implements the TLM interface, we reused the same code with some modifications to the interruption mechanism. Thus, the ISS SystemC module implements 2 TLM I/O interfaces: the first one receives interrupts from external sources such as a controller (*sc_export*), and the second one sends memory access requests to the memory (*sc_port*). The R3000 pipeline implements *precise exceptions* mechanism in order to avoid any type of pipeline anomalies [62]. So whenever an external interrupt occurs, the R3000 pipeline is flushed. The flushing mechanism occurs by inserting a 'trap' instruction in the IF-stage. The instructions in the pipeline finish their execution normally. When the 'trap' instruction reaches the WB-stage, it signals that the pipeline is now empty, and that the execution of the interrupt service routine is allowed. Then, the appropriate interrupt service routine is called depending on the interrupt type. For instance, we support 3 types of TLM interrupts: *start* a new task, *preempt* the current task with a new task, and *stop* the current task. The TLM interrupt protocol is a modified ArchC TLM protocol [118].

The performance evaluation of our cycle-accurate model necessitates the extraction of pipeline statistic values. Any degradation in the processor performance is mainly due to pipeline stalls. Those stalls arise from two types of sources: data dependencies (data and control hazards), and pipeline interlocks. The latter is due to long memory access latencies when load instructions are in

the MEM stage and there is a data cache miss. In our model, we can measure the total number of pipeline stalls due to data dependencies and pipeline interlocks.

This cycle-accurate ISS can only execute one thread context (TC) at a time. The next paragraph describes the development of a multithreaded ISS, which is able to execute multiple threads at a time.

3.2.3 Multithreaded cycle-accurate ISS model

The multithreaded ISS [16] is designed to be integrated in a typical processor system environment based on SystemC language. It keeps the same TLM I/O interfaces as the monothreaded ISS described in section 2.2.2 in order to look as one ISS/processor to the external world. The multithreaded ISS uses a modular cycle-accurate technique to mimic the behavior of a scalar multithreaded RISC. It encapsulates n pre-validated cycle-accurate ISS for the MIPS-I R3000, each corresponding to one TC. It receives TLM interrupt requests from an external module such as a hardware controller, and sends TLM memory access requests to the caches. Internally, a *scheduler* module synchronizes and schedules all the memory access requests of the n ISS. Figure 3.10 shows the internal structure of the multithreaded ISS model, denoted by PE_MT. Each R3000 ISSi in PE_MT simulates only the pipeline stages, which are described previously in Figure 3.9. The R3000 ISSi is generated automatically in ArchC using *actsim* tool as described in [15], while the other block modules (scheduler, TLM demultiplexer) are developed in SystemC.

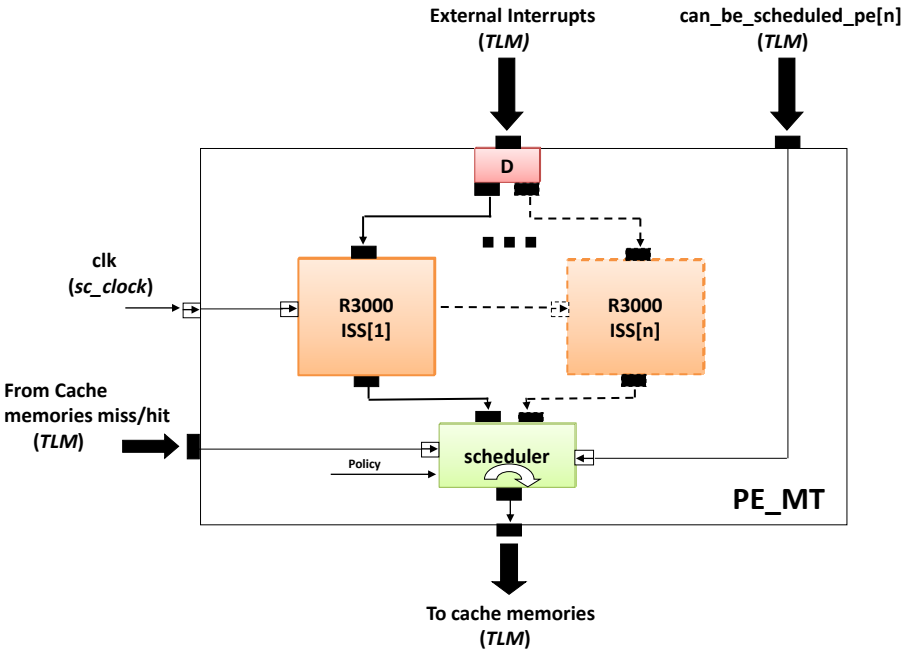


Figure 3.10: Multithreaded ISS model [16].

For the controller, the PE_MT looks as n virtual processors. Each internal ISS is a TC; therefore it has a unique id (vt_id). A vt_id parameter is added to the TLM protocol, so that every incoming and outcoming TLM packet can be tracked in the platform. All the external

interrupts are input to a 1-to-n TLM demultiplexer (labeled as D in Figure 3.10). It checks the vt_id of the TLM packet and then forwards it to the corresponding ISS. Then, the ISS handles the request, updates its internal state and executes the corresponding task. It generates two types of TLM memory requests: an instruction fetch from the IF-stage and a data memory access from the MEM-stage. The *scheduler* module receives TLM memory requests from the n ISS. It synchronizes and schedules the packets according to a pre-defined scheduling policy implemented as an FSM diagram. Then, it selects one of these packets at a time and transfers it to the cache memories. In order to facilitate the scheduling decisions of the *scheduler*, we provide 2 types of information as input to the multithreaded module:

1. The scheduling status (active/idle) of each ISS, which comes from the external controller using the can_be_scheduled_pe[n] input (*sc_export TLM*), where n designates the TC id.

2. The caches hit/miss input (*sc_export TLM*), which inform the scheduler of the status of each memory access request in the caches.

For scalar monothreaded processors, there exist 2 multithreading scheduling techniques: IMT and BMT. Each one has its own FSM diagram implemented in the *scheduler*. Due to the facility of their FSM representation and implementation, the IMT is implemented as a Mealy FSM and BMT as a Moore FSM. Therefore, to add a new multithreading technique, the designer just have to embed the FSM diagram code in the *scheduler* without modifying the other components.

Note that the *scheduler* module is not clocked and is only synchronized by SystemC events. This is important when a functional ISS (not clocked) is used instead of a cycle-accurate ISS, which makes the *scheduler* more general.

In the next sections, we will describe in more details the implementation of an IMT and BMT multithreaded ISS with 2 TCs (n=2).

3.2.3.1 Interleaved multithreading ISS

An IMT processor executes an instruction from one active thread at a time in a round-robin way. Therefore, in any 2 consecutive pipeline stages, there is an instruction from a different TC. However, if one thread is stalled for a long latency event, then the whole pipeline is stalled.

To model this behavior using n separate ISS, the scheduler FSM should allow the execution of one ISS pipeline until completion, and then switch to another active ISS pipeline in zero cycles. During the pipeline execution cycle, it generates a maximum of 2 TLM cache memory requests, one from IF-stage and one from MEM-stage. The FSM switches the thread execution whenever an ISS pipeline is fully processed. Therefore, we differentiate between an IF-stage and a MEM-stage TLM packet by adding a parameter to the TLM protocol. The FSM for the IMT model with 2 TCs, shown in Figure 3.11, is implemented as a Mealy FSM.

Since each ISS is cycle-accurate, small latency pipeline stalls due to data dependencies are captured by the ISS itself. As for the cache misses, they are modeled intuitively by the TLM interface blocking mechanism.

The sequential thread program execution on each ISS does not reflect the actual behavior of the IMT pipeline. In fact, the execution speed of each thread should be divided by n and the pipeline stalls due to data dependencies should be eliminated. This is done by inserting *n-1* "dummy nop"

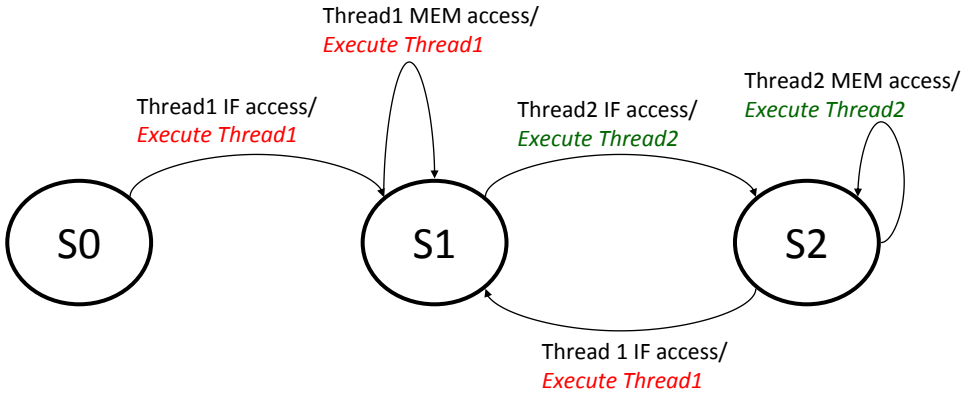


Figure 3.11: Interleaved multithreading scheduler FSM (Mealy machine).

instructions after each fetched instruction. The "dummy nop" instruction does not access the memory, thus does not generate an IF-stage TLM request and is transparent to the *scheduler*. This requires a slight modification to the IF-stage code of the original MIPS-I R3000 model. A pipeline execution model of 2 TCs is shown in Figure 3.12.

	TC1	+	TC2	=	PE_MT
cycle i:					
IF:	add		*nop (dummy)*		add (TC1)
ID:	*nop (dummy)*		sub		sub (TC2)
EX:	beq		*nop (dummy)*		beq (TC1)
MEM:	*nop (dummy)*		j		j (TC2)
WB:	sub		*nop (dummy)*		sub (TC1)
cycle i+1:					
IF:	*nop (dummy)*		add		add (TC2)
ID:	add		*nop (dummy)*		add (TC1)
EX:	*nop (dummy)*		sub		sub (TC2)
MEM:	beq		*nop (dummy)*		beq (TC1)
WB:	*nop (dummy)*		j		j (TC2)

Figure 3.12: Interleaved multithreading pipeline representation.

As we can notice, by overlapping the pipeline stages of all the ISS ("dummy nop" are transparent), we get the pipeline behavior of a scalar IMT processor.

Finally, the *scheduler* should keep track of the scheduling status of each TC using the can_be_scheduled_pe[n] input signals from the controller. If one of the threads is scheduled/descheduled, then the *scheduler* informs the other ISS to adjust the number of "dummy nop" instructions.

3.2.3.2 Blocked multithreading ISS

A BMT processor executes one thread as on a monoprocessor, and switches to another thread whenever a cache miss occurs. Thus, small latency pipeline stalls such as pipeline dependencies are not tolerated by this model. Therefore, a thread status is defined as:

1. ACTIVE: if Thread[i] is scheduled and executing properly without long latency events.

2. NOT ACTIVE: if Thread[i] is not scheduled by the controller or has a long latency event such as a cache miss and TLB miss or is stalled on data synchronization with another Thread[i+1].

The *scheduler* FSM requires external signals from the caches (cache memories miss/hit signals shown in Figure 3.13) in order to perform its decision. In our work, we implement a "greedy" BMT protocol, where one main thread (R3000 ISS1) has a higher priority than the others (R3000 ISS2 to R3000 ISSn), thus its execution speed is not altered. This scenario considers that the low priority threads are helper threads. However, if there are not enough memory stall latencies, the "greedy" protocol may cause starvation to some helper threads. The FSM diagram for 2 TCs, shown in Figure 3.13, is implemented as a Moore FSM.

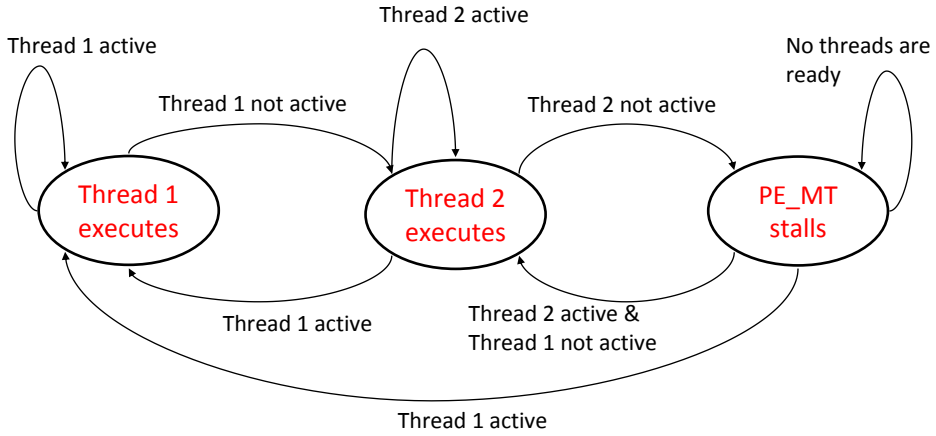


Figure 3.13: Blocked multithreading scheduler FSM using greedy protocol (Moore machine).

Initially, Thread1 executes as long as there is a cache hit. Whenever there is a miss, Thread2 starts the execution and fills the stalling slot cycles of Thread1. When Thread1's data is returned, it resumes the execution. Otherwise, Thread2 continues the execution until there is a miss. Then the whole processor is stalled and waits for one of the threads' returned data in order to resume the execution, with a higher priority to Thread1 in case of a simultaneous response.

Opposed to the IMT model, the BMT model does not require any changes to the monothreaded ISS, such as "dummy nop" insertions and memory access packet distinction. The latter implies that the BMT model is sensitive to a long latency event, whether it comes from an IF-stage or MEM-stage packet.

In the next section, we will evaluate the performance of the multithreaded processors in SCMP architecture (MT_SCMP) in order to evaluate its efficiency.

3.3 Performance evaluation

In this section, we will evaluate the MT_SCMP architecture, which is modeled in the SESAM simulation framework described in section 3.1 with multithreaded ISS described in section 3.2 as processing elements. We run two types of applications: control-flow and dataflow, which are described in section 3.3.1. Then, we decide which multithreaded processor type (IMT v/s BMT) suits best for the asymmetric MPSoC requirements. We compare two global thread scheduling strategies (VSMP v/s SMTC) and choose the one that gives the best performance. The last two parts evaluate the transistor efficiency of MT_SCMP and compare it to that of SCMP with monothreaded processors. First, we compare their area occupation, then their performances using both types of applications.

3.3.1 Applications description

As stated earlier, we evaluate the MT_SCMP with 2 types of applications: control-flow and streaming.

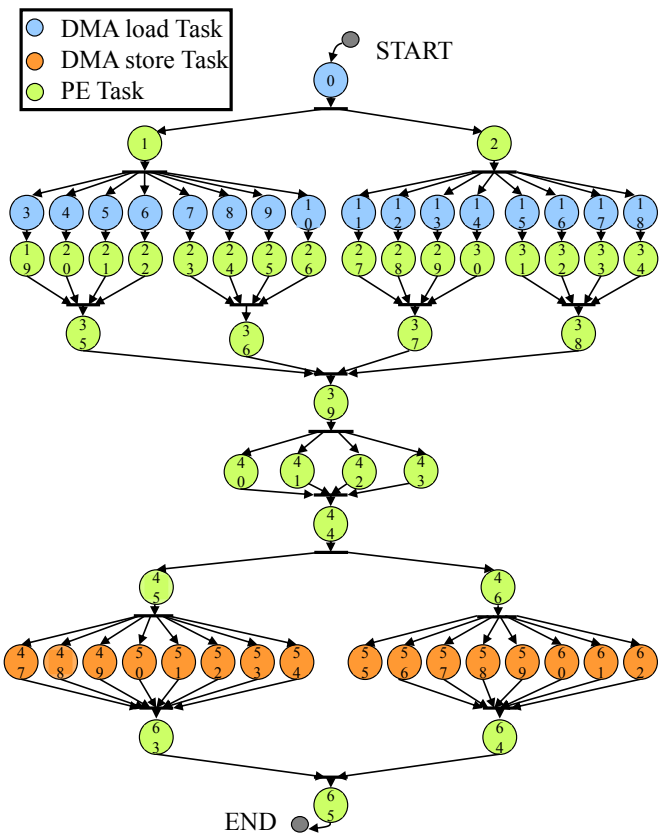
3.3.1.1 Control-flow: labeling algorithm

For the control-flow application, we have chosen an embedded application called ADAS (Advanced Driver Assistance Systems). It consists of a camera installed in a car that detects humans on the roads, in order to detect a pre-crash situation. This is a critical application for automotive systems and is particularly relevant to this study in terms of *dynamism, parallelism and control dependencies*. In ADAS, one part of the obstacle detection process is the connected component labeling algorithm. The labeling algorithm transforms a binary image into a symbolic image so that each connected component is uniquely labeled based on a given heuristic. It detects unconnected regions in binary images. Various algorithms have been proposed [71] [77], but we have chosen an algorithm using the contour tracing technique [45]. This very fast technique labels an image using only one single pass over the image. It can detect external and internal contours, and also identify and label the interior area for each component.

The initial algorithm is parallelized by creating independent tasks with control dependencies explicitly represented in a CDFG as shown in Figure 3.14. Thus, the application follows the control-flow programming model.

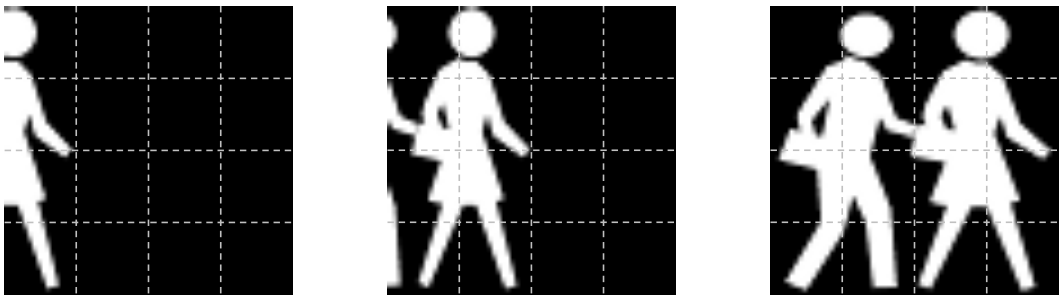
To get multiple independent tasks, we cut the image into sub-images and apply the algorithm on each sub-image. Then, we carry out successively a vertical and a horizontal fusion of labels in analyzing frontiers between sub-images, and finally we construct the corresponding tables between labels and change in parallel all labels into sub-images. As input images, we use a 128x128 pixel image, cut into 16 8x8 sub-images. This implies that the maximum parallelism is *16*. The input images are a sequence of 3 images taken at different time intervals. They show 2 pedestrians crossing a road (Figure 3.15), and they are close to a car (order of 10 meters). The labeling algorithm is implemented on each image.

The computation requirement differs for the 3 images as shown in Figure 3.16. Pedestrian3 image takes about **3 times** more processing than pedestrian1 image. Pedestrian1 image has 25% of its sub-images executing the labeling code, since the others are black sub-images (non-balanced workload). Similarly, pedestrian2 image has 50% (semi-balanced workload) and pedestrian3 has



TASK(S)	Function
0	DMA loading whole image
1 & 2	Fictive task
3 to 18	DMA loading all subimages
19 to 34	Labeling
35 to 38	H_correspondence and fusion
39	Data Synchronization
40 to 43	V_correspondence
44	Fusion
45 and 46	synchronization
47 to 62	DMA storing all subimages
63 and 64	synchronization
65	Free shared data and End of application

Figure 3.14: CDFG of the connected component labeling algorithm used as input for the centralized control core. The labeling algorithm is decomposed into 16 parallel tasks, where each task performs the labeling algorithm independently on a sub-image.



(a) Pedestrian1 (b) Pedestrian2 (c) Pedestrian3

Figure 3.15: 3 images of 2 pedestrians crossing a road.

100% (fully-balanced workload). This behavior reveals the *dynamism* of the connected component labeling application with respect to the input data.

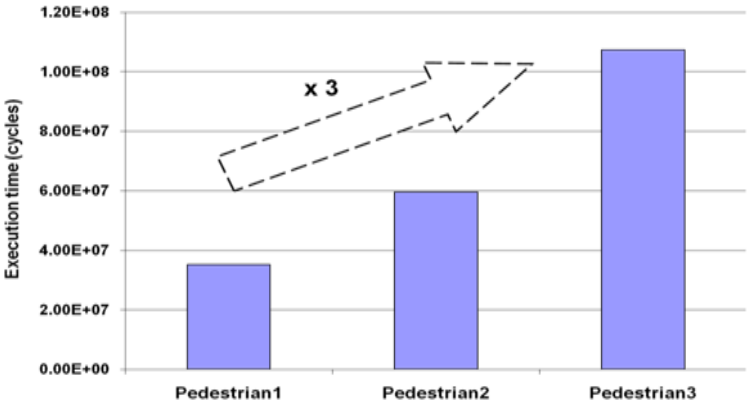


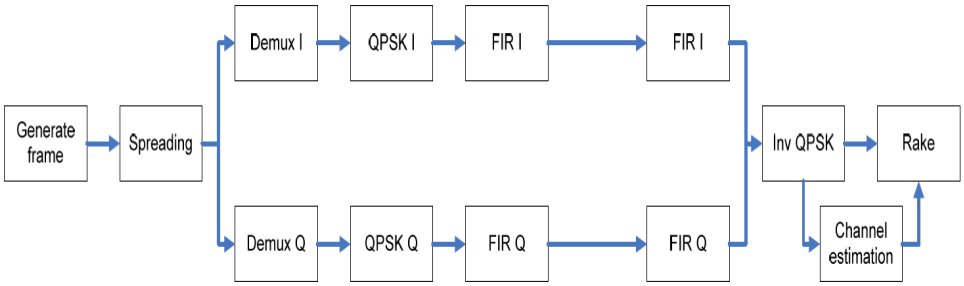
Figure 3.16: Dynamic behavior of the connected component labeling application when executed with different pedestrian images on a single-processor.

3.3.1.2 Dataflow: WCDMA

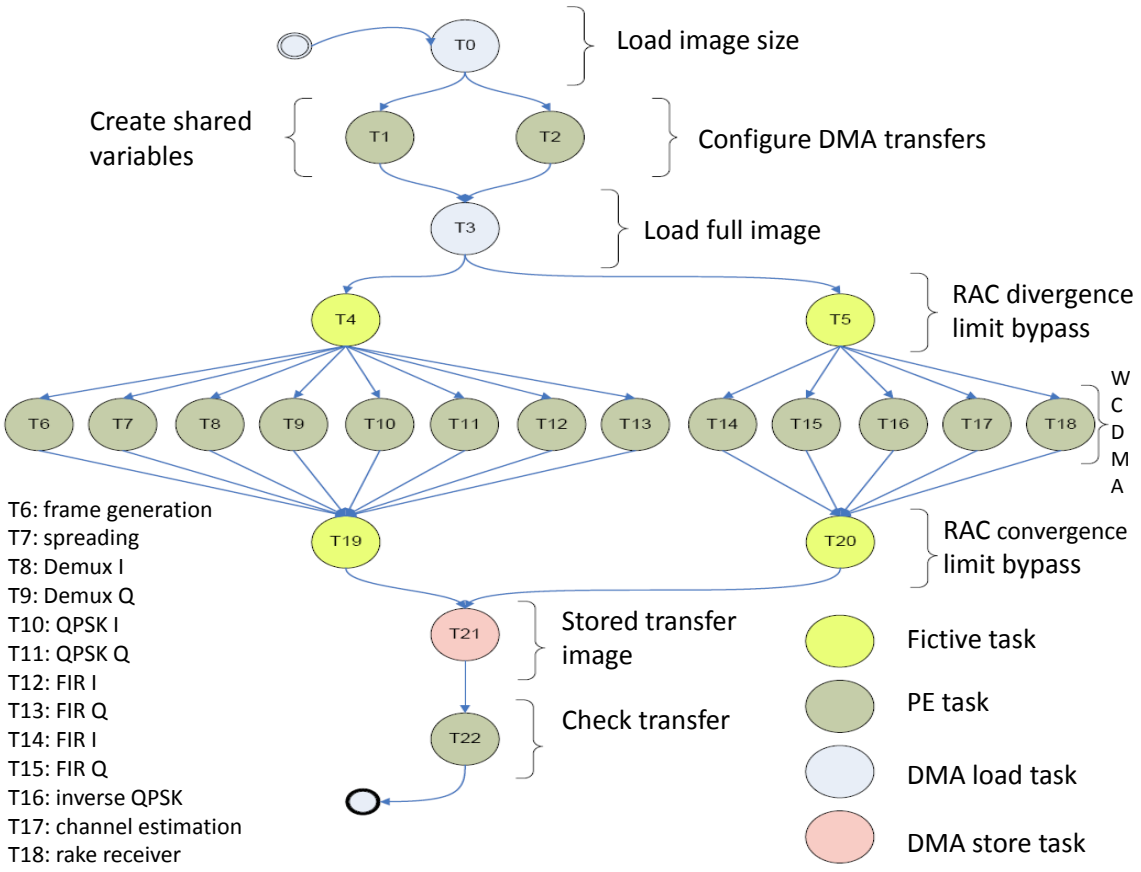
The dataflow or streaming application is a complete WCDMA (Wideband Code Division Multiple Access) encoder and decoder [117]. This communication technology is based on the use of Orthogonal Variable Spreading Factor (OVSF) to allow several transmitters to send information simultaneously over a single communication channel. This application uses a rake receiver with a data aided channel estimation method. Known pilot symbols are transmitted among data. The channel estimation algorithm operates on the received signal along with its stored symbols to generate an estimate of the transmission channel. The processing of pilot frames generates a dynamic behavior of the application, since this induces a variable execution length. Different blocks of the application are shown in Figure 3.17(a). The application is pipelined into 13 different tasks as shown in Figure 3.17(b). To maximize the concurrency between pipelined tasks, a double buffer is used between each task. Thus, tasks can independently execute the next frame from the previous pipelined stage results.

3.3.2 Which multithreaded processor system?

In order to choose which multithreaded processor suits best for MT_SCMP, we run the labeling algorithm with pedestrian3 image (see Figure 3.15(c)). For the MT_SCMP configuration, we use one multithreaded processor, which can be either IMT or BMT with 2 TCs. We vary the L1 I$ and D$ size from 512-B to 8-KB. The L1 caches are direct-mapped with 16 Bytes/line. In fact, by fixing the cache associativity and the number of words per line, we only compare IMT and BMT without any cache interference. For the labeling application, the parallel tasks executes almost the same code but with different data. Therefore, we segment the L1 caches per TC, and we give each TC the same amount of cache memory. This is the best cache architecture for this type of application. For instance, a L1 cache size of 1-KB means 1-KB for each TC. In Figure 3.18, we show the L1 I$ and D$ miss rate of the overall cache memory. The I$ and D$ miss rates vary from 15.5% to 0.91% and from 14.7% to 0.34% respectively. It is clear from the results that for cache sizes more than 8-KB,



(a) WCDMA Application



(b) WCDMA CDFG

Figure 3.17: WCDMA application and CDFG.

we would reach almost an ideal cache memory with no miss rate, which makes the multithreaded processor not an interesting solution. Therefore, we will limit our exploration to 8-KB.

In Figure 3.19(a), we compare the performances in execution cycles of the monothreaded, IMT and BMT processors. To better understand the sources of latencies, we decompose the total execution time into 6 parts: effective execution time when the processor is never stalled, stall time

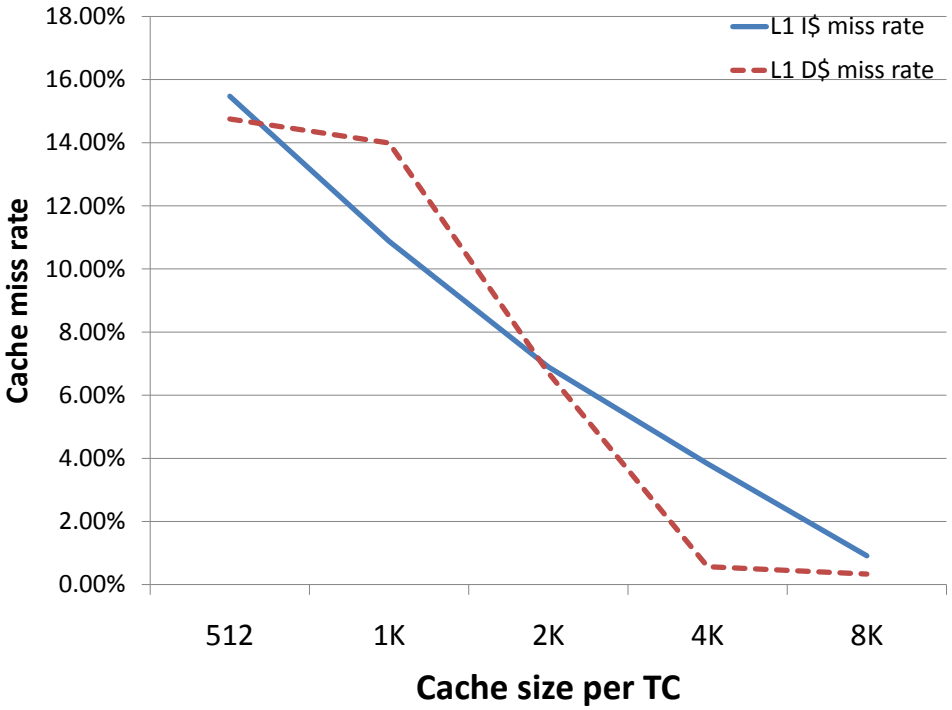


Figure 3.18: L1 I$ and D$ miss rate for cache sizes between 512-B and 8-KB and for the connected component labeling application.

penalty due to I$ miss, stall time penalty due to D$ miss, pipeline stall due to data dependencies, context switch overhead in the case of BMT processor, and other sources of pipeline stalls that can come from a TLB miss, cache flush, MCMU processing, etc...These statistics can be revealed from our multithreaded ISS since it is cycle-accurate.

The results show clearly that the BMT processor overcomes the performance of IMT processor for all cache configurations. By varying the cache sizes, we vary the penalties due to cache misses. The statistics show that the monothreaded processor is stalled for a significant portion of execution time because of I$ and D$ misses, which is almost 75% of the total time for 512-B. Under these conditions, the BMT processor is able to mask those cache miss latencies by executing instructions from another TC. And even if the context switch penalty is so high because there are lot of cache misses, the performance gained in BMT is still higher than the IMT. As shown in Figure 3.19(b), the BMT has a performance gain of 36% compared to the monothreaded processor, while the IMT has a gain of only 15.5%. As the size of the L1 caches increase, the cache miss rate decreases and so is the pipeline miss penalty. Therefore, the multithreaded processors does not have enough cache miss penalties to be masked, and their performance gain is reduced to 9.1% (BMT) and 5.3% (IMT) for the 8-KB L1$.

Thus, since the BMT has a better performance and a smaller area than IMT, it suits best for MT_SCMP and we will choose it for the future explorations. It is worth to note that the results taken in this section are different from that of section 2.3. In fact, in MT_SCMP, the sources of pipeline stalls and their weight are much higher than that of standalone AntX. This is why BMT

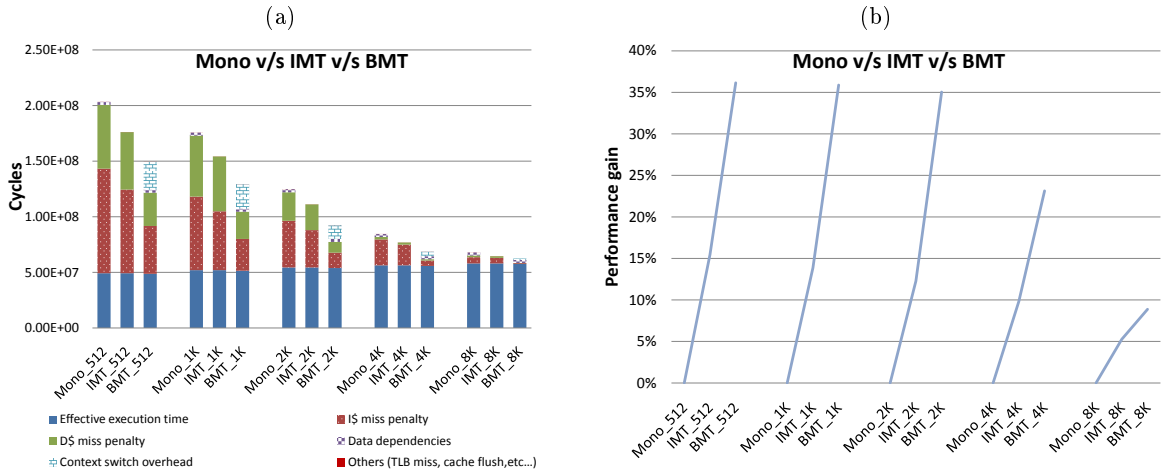


Figure 3.19: Performance results of MT_SCMP with 1 processor: Monothreaded v/s IMT v/s BMT. The sizes of the L1 I$ and D$ are varied simultaneously between 512-B and 8-KB. The performance gain of each multithreaded processor with respect to the monothreaded is plotted in b).

performs better under MT_SCMP conditions than IMT for small memory access latencies.

3.3.3 Which global thread scheduling strategy? VSMP v/s SMTC

In this section, we analyze which global thread scheduling strategy suits best MT_SCMP [18]. For this reason, we run the labeling algorithm on 4 different types of scheduling strategies: VSMP static, VSMP dynamic, SMTC static and SMTC dynamic. We vary the workload by varying the input images shown in Figure 3.15. For all the experiments, the number of PE_MTs varies between 1 and 8, where each PE_MT has 2 TCs. The L1 I$ and D$ size is fixed to 2-KB, which gives a cache miss rate around 10% for the connected component labeling application. In fact, since we implement the blocked multithreading policy, there should be sufficient pipeline stalls (i.e. cache misses) in order to guarantee that all the TCs will execute, otherwise we risk resource starvation and some TCs will never have their share of execution.

3.3.3.1 Static v/s dynamic thread scheduling

In this experiment, we compare the static and dynamic algorithms of the VSMP and SMTC thread scheduling architectures. In Figure 3.20(a), we plot the number of cycles taken to execute the static and dynamic versions of VSMP for the 3 pedestrian images. The speedup is more significant for 2 and 4 PE_MTs, which reaches 40% for the pedestrian1 image and goes down to 7% for pedestrian3 image. However, the execution time for 1 and 8 PE_MTs is similar for static and dynamic VSMP. In fact, in VSMP, there is only one runqueue per PE_MT, which gives the same performance on 1 PE_MT for both static and dynamic algorithms. As for 8 PE_MTs, the VSMP algorithm allocates 2 tasks on each runqueue in the same way for static and dynamic algorithm, since the VSMP does not see the actual workload per PE_MT.

For the SMTC scheduler, the speedup between the static and dynamic versions is more signifi-

cant than VSMP as shown in Figure 3.20(b). It reaches 51% for the pedestrian1 image. To understand better the reason for this large performance difference, let's consider the case of SMTC static v/s dynamic in Figure 3.20(b) for pedestrian1 image 3.15(a) and 2 PE_MTs, where the speedup is 51%. The image is cut into 16 sub-images, and the labeling tasks of each sub-image are allocated first horizontally then vertically. The sub-images identifiers are set from 1 to 16 respectively. This means that tasks [T1,T5,T9,T13] contain pixels that need to be processed by the labeling algorithm, which implies more processing times. If PE_MT1{TC1,TC3} and PE_MT2{TC2,TC4}, then TC1=[T1,T5,T9,T13]; TC2=[T2,T6,T10,T14]; TC3=[T3,T7,T11,T15]; TC4=[T4,T8,T12,T16]. Thus we can clearly see that all the heavy computation tasks are assigned to TC1 runqueue for the static SMTC scheduler, which leverages the need of a dynamic scheduler.

One other observation is the speedup for 1 and 8 PE_MTs cases. This can be explained by the fact that the dynamic SMTC scheduler is able to see the exact occupation rate of each TC and balance the workload between the runqueues so to take a full advantage of the multithreaded processors. For example, for the pedestrian1 image with 1 PE_MT{TC1,TC2}, all the heavy computing tasks are allocated on TC1 runqueue in the static SMTC version. This means that TC2 runqueue will be processed much faster than TC1, and the remaining tasks on TC1 will not be migrated in the static version, which is not the case in dynamic SMTC.

In summary, when we have a balanced workload as in the case of pedestrian3 image, the difference between static and dynamic is not significant (less than 10%). But in real-case scenarios, we expect on average a semi-balanced workload similar to pedestrian2 image.

3.3.3.2 VSMP v/s SMTC

In Figure 3.20(c), we compare the dynamic algorithm of VSMP and SMTC for the 3 pedestrian images. In all the configurations, the dynamic SMTC has a better performance than dynamic VSMP. The speedup varies between 1% and 11%. Again, the speedup is more important for non-balanced and semi-balanced workloads, and especially for the cases with large number of multithreaded processors (PE_MT = 8). In fact, when the number of multithreaded processors increases, the complexity of finding the optimal scheduling decision also increases. This is due to the fact that the scheduling decision for multiple multithreaded processors is different and more complex than monothreaded processors, which necessitates the need of an effective and reactive global thread scheduler for proper load balancing between the runqueues. Hence, dynamic SMTC gives superior performance on dynamic VSMP, and this difference would be more important if the number of TCs per PE_MT is bigger than 2.

3.3.3.3 Scheduling overhead

Finally, in Figure 3.20(d), we compare the complexity of the 4 types of thread schedulers. The results show the average number of cycles taken to complete a scheduling tick. As expected, the static versions take less time to finish a scheduling tick compared to their corresponding dynamic versions. One clear observation is that the scheduling tick of the SMTC dynamic is much longer than VSMP dynamic, especially when the number of multithreaded processors increases (around 4000 clock cycles difference for 8 PE_MTs). The difference is expected to increase more when the number of TC per PE_MT is bigger. This result is not surprising, since the dynamic SMTC

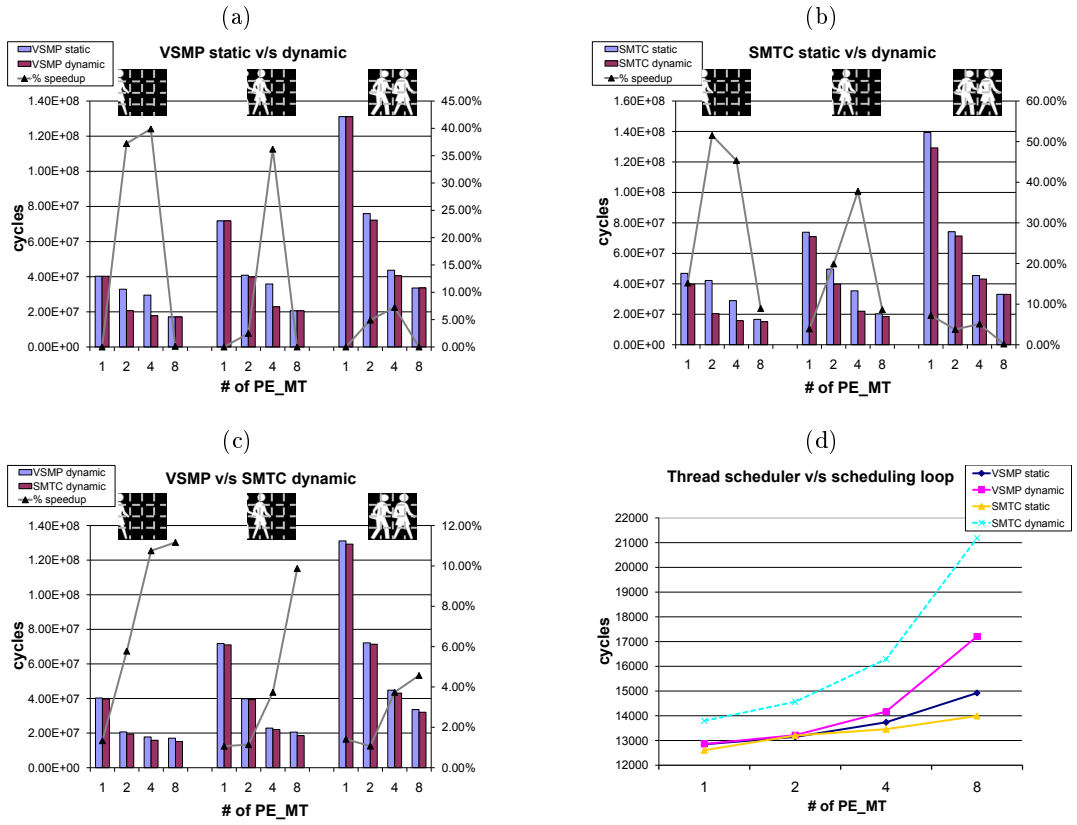


Figure 3.20: Performance comparison of the different thread scheduling strategies: a) VSMP static v/s dynamic. b) SMTC static v/s dynamic. c) VSMP dynamic v/s SMTC dynamic. d) Thread scheduler v/s number of clock cycles per scheduling tick. The speedup line in the first 3 figures is the speedup with respect to the same configuration.

algorithm has one runqueue per TC, thus performing more tests in order to choose the best TC's runqueue to allocate the SW task. (see section 3.1.2). In fact, the SMTC has a complexity of O(NxM), while VSMP is O(N), where N is the number of multithreaded processors and M is the number of TCs per PE_MT. But, as we saw previously from the results, the scheduling overhead does not impact the performance, since in an asymmetric architecture, the global thread scheduler executes in parallel to the computation. On the other hand, in a symmetric approach, the scheduler executes on the same processor as the computation and hence needs to finish the scheduling tick as fast as possible.

In summary, the dynamic SMTC global thread scheduling policy is retained for future exploration.

3.3.4 SCMP v/s MT_SCMP: chip area

In this section, we estimate SCMP and MT_SCMP areas. We want to know the overhead implied due to multithreading. For this reason, we estimate the area of the components that are affected by multithreading and contribute to increase the overall die area. For instance, the PE system

(processor and caches) and the interconnection busses (control and data) are the key components that are affected by multithreading. Components such as CCP, on-chip memory and MCMU are not taken into consideration, since their area is the same for SCMP and MT_SCMP.

In Figure 3.21, we show the area of the processors (Figure 3.21(a)), cache memories estimated with the CACTI 6.5 tool (Figure 3.21(b)), and the multibus interconnection network (Figure 3.21(c)). For the multibus, we do not consider the area of the wires, so it is only the synthesis results of the I/O ports, buffers, and arbiter. The technology used by CACTI tool is based on ITRS roadmap [125], but it is not similar to TSMC technology. Therefore, the processor system is not synthesized with the same technology, but this gives us an idea of the relation between cache size and processor size. So, all these components are synthesized/estimated in 40 nm technology.

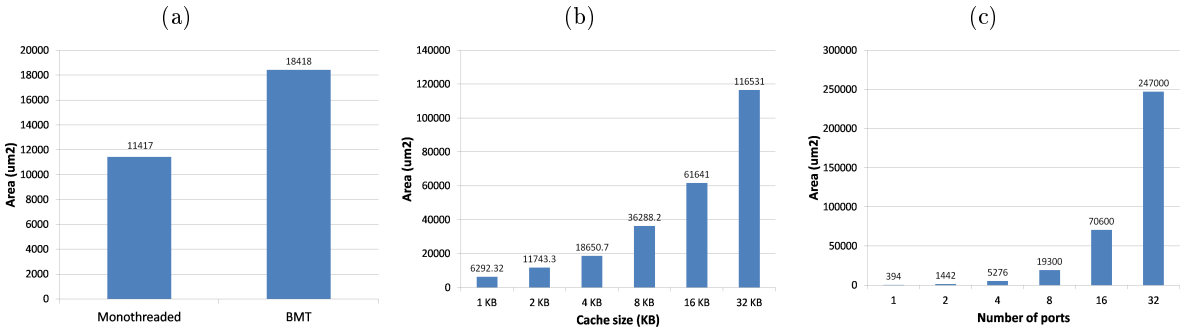


Figure 3.21: Components area in 40 nm technology for a) AntX processor monothreaded and BMT (TSMC) b) L1 cache memory with 16 Byte line and direct-mapped (CACTI 6.5) c) Multibus interconnection network with variable number of I/O ports (TSMC).

Based on these components area values, we estimate SCMP and MT_SCMP areas. We vary the number of processors between 1 and 32, and the size of the caches between 1-KB and 32-KB. In Figure 3.22, we show the percentage increase of each SCMP system with respect to the reduced SCMP system: 1 monothreaded PE and 1-KB L1 cache memories.

We can notice that as the cache size increases, the difference between SCMP and MT_SCMP systems is negligible. This is because we are using very small cores compared to big L1 cache sizes. Another observation is the importance and possible transistor efficiency gain of MT_SCMP system. In fact, a MT_SCMP architecture with n multithreaded processors can process the same number of threads as a SCMP system with 2n monothreaded processors. The area overhead of the SCMP system is much bigger than that of MT_SCMP. So, if we show that the MT_SCMP system can give comparable performance to that of SCMP, then MT_SCMP is more transistor efficient. In the next section, we will examine the performance of both systems using two applications with different execution models.

3.3.5 SCMP v/s MT_SCMP: performance

In this section, we will compare the performance of SCMP and MT_SCMP architectures. We use 2 embedded applications that suit our requirements in terms of dynamism and parallelism. Each application has a different execution model. The first one, the connected component labeling algo-

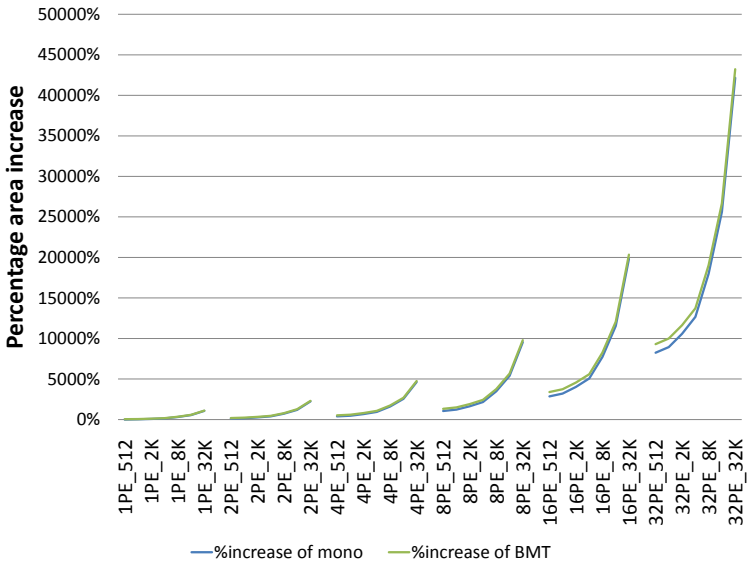


Figure 3.22: Percentage area increase of SCMP with number of processors varying from 1 to 32 and L1 cache size varying from 1-KB to 32-KB. The processors can be monothreaded and BMT. The percentage area increase is compared with respect to the initial SCMP system: 1 monothreaded PE and 1-KB L1 cache memories.

rithm described earlier in section 3.3.1.1, has a control-flow execution model. The second one, the WCDMA application from the telecommunication domain (see section 3.3.1.2) has a streaming/-dataflow execution model. Each execution model has its own characteristics that induce different types of processor stalls as we will see in the following sections.

3.3.5.1 Control flow

In this experiment, we evaluate the performance and transistor efficiency of MT_SCMP with respect to SCMP, by running a control-flow application, connected component labeling, with pedestrian3 image (Figure 3.15(c)) as input. As depicted in Figure 3.14, this application has a maximum thread-level parallelism of 16. Therefore, since the multithreaded processor has 2 TCs, we vary the number of processors from 1 to 8. In addition, we vary the L1 I$ and D$ sizes from 512-B to 8-KB. In fact, for a cache size greater than 8-KB, the cache hit is almost 100% (see Figure 3.18), so there is no interest in implementing a multithreaded processor. The L1 caches implements the write-back + write-allocate cache coherence policy, since it generates less traffic on the bus between the caches and the memory. In MT_SCMP, each TC has an equal size of cache. For instance, a 1PE_1K system means that each processor in SCMP has 1-KB of L1$ and each TC in MT_SCMP has 1-KB. So the L1 cache sizes in MT_SCMP is doubled. This keeps the same cache miss rate with respect to SCMP, thus it is easier to compare the performances.

In Figure 3.23(a) and Figure 3.23(b), we compare the performances of SCMP and MT_SCMP respectively. We decompose the total execution time into 3 parts: effective execution time, CCP scheduling overhead, and synchronization overhead. The 'CCP scheduling overhead' is the time taken by the central controller CCP to schedule threads on the processors. The 'synchronization

overhead' parameter is equal to zero in the control-flow application, since it is a run-to-completion execution model and no threads are synchronizing with each other. In addition to those parameters, we add the 'context switch penalty' for the MT_SCMP. This penalty is due to switching the TC and flushing the processor pipeline during a cache miss.

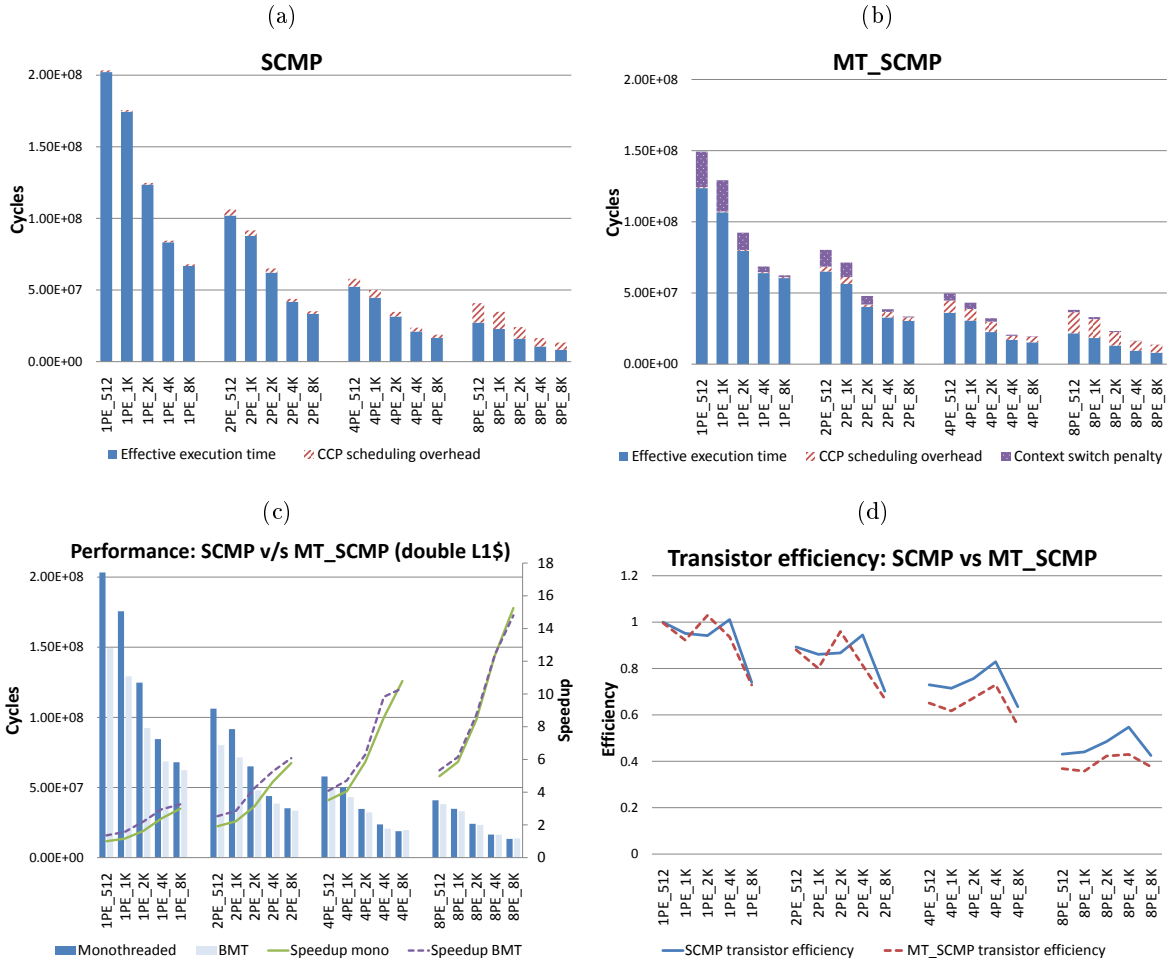


Figure 3.23: Performance of SCMP v/s MT_SCMP for control-flow (labeling) application. The number of processors is varied from 1 to 8 and the cache sizes from 512-B to 8-KB. The cache sizes is per TC. 1PE_1K means The performance is plotted in clock cycles a) SCMP system performance b) MT_SCMP system performance c) Comparison between SCMP and MT_SCMP performance and speedups d) Transistor efficiency of SCMP and MT_SCMP with respect to initial system: 1 PE and 512 Byte cache size.

The performance results show that the 'CCP scheduling overhead' is high when the number of processors is high for both systems. This is due to 2 reasons: the CCP takes longer time to complete a scheduling cycle when the number of processors is high, and there is no enough thread parallelism to occupy all the processors. As for MT_SCMP, we notice that the penalty due to context switching is high when the cache sizes are small. This is logical because there are more cache misses, hence more local thread context switches.

In Figure 3.23(c), we compare the performance of both systems and we plot the speedup of

each system with respect to the initial SCMP system: 1 PE and L1$ size equal to 512 Bytes. We notice that MT_SCMP overcomes the performance of SCMP system. The speedup is higher when the cache misses are higher. However, when the cache misses are negligible, both SCMP and MT_SCMP have almost similar performances.

Finally, we compare the transistor efficiency of both systems with respect to the initial SCMP system (Figure 3.23(d)). The estimated area of each system is taken from section 3.3.4. The transistor efficiency can be greater, equal or less than 1, which corresponds to supralinear, linear, and sublinear scalability. The latter can be referred as 'typical scalability'. For the majority of the systems, the transistor efficiency is sublinear. We can notice that the transistor efficiency of the SCMP system is higher than MT_SCMP for most of the system configurations.

To summarize, the MT_SCMP architecture gives better peak performance but lower transistor efficiency than the SCMP architecture for the control-flow application. This is the case when MT_SCMP has a double L1 cache size than SCMP. However, if we consider that the L1 cache sizes are equal for both architectures, which means that each TC in MT_SCMP has a half cache size or in other words the PE has an equal cache size as that of SCMP, the results would be different as shown in Figure 3.24.

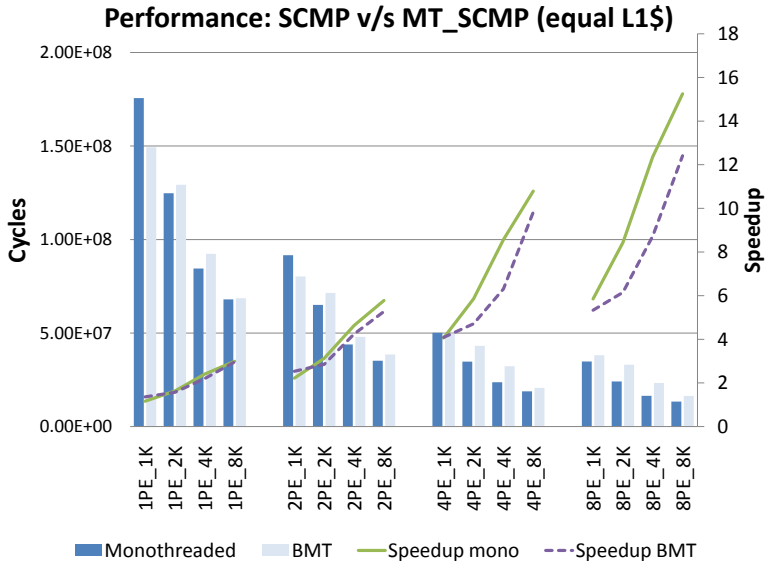


Figure 3.24: Performance and speedup comparison between SCMP v/s MT_SCMP for control-flow (labeling) application. The number of processors is varied from 1 to 8 and the cache sizes from 1-KB to 8-KB. The cache sizes is per PE.

By dividing the cache size of each TC, the number of cache misses is higher for MT_SCMP than for SCMP. As was shown previously in Figure 3.18, the cache miss rates does not have a linear increase/decrease by varying the cache size. This is why for some configurations (i.e: L1 D$ = 1-KB), the cache miss rate difference between a SCMP and a MT_SCMP architecture is not significant, therefore the MT_SCMP performs better. However, for most of the other cache configurations, the difference is almost linear. Thus, even if the multithreaded processors in MT_SCMP are able to hide some of these latencies, they are not able to perform better than the SCMP architecture.

This analysis is application dependent and might vary from one application to another depending on their L1 cache access behavior. In the next section, we will evaluate the performances of SCMP and MT_SCMP with respect to a streaming application.

3.3.5.2 Dataflow

In this experiment, we evaluate the performance and transistor efficiency of MT_SCMP with respect to SCMP, by running a streaming/dataflow application described in section 3.3.1.2: Wide-band Code Division Multiple Access (WCDMA).

For this experiment setup, we vary the number of processors from 1 to 8 and the L1 I$ and D$ sizes from 256-B to 4-KB. In fact, for a cache size greater than 4-KB, the cache hit is almost 100%, so there is no interest in implementing a multithreaded processor.

In Figure 3.25(a) and Figure 3.25(b), we compare the performances of SCMP and MT_SCMP respectively. In the streaming execution model, the 'synchronization overhead' parameter contributes to a non-negligible part of the overall execution time. Initially, we believed that this overhead can be hidden by using multithreaded processors instead of monothreaded. However, as Figure 3.25(b) shows, it still occupies almost the same percentage of the overall execution time. In fact, the CCP scheduler implements a dynamic thread scheduling. When a thread is stalled on a synchronization, the CCP scheduler is directly informed by the MCMU. Thus, if there are ready tasks in the scheduling queue, it preempts the stalled processor and executes an active task instead. The same execution behavior is also applied to MT_SCMP. Thus, the only real limitation is the application parallelism. If there are not enough task parallelism, then SCMP/MT_SCMP might suffer from processor stalls due to synchronization overhead. This scenario occurs for the case of 8 processors.

In Figure 3.25(c), we compare the performance of both systems and we plot the speedup of each system with respect to the initial SCMP system: 1 PE and L1$ size equal to 256 Bytes. Similarly to the control-flow application, the MT_SCMP overcomes the performance of SCMP system. The speedup is higher when the cache misses are higher (small cache size). However, when the cache misses are negligible, both SCMP and MT_SCMP have almost similar performances. We notice also that for 8 processors, there is no difference between the performances. In fact, for this configuration, the application has reached the maximum level of parallelism and the processors remain stalled most of the time.

Finally, we compare the transistor efficiency of both systems with respect to the initial SCMP system (Figure 3.25(d)). The estimated area of each system is taken from section 3.3.4. For the majority of the systems, the transistor efficiency is sublinear. We can notice that the transistor efficiency of the SCMP system is higher than MT_SCMP for most of the system configurations.

To summarize, the MT_SCMP architecture gives better peak performance but lower transistor efficiency than the SCMP architecture for the streaming/dataflow application.

3.3.6 Synthesis

In this chapter, we explored the advantages/disadvantages of using multithreaded processors in an asymmetric MPSoC architecture: SCMP. For this reason, we developed a new multithreaded ISS in SystemC language and integrated it in SESAM, which is the simulation environment for SCMP.

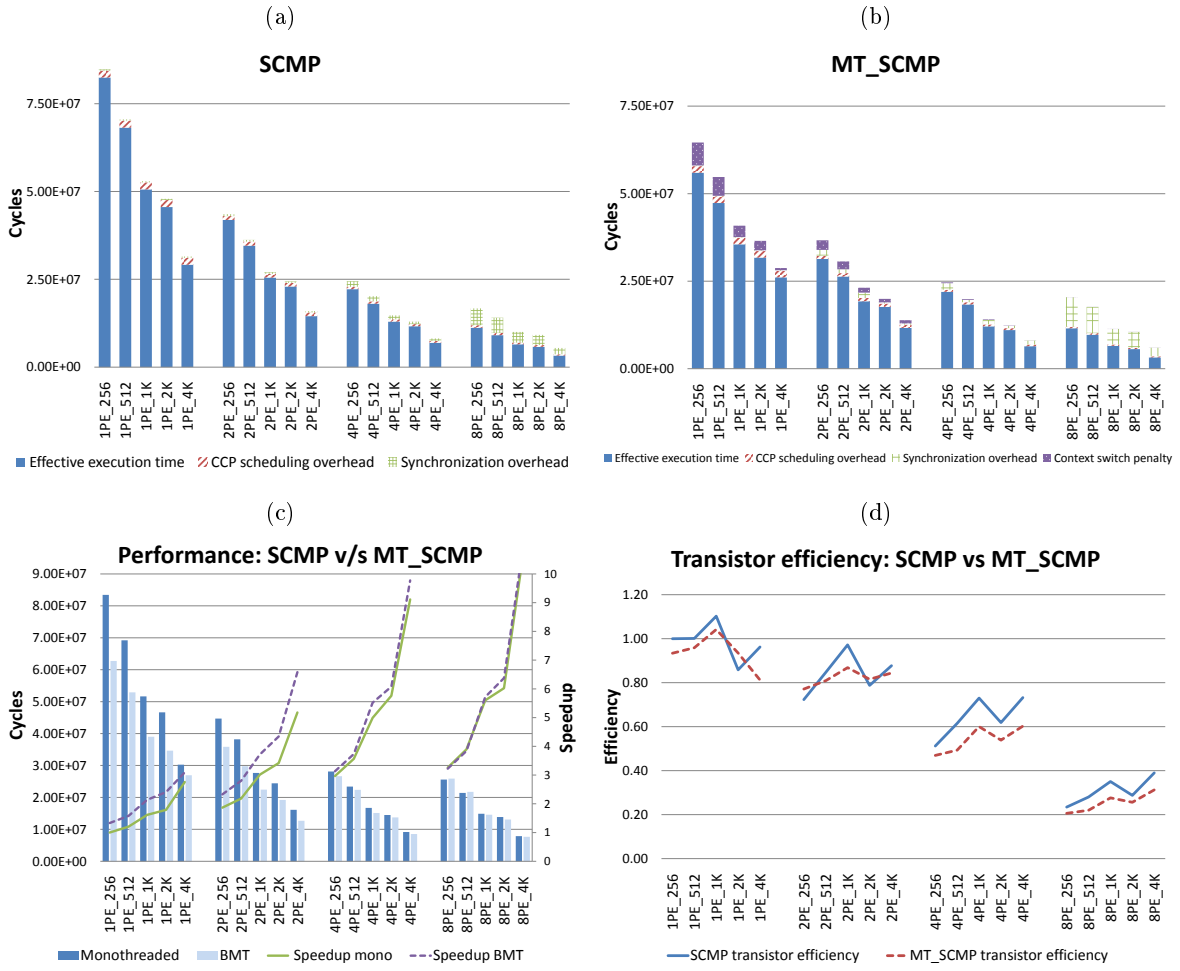


Figure 3.25: Performance of SCMP v/s MT_SCMP for streaming (WCDMA) application. The number of processors is varied from 1 to 8 and the cache sizes from 256-B to 4-KB. The performance is plotted in clock cycles a) SCMP system performance b) MT_SCMP system performance c) Comparison between SCMP and MT_SCMP performance and speedups d) Transistor efficiency of SCMP and MT_SCMP with respect to initial system: 1 PE and 256-B cache size.

The new SCMP architecture with multiple multithreaded processors is called MT_SCMP.

We conducted several benchmarks based on a control-flow and streaming applications in order to choose which multithreaded processor suits best for MT_SCMP (IMT v/s BMT), which global thread scheduling for multiple multithreaded processors gives the best performance (VSMP v/s SMTC), and which asymmetric MPSoC architecture is the most performant and transistor efficient (SCMP v/s MT_SCMP).

The results showed that the blocked multithreaded processor (BMT) and the SMTC scheduler suits best for MT_SCMP [18], and thus are adapted as fixed system design parameters for this architecture. Finally, we compared the performances and transistor efficiency of SCMP and MT_SCMP by running 2 types of applications: control-flow and streaming. In order to estimate both system surfaces, we used synthesis results in 40 nm TSMC technology for the processors and

the interconnection networks, and estimated the cache sizes using the CACTI 6.5 tool. To summarize the results, the MT_SCMP gave better peak performance, but less transistor efficiency than SCMP. In fact, the performance of MT_SCMP highly depends on 5 main parameters: application thread-level parallelism, caches miss rate, caches miss latency, memory hierarchy, and global thread scheduling. The latter implies that for dynamic applications, a dynamic load balancing and scheduling gives the optimal performance. This is why SCMP is a highly efficient architecture. Whether to choose multithreaded processors for SCMP or not, depends on the system designer. If peak performance is a key parameter, then multithreaded processors are an interesting solution. However, for transistor efficiency, monothreaded processors remain a more efficient solution.

The SCMP architecture is a transistor efficient architecture for dynamic embedded applications. However, for high-end massively-parallel dynamic embedded applications with large data sets, there are lot of parallelism at the thread level (TLP) and at the loop level (LLP) that should be exploited by the architecture. Thus, SCMP has the following limitations for such applications:

1. *Scalability:* SCMP is limited to 32 cores. In fact, the CCP is one source of resources contention and cannot handle efficiently more than 32 cores. In addition, the surface of the multibus network increases a lot (order of 10 mm2 in 40 nm TSMC technology) for more than 128 I/O ports [56], given that each core needs 3 I/O ports. Thus, SCMP does not meet the manycore requirements for embedded applications.

2. *Extensibility:* SCMP has a limited on-chip memory for data, which makes it not extensible for large data set applications.

3. *Programmability:* Data should be explicitly prefetched from the off-chip memory using dedicated DMA tasks prior to their utilization by computing tasks. The DMA tasks are identified and inserted offline in the application's CDFG. This lies a burden on the programming environment.

4. *Parallelism:* SCMP does not exploit parallelism at the loop level. Therefore, all the loop codes are executed sequentially on one PE.

In addition to those limitations, the SCMP architecture does not have enough stall latencies to be exploited by the multithreaded processors. In fact, SCMP has a dedicated central controller (CCP) that performs dynamic load-balancing whenever long latency stalls occur due to task synchronization. In this case, the task synchronization overhead will be the same for the monothreaded and multithreaded processor. Therefore, the only type of stall latencies that remain to be masked are the memory access latencies due to cache misses, which are very fast in SCMP architecture (less than 10 cycles).

Based on this conclusion, we will design in the next chapter a new manycore architecture that tackles the challenges of future high-end massively parallel dynamic applications called: AHDAM architecture.

Chapter 4

AHDAM: an Asymmetric Homogeneous with Dynamic Allocator Manycore architecture

Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius, and a lot of courage, to move in the opposite direction. – Albert Einstein, physicist

Contents

As we saw previously in chapter 1, a manycore architecture is a natural solution for future high-end massively parallel embedded applications. Those applications are becoming more dynamic with a variable execution time. Thus, an asymmetric homogeneous MPSoC architecture handles efficiently this dynamism by using a dedicated control core that performs dynamic load-balancing of the tasks between the processing resources.

In addition, we saw in chapter 3 that currently existing asymmetric homogoneous architectures, such as SCMP, are not scalable to the manycore level. Furthermore, the manycore architecture should support the processing of large data set sizes that cannot be known in advance and does not fit in the on-chip memory. This implies a frequent access to the off-chip memory that stalls the processors and degrades the performance. For this reason, multithreaded processors could be one

key element to increase the aggregate IPC of the chip with little die area overhead. Therefore, there is a need for new transistor and energy efficient manycore architectures that tackle the challenges of future massively-parallel dynamic embedded applications.

For all these requirements, we present a new manycore architecture called AHDAM [17]. AHDAM stands for *Asymmetric Homogeneous with Dynamic Allocator Manycore architecture*. It is used as an accelerator for massively parallel dynamic applications. In particular, it is designed to accelerate the execution of the loop codes, which often constitutes a large part of the overall application execution time.

In this chapter, we present in details the AHDAM architecture. First, we start by describing its applicative system environment in section 4.1 and its programming model in section 4.2. Then, we explain the overall architecture description and the motivations for each architectural choice in section 4.3. In particular, we motivate the design of the memory hierarchy architecture and the need for blocked multithreaded processors as key processing elements by conducting an analytical study. The functionalities and interoperabilites of the hardware components are described in details. Then, in section 4.4, we illustrate the AHDAM's execution model. One particularity is that the loops can be parallelized using OpenMP or any fork-join programming model, and then executed on special processing elements. Finally in section 4.5, we analyze the maximal scalability that the AHDAM architecture can reach by conduct a bandwidth analysis study.

4.1 System description

AHDAM is used as an accelerator component for high-end massively parallel dynamic embedded applications as shown in Figure 4.1.

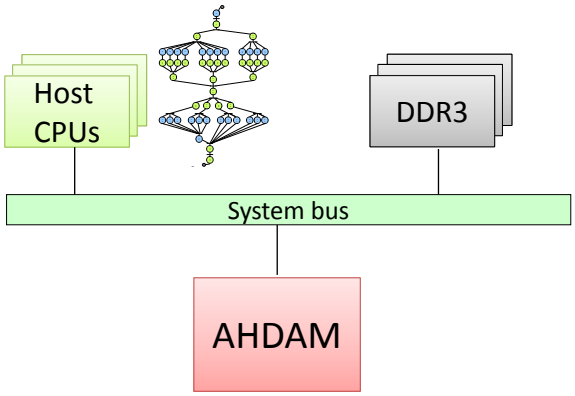


Figure 4.1: AHDAM system environment: multiple host CPUs offloading massively-parallel applications to the AHDAM architecture. The large data set is stored in multiple off-chip DDR3 memories.

Depending on the computation requirements, it can be used as a shared accelerator for multiple host CPUs, or a private accelerator for each host CPU. The host CPU is running an operating system or bare-metal applications. Typical applications are from the cloud computing, database, and networking domains. During their runtime, the applications' codes and data are stored in multiple DDR3 memory banks. Those applications are massively-parallel, hence they require lot of

computation power that is highly efficient to be processed by AHDAM architecture. When a host CPU encounters a massively-parallel application, it sends an execution demand to AHDAM and wait for its acknowledgment. Then, the host CPU offloads the massively-parallel application to AHDAM. The application is already decomposed into concurrent tasks. The tasks are represented in a CDFG graph that shows their control dependencies, hence their activation sequence. AHDAM has sufficient resources to process multithreaded tasks in parallel. In addition to the task level parallelism (TLP), it can increase the parallelism by exploiting the concurrency at the loop level (LLP). In fact, most of the application execution time is spent in loops.

In the next section, we will explore in more details the AHDAM programming model.

4.2 AHDAM programming model

The programming model for AHDAM architecture is specifically adapted to dynamic applications and global scheduling methods. It is based on a streaming programming model. The chip's asymmetry is tackled on 2 levels: a fine-grain level and a coarse-grain level. The proposed programming model is based on the explicit separation of the control and the computing parts as shown in Figure 4.2. A sequential application is manually cut into independent tasks from which explicit execution dependencies are extracted (TLP). Then, the parallel application follows a second path where OpenMP pragmas are inserted at the beginning of possibly parallelized 'for-loop' blocks (fine-grain). In fact, OpenMP [106] is a method of Single-Program-Multiple-Data (SPMD) parallelization, where each program contains one or more loop regions (LLP). The master thread forks a specified number of slave threads, and a task is divided among them. Then, the child threads run in parallel, with the runtime environment allocating threads to different cores. A possible solution to automate the application decomposition process and insertion of OpenMP pragmas is to use the *PAR4ALL* tool from HPC Project [108]. The *PAR4ALL* tool supports AHDAM HAL for proper tasks generation. At this stage, the computing tasks and the control task are extracted from the application, so as each task is a standalone program. The greater the number of independent and parallel tasks that are extracted, the more the application can be accelerated at runtime, and the application pipeline balanced.

The control task is a Control Data Flow Graph (CDFG) extracted from the application (Petri Net representation), which represents all control and data dependencies between the computing tasks (coarse-grain). The control task handles the computing task scheduling, activations, and other control functionalities, like synchronizations and shared resource management for instance. A specific and simple assembly language is used to describe this CDFG and must be manually written or automatically generated by the *PAR4ALL* tool. A specific compilation tool is used for the binary generation from the CDFG.

For the computing tasks, a specific Hardware Abstraction Layer (HAL) is provided to manage all memory accesses and local synchronizations, as well as dynamic memory allocation and management capabilities. A special on-chip unit called MCMU (Memory Configuration and Management Unit) is responsible for handling these functionalities (more details in section 4.3.1). With these functions, it is possible to carry out local data synchronizations or to let the control manager taking all control decisions. Concurrent tasks can share data buffers through local synchronizations handled by the MCMU (streaming execution model). Each task is defined by a task identifier, which is used to

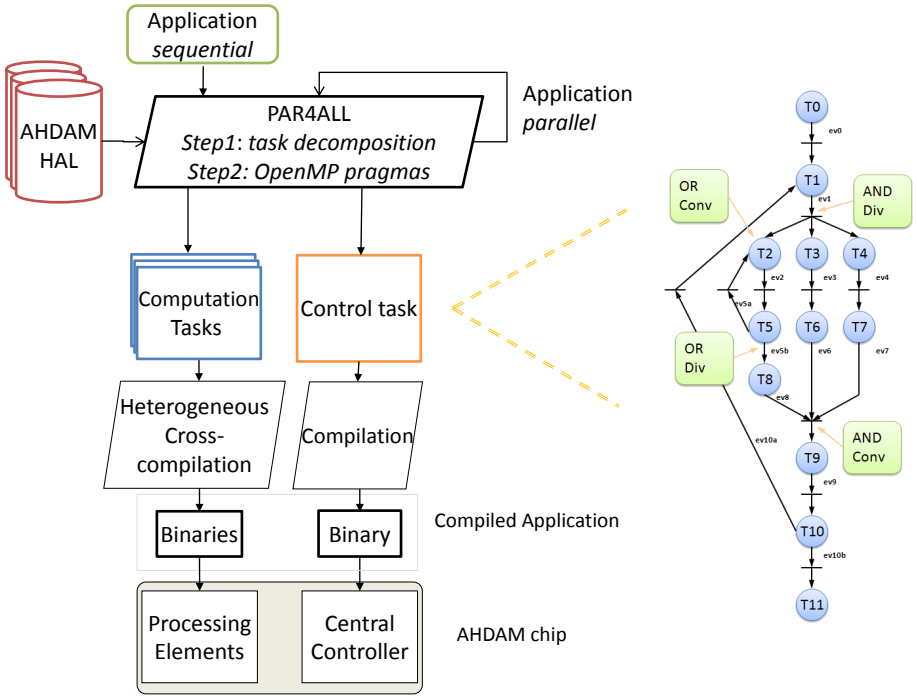


Figure 4.2: AHDAM programming model and an example of a typical CDFG control graph.

communicate between the control and the computing parts. A task suspends/resumes its execution based on data availability from other tasks. A data is allocated in a data buffer. It follows the streaming/dataflow execution model. When a data is produced by Task A, then Task B resumes its execution. When a data is consumed by Task B, then it suspends its execution. Each task has the possibility to dynamically allocate or deallocate buffers (or double buffers) in the shared memory space through specific HAL functions. An allocated buffer is released when a task asks for it and is the last consumer. A buffer cannot be released at the end of the execution of the owner task. A dynamic right management of buffers enables a dataflow execution between the tasks: it is handled by the MCMU.

Once each application and thread has been divided into independent tasks, the code is cross-compiled for each task. For heterogeneous computing resources, the generated code depends on the type of the execution core. In the next section, we will describe in details how the AHDAM architecture is designed.

4.3 AHDAM architecture design

AHDAM architecture is an improved version of the SCMP architecture. Our design choices are based on the benchmarking results conducted in chapter 3 on SCMP with multithreaded processors and the conclusion deduced in section 3.3.6. We identified the following limitations in the SCMP architecture:

1. *Scalability:* SCMP is limited to 32 cores. In fact, the CCP is one source of resources contention and cannot handle efficiently more than 32 cores. In addition, the surface of the multibus network increases a lot (order of 10 mm^2 in 40 nm TSMC technology) for more than 128 I/O ports [56], given that each core needs 3 I/O ports. Thus, SCMP does not meet the manycore requirements for embedded applications.

2. *Extensibility:* SCMP has a limited on-chip memory for data, which makes it not extensible for large data set applications.

3. *Programmability:* Data should be explicitly prefetched from the off-chip memory using dedicated DMA tasks prior to their utilization by computing tasks. The DMA tasks are identified and inserted offline in the application's CDFG. This lies a burden on the programming environment.

4. *Parallelism:* SCMP does not exploit parallelism at the loop level. Therefore, all the loop codes are executed sequentially on one PE.

In this section, we present the AHDAM architecture. First, we describe the architecture and its main components. Then, we present the memory architecture and we build an analytical model for the processor-memory system to compare different memory hierarchies and the processor type. We show that the split-memory hierarchy adapted in AHDAM is more performant than other systems. Afterward, we explain the control unit and Tile unit roles and their interoperabilities. Also, by using a modified analytical model, we show that the blocked multithreaded processor with 2 TCs boosts the performance over the monothreaded processor. Finally, we conduct a bandwidth analysis study on different parts of the architecture and show the maximum scalability that AHDAM can reach.

4.3.1 Architecture description

In fact, the architecture's asymmetry is tackled on 2 levels: a coarse-grain level and a fine-grain level. The coarse-grain level represents the concurrent tasks in the CDFG application graph as was shown in section 4.2, while the fine-grain level represents the tasks' parallelized loop-region codes. The AHDAM architecture is shown in Figure 4.3.

AHDAM architecture is composed of 3 main parts: memory units, control unit, and computation units. The AHDAM architecture is an enhancement to the SCMP architecture, thus they share some similar functionalities. For instance, the control unit, tasks prefetching and the MCMU are the same. The latter has some added functionalities that will be explained later in this section. As for the differences, AHDAM architecture has a different memory hierarchy and a different interpretation of the computation unit. In fact, the computation unit is represented as a *Tile* instead of a SCMP's *PE*. In addition, there are no DMA engines, since data load/store from external memory is implemented in the task code to easy the programmability. In the following sections, we will describe in more details the functionality of each unit.

4.3.1.1 Memory units

The AHDAM memory hierarchy is composed of separated L2 instruction memory and data cache memories.

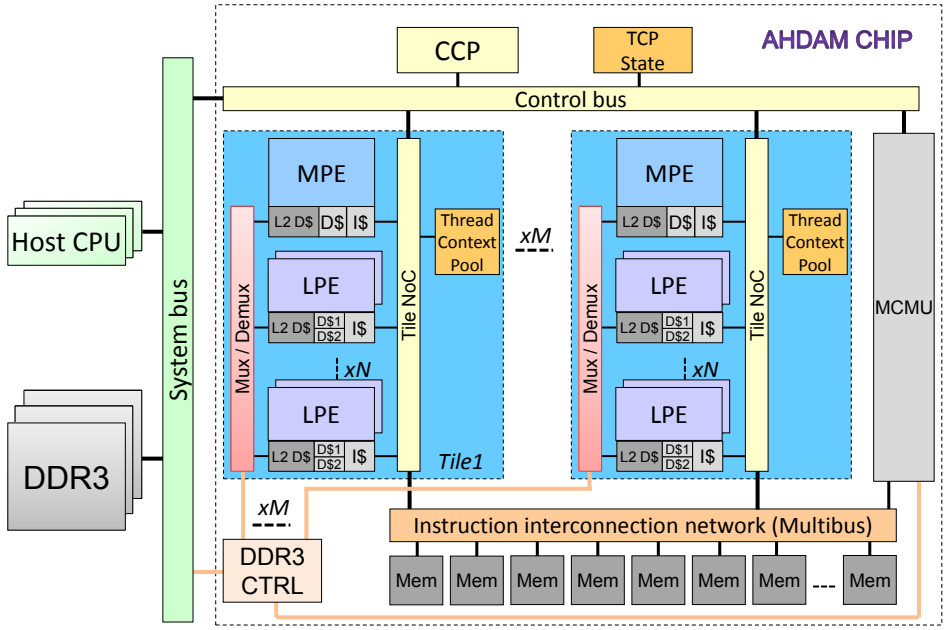


Figure 4.3: The AHDAM architecture.

The instruction memory is a shared on-chip multi-banked SRAM memory that stores the codes of the tasks. The instruction memory size can be well dimensioned since it is designed for the worst-case size of the different sets of application codes that will be running on AHDAM simultaneously. In addition, the instruction memory is a shared memory, which is suitable for inter-tile task migration and load-balancing. The shared on-chip instruction memory is the last level of instruction memory. As we will see later, the execution model assumes that the instructions are already prefetched in the instruction memory. Besides, it is implemented as a multi-banked memory instead of a single-banked multiple Read/Write ports memory. To compare the area occupation of both types of memories, we use the CACTI 6.5 tool [100] in 40 nm technology and we vary the number of memory banks v/s the number of R/W ports for a 1-MB memory. The area occupation of each configuration is shown in Figure 4.4. It is clear that a multi-banked memory with 1 R/W port uses less area than one-bank memory with multiple R/W ports.

The multi-bank memory reduces the contentions per bank when multiple Tiles are accessing simultaneously the instruction memory. This happens when the instruction codes for the tasks are stored in different memory banks.
The *Instruction interconnection network* connects the M Tiles to the multi-banked instruction memory. It is a **multibus**. According to the author [56], the multibus occupies less die area that other types of NoCs for small to medium interconnections, and has less energy consumption and memory access latency.

On the other hand, since we target applications with large data sets, we implement a L2 data cache memory instead of an on-chip SRAM memory as in SCMP. Cache memories have a bigger area and are less area/energy efficient than SRAM memories. But caches facilitates the programmability since the memory accesses to external DDR3 memory are transparent to the

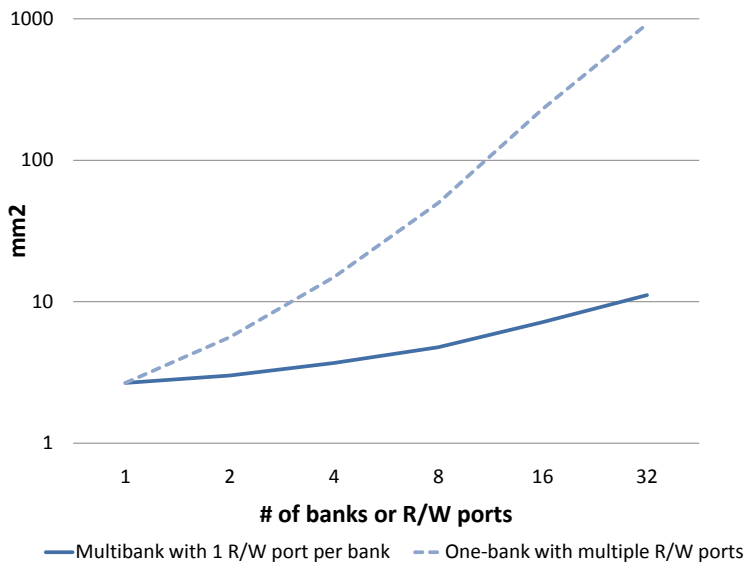


Figure 4.4: Estimated surface of multi-banked memories with 1 R/W port v/s one-bank memory with multiple R/W ports for a 1-MB memory. We vary the number of banks and R/W ports from 1 to 32. The tool used is CACTI 6.5 in 40 nm technology. The surface in mm^2 is plotted as log-scale.

programmer and independent from the data set size. This eliminates the need for explicit data prefetching using DMA engines which hardens the task decomposition and synchronization as it happens with the IBM CELL processor [120] for instance. All the L2 data cache memories are connected to an on-chip DDR controller, which transfers the data memory access requests to the off-chip DDR3 memories. More details will be provided in section 4.3.1.3.

A special unit called MCMU (Memory Configuration and Management Unit) handles the memory configuration for the tasks. It divides the memory into pages. In addition, MCMU is responsible of managing the tasks' creation and deletion of dynamic data buffers at runtime, and synchronizing their access with other tasks. There is one allocated memory space per data buffer. A data buffer identifier is used by tasks to address them. Each task has a write exclusive access to a data buffer. Since all the tasks have an exclusive access to data buffers, the data coherency problems are eliminated without the need for specific coherency mechanisms. A data buffer access request is a blocking demand, and another task can read the data buffer when the owner task releases its right. Multiple readers are possible even if the memory latency will increase with the number of simultaneous accesses.

4.3.1.2 Control unit

In AHDAM, the CCP (Central Controller Processor) controls the tasks prefetching and execution. It has similar properties to the SCMP's CCP as described in section 1.3.1. The CCP module is shown in Figure 1.6(b). The application CDFG is stored in dedicated internal memories. The CCP is a programmable solution that consists of an optimized processor for control called AntX (see section 2.2.1), which is a small RISC 5-stage, in-order, and scalar pipeline core. Thus, the RTOS

functionalities are implemented in software. The programmability feature of CCP allows us to implement a new and optimised scheduling algorithm in a small amount of time. In addition, the CCP has special interfaces for receiving/sending interruption demands to the computation units.

4.3.1.3 Computation units

The AHDAM architecture supports M Tiles. The CCP views a Tile as 1 computation unit. But actually, a Tile has one MPE and N LPEs. In addition, it has a 'special' scratchpad memory called *Thread Context Pool* that stores the thread contexts to be processed by the LPEs. The *Thread Context Pool* represents the tasks runqueue per Tile, thus AHDAM has M runqueues. Each runqueue can have one or more thread contexts, where each thread context holds the following information: start address, input arguments, parent thread identifier, child thread identifier, execution state, etc...The occupation status of all the Tiles' *Thread Context Pool* are updated in a special shared scratchpad memory unit called the *TCP state*. The *TCP state* is shared by all the Tiles. The MPE is the Master PE that receives the execution of a coarse-grain task or master thread from the CCP. It is implemented as a monothreaded processor with sufficient resources (ALUs, FPUs, etc...) for executing the tasks' serial regions. On the other hand, the LPE or Loop PE, is specialized in executing child threads that represent loop regions. The LPEs are implemented as blocked multithreaded VLIW processors with 2 hardware thread contexts (TC). In fact, the blocked multithreaded processor increases the LPE's utilization by masking the long access to the off-chip DDR3 memory that stalls the processors. In addition, the VLIW architecture [48, 115] is a transistor efficient solution that increases the LPE performance by exploiting the ILP of the loop tasks. Each MPE and LPE has a private L1 I$, L1 D$, and L2 D$. For the multithreaded LPE, the L1 I$ is shared by both TCs, while the L1 D$ is segmented per TC. In this way, we privilege the execution of 2 child threads from the same parent thread, while limiting their interferences on the data memory level. The *Tile NoC*, which is a set of multiple busses interconnecting all the units to each other and to the external world (control and memory busses), is responsible of forwarding the cores' request accesses to the corresponding external unit. However, for the memory data accesses, the requests are grouped by a special MUX/DEMUX unit that forwards the data request to the DDR controller, then to the off-chip DDR3 memory. The *Tile NoC* provides one serial connection of the Tile to the external world, which eases the implementation of the Control and Instruction busses.

In summary, AHDAM provides architectural solutions that give ideal conditions for massively parallel dynamic applications with OpenMP-like tasks such as:

- *Two-level asymmetries*: this tackles the dynamism of the applications on the task level and loop level. The former is handled by the CCP and the latter by the MPEs and LPEs. The CCP handles dynamically the load-balancing of the tasks between the Tiles. The LPEs implement a farming execution model to improve the scheduling process for irregular loops.
- *Manycore*: for providing sufficient processing elements to execute the massively parallel tasks.
- *Shared on-chip instruction memory*: for fast inter-tile task migration and load balancing.
- *Shared off-chip data memory*: for supporting large data set applications that cannot fit normally in the on-chip data memory.

- *Shared inter-tiles LPEs*: for handling variable tasks' computing requirements and increasing the overall resources occupation. It will be explained in more details in section 4.4.
- *Multithreaded LPEs*: for hiding the LPE pipeline stalls when accessing the off-chip DDR3 memory, hence increasing the pipeline utilization.

In the following sections, we motivate different architectural design choices. In particular, we construct an analytical model for the memory hierarchy. Two processing elements are considered: monothreaded and blocked multithreaded processors. We demonstrate why the L2 instruction and data memory are splitted, and the advantage of implementing the LPEs as blocked multithreaded processors.

4.3.2 Why splitting the L2 instruction and data memories?

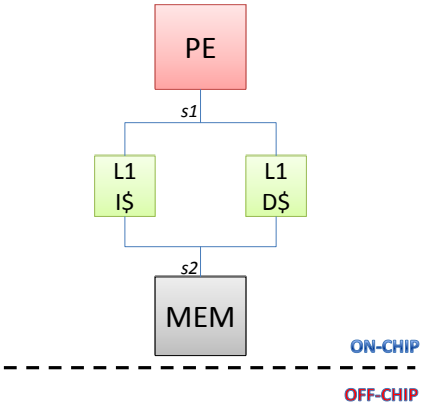
There is always a compromise for choosing the best memory architecture. In our case, the compromise is the on-chip memory size v/s the processor performance. A good architecture design is achieved when the processor is able to achieve a high performance with the least required on-chip memory. The processor's CPI is one key metric to measure its performance. Therefore, we build an analytical model that calculates the monothreaded processor's CPI for different memory hierarchy architectures. We model 4 different types of processor-memory architectures: SCMP with 1-level cache memory and on-chip main memory, SCMP with 1-level cache memory and off-chip main memory, SCMP with 2-levels cache memory and off-chip main memory, AHDAM with 1-level instruction cache memory and on-chip instruction main memory and 2-levels data cache memory and off-chip data main memory. The 4 processor-memory systems are shown in Figure 4.5. The processor that is considered in this study is monothreaded, scalar, in-order, with a 5-stage pipeline. But this study can also apply for a VLIW processor.

The CPI formula for a processor is given by equation 4.1:

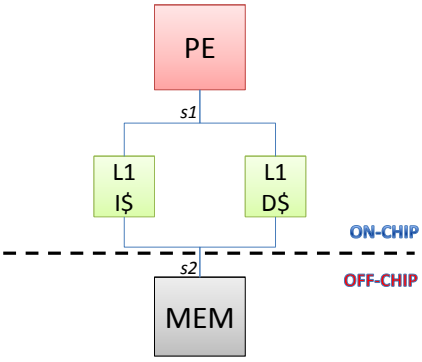
$$CPI = CPI_{\text{BASE}} + (1 + \%loads/stores) * StallCyclesPerMemoryAccess \tag{4.1}$$

The CPI formula measures the average clock cycles to execute an instruction. The formula states that the CPI is the summation of the base CPI and the total number of memory accesses from the IF-stage and MEM-stage multiplied by the average number of stall cycles for each memory access (*StallCyclesPerMemoryAccess*). The base CPI is the CPI with an ideal memory of zero cycle access. Thus, the base CPI captures only the pipeline data and control dependencies stalls between the instructions. This number depends on the application code. But for case of simplicity, we consider that almost 10% of the instructions exhibit pipeline stalls for a 5-stage pipeline. Thus, CPI_{BASE} is equal to *1.1*.

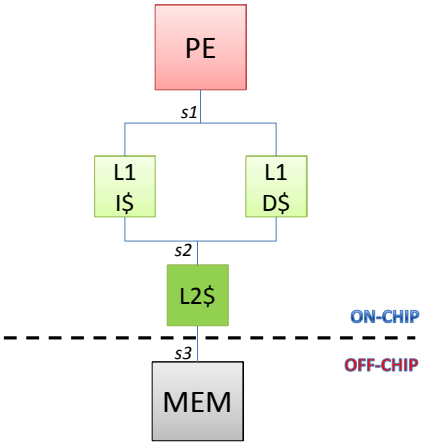
Now, the remaining parameter to compute the CPI is the *StallCyclesPerMemoryAccess*. This parameter depends on the processor type, the memory hierarchy, the cache memories miss rates, and the access time between each memory stages. Let us consider the processor-memory system of Figure 4.5(a). In Table 4.1, we show the reasoning for calculating the *StallCyclesPerMemoryAccess* parameter using the probability theory. For each memory unit, we note its memory access time cost and the probability of accessing this memory throughout the execution. Therefore, for the monothreaded processor, if the number of memory units is n, then the number of cases is also n.



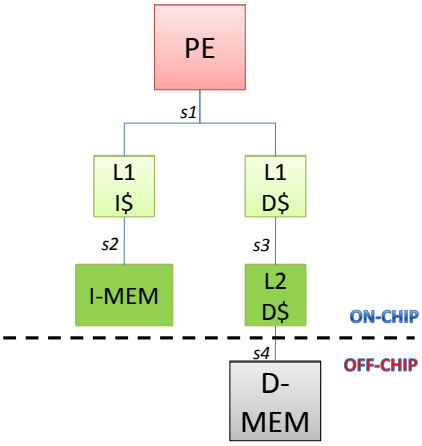
(a) Monothreaded PE with 1-level cache memory and on-chip main memory



(b) Monothreaded PE with 1-level cache memory and off-chip main memory



(c) Monothreaded PE with 2-levels cache memory and off-chip main memory



(d) Monothreaded PE with 1-level instruction cache memory and on-chip instruction main memory and 2-levels data cache memory and off-chip data main memory

Figure 4.5: 4 different processor-memory system architectures.

Then, *StallCyclesPerMemoryAccess* for the monothreaded processor is calculated by summing the cost and the probability of all the memory units as shown in equation 4.2:

$$StallCyclesPerMemoryAccess = \sum CostCycles * P(TC1) \tag{4.2}$$

The same reasoning applies for all the others processor-memory systems. Now, let us compare the CPI performance for the 4 systems. There are lot of architectural parameters that can be varied. But to facilitate the readability and the interpretation of the results, we will fix the following parameters:

- CPI_{BASE}: as mentioned earlier, the base CPI with ideal memory is equal to 1.1 for a scalar

TC1 in memory unit	Cost (cycles)	P(TC1)
L1 I$	s1	a1
L1 D$	s1	b1
MEM	s2	c1

a1 = %instructions x I$_L1_HitRate
b1 = %data x D$_L1_HitRate
c1 = %instructions x (1 - I$_L1_HitRate) + %data x (1 – D$_L1_HitRate)

Table 4.1: Probability table showing the stall cycles per memory access of level 1 memory hierarchy with seperate I$ and D$ for a monothreaded PE. Each memory unit in the first column has a penalty time in the second column. The third column shows the probability of TC1 to access the corresponding memory unit.

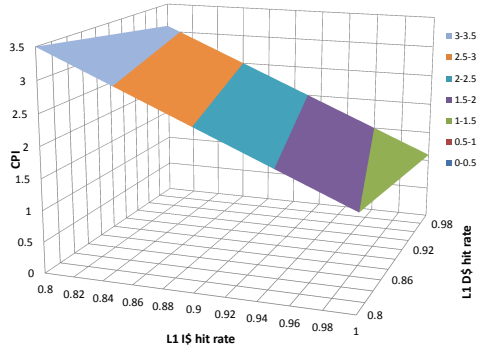
and 5-stage pipeline.

- *Percentage of load/store instructions*: we assume a rule of thumb that 20% of an application instructions are load/store instructions (%loads/stores). This implies that the fraction of instruction fetches out of all memory accesses (%instructions) and the fraction of data accesses out of all memory accesses (%data) are around 83% and 17% respectively.
- *Memory access time*: we consider that a L1 cache memory access time (s1) is included in the CPIBASE with no extra penalty, the access to the on-chip main memory and the L2 cache memory takes 10 cycles taking into consideration memory bank contentions, and the access to the off-chip main memory takes 50 cycles. Any small changes in the latency values do not alter our conclusions.
- *L2 cache memory hit rate*: the L2$ memory hit rate is fixed to 80%, which is a reasonable number for most of the systems.

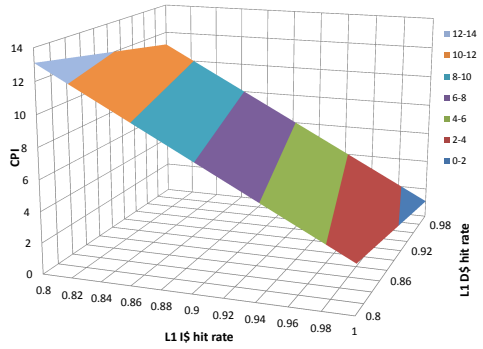
The only variable parameters are the L1 I$ and D$ hit rate, which vary from 80% to 100%.

In Figure 4.6, we show the CPI for all the 4 processor-memory systems, while varying the L1 I$ and D$ hit rate.

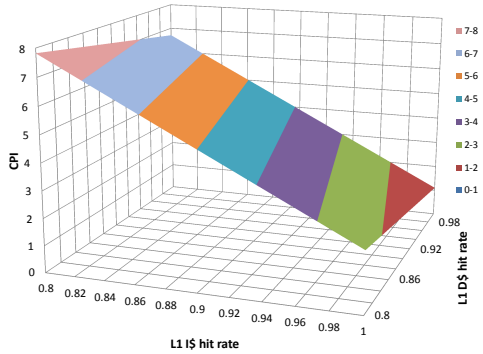
We observe that the processor performance in the SCMP base memory architecture (Figure 4.6(a)) with an on-chip main memory has the lowest CPI, hence the best performance. However, this implies that SCMP should have enough on-chip memory to store all the applications codes and data, which is not suitable for large-data set applications. In the second system (Figure 4.6(b)), the chip has a limited on-chip cache memory, and then it accesses the off-chip main memory whenever a cache miss occurs. In this case, the on-chip memory size is minimal but the processor might suffer from severe pipeline stalls due to the long off-chip memory access time. The CPI reaches 13 for the case of 80% L1 I$ and D$ hit rate. To limit the stall time penalty and increase the processor utilization, most manufacturers adapt a 2-levels on-chip cache memory. This model improves the processor's CPI as can be seen in Figure 4.6(c), but its performance is still lower than the base SCMP architecture. If we examine the key parameters that affect the overall performance, we observe that a typical application has 4/5 of instruction accesses and 1/5 of data accesses. The



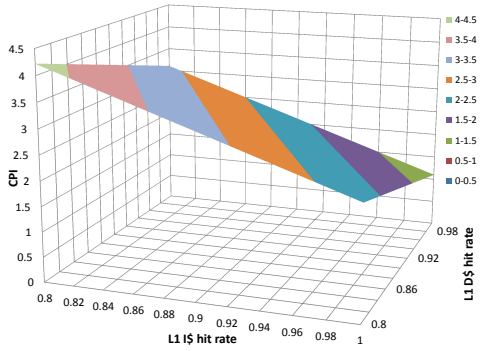
(a) Monothreaded PE CPI with 1-level cache memory and on-chip main memory (10 cycles)



(b) Monothreaded PE CPI with 1-level cache memory and off-chip main memory (50 cycles)



(c) Monothreaded PE CPI with 2-levels cache memory (L2 access=10 cycles) and off-chip main memory (50 cycles)



(d) Monothreaded PE CPI with 1-level instruction cache memory and on-chip instruction main memory (10 cycles) and 2-levels data cache memory (10 cycles) and off-chip data main memory (50 cycles)

Figure 4.6: CPI performance of 4 different processor-memory system architectures. A smaller CPI is better.

most design limiting factor is the variable data set size that varies a lot between the applications. Therefore, in AHDAM architecture, we integrate an on-chip instruction memory similar to that of SCMP and has fast access times. The data memory hierarchy is splitted from the instruction memory. Each processor has its private L1 D$ and L2 D$ on-chip memories, while the large data set is stored in an off-chip memory that is accesses during L2 D$ memory misses. As can be seen in Figure 4.6(d), this model improves the processor utilization and has almost the same performance as the base SCMP architecture. In addition, the on-chip memory size is almost similar to the third system (Figure 4.5(c)), since the instruction memory size does not occupy lot of space compared to the data memory. Therefore, AHDAM memory architecture has the best compromise compared to the other 3 processor-memory systems. Table 4.2 summarizes our discussion.

Processor-memory system	On-chip memory size	Processor performance
SCMP with 1-level cache memory and on-chip main memory	--	++
SCMP with 1-level cache memory and off-chip main memory	++	--
SCMP with 2-levels cache memory and off-chip main memory	+	+
AHDAM with 1-level instruction cache memory and on-chip instruction main memory and 2-levels data cache memory and off-chip data main memory	+	++

Table 4.2: Comparison between the 4 processor-memory systems. AHDAM memory hierarchy architecture has the best compromise between on-chip memory size and processor performance.

4.3.3 Why is the LPE a blocked multithreaded processor?

After we have defined AHDAM memory architecture in section 4.3.1, now we will examine the effect of replacing the monothreaded processor with a blocked multithreaded processor with 2 hardware TCs. For the BMT processor, the L1 instruction and data cache memories are segmented per TC as shown in Figure 4.7(a).

To build the analytical model for the BMT processor, we apply the same reasoning as for the monothreaded processor discussed in section 4.3.2. For case of simplicity, we will show the approach for the SCMP system with on-chip memory and with a BMT processor with 2 TCs (see Figure 4.7(b)).

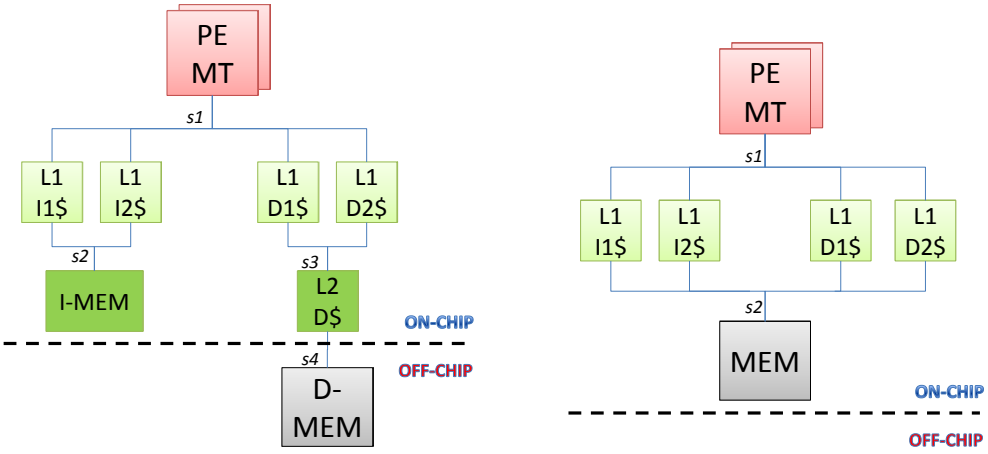
Table 4.3 shows the reasoning for calculating the *StallCyclesPerMemoryAccess* parameter for the BMT processor with 2 TCs (TC1 and TC2). Each TC can be in one of the memory units. So first, we consider all the possible combinations between the memory units of both TCs. Then, for each combination, we choose which TC is allowed to execute depending the scheduling protocol, and we note the cost in cycles and the probability of accessing each memory unit. For a BMT processor with m TCs and a memory system with n units, the total number of possible combinations is n^m.

Then, *StallCyclesPerMemoryAccess* for the BMT processor is calculated by summing the cost and the probability of all the memory units as shown in equation 4.3:

$$StallCyclesPerMemoryAccess = \sum CostCycles * P(TC1) * P(TC2) \tag{4.3}$$

We fix the same parameters (CPI_{BASE},Percentage of load/store instructions, Memory access time, L2 cache memory hit rate) as done previously in section 4.3.2, and we vary only the L1 I$ and D$ hit rate between 80% and 100%.

In Figure 4.8(a), we show the CPI for the AHDAM architecture (see Figure 4.7(a)) with a BMT processor and in Figure 4.8(b) we compare its speedup with respect to the same system with a monothreaded PE.



(a) Blocked multithreaded PE with 2 TCs and with 1-level segmented instruction cache memory and on-chip instruction main memory and 2-levels data cache memory and off-chip data main memory. The L1 D$ memory is segmented per TC

(b) Blocked multithreaded PE with 2 TCs and with 1-level segmented cache memory and on-chip main memory. The L1 I$ and D$ memories are segmented per TC

Figure 4.7: 2 different processor-memory system architectures with a BMT processor.

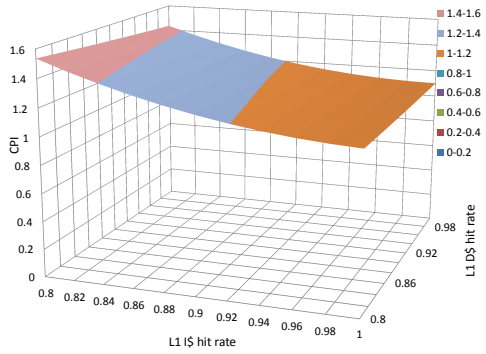
TC1 in memory unit	TC2 in memory unit	TC choice ?	Cost (cycles)	P(TC1)	P(TC2)
L1 I1$	L1 I2$	TC1	s1	a1	a2
L1 I1$	L1 D2$	TC1	s1	a1	b2
L1 I1$	MEM	TC1	s1	a1	c2
L1 D1$	L1 I2$	TC1	s1	b1	a2
L1 D1$	L1 D2$	TC1	s1	b1	b2
L1 D1$	MEM	TC1	s1	b1	c2
MEM	L1 I2$	TC2	s1	c1	a2
MEM	L1 D2$	TC2	s1	c1	b2
MEM	MEM	TC1	s2	c1	c2

a1 = %instructions1 x I1$_L1_HitRate
b1 = %data1 x D1$_L1_HitRate
c1 = %instructions1 x (1 – I1$_L1_HitRate) + %data1 x (1 – D1$_L1_HitRate)

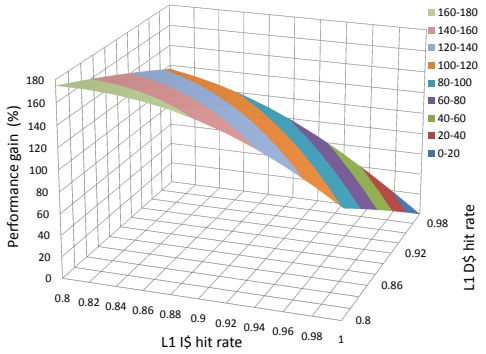
a2 = %instructions2 x I2$_L1_HitRate
b2 = %data2 x D2$_L1_HitRate
c2 = %instructions2 x (1 – I2$_L1_HitRate) + %data2 x (1 – D2$_L1_HitRate)

Table 4.3: Probability table showing the stall cycles per memory access of level 1 memory hierarchy with seperate and segmented I$ and D$ for a blocked multithreaded PE.

It is clear from these results that the BMT processor increases the pipeline utilization by executing another TC while there are long waiting stall cycles, hence a lower CPI. The BMT processor



(a) Blocked Multithreaded with 2 TCs' CPI with 1-level instruction cache memory and on-chip instruction main memory and 2-levels data cache memory and off-chip data main memory. The L1 I$ and D$ are segmented per TC



(b) Performance gain of BMT with respect to monothreaded PE for the AHDAM architecture

Figure 4.8: CPI performance of AHDAM with BMT processor compared to monothreaded processor.

with 2 TCs has a performance gain that reaches 175% (for 80% L1 cache hit rate) compared to the monothreaded PE for the AHDAM architecture. As the cache hit rate reaches 100%, we see in Figure 4.8(b) that there will be no performance gain. Therefore, there is a possibility to reduce the cache sizes, hence less on-chip memory die area and still guarantee a performance gain for the BMT processor. In summary, the LPE cores are BMT cores with 2 TCs instead of monothreaded cores in AHDAM.

4.4 Execution model

In this section, we describe a typical execution model sequence in the AHDAM architecture. At the beginning, AHDAM receives an application execution demand from an external host CPU through the *System bus*. The CCP handles the communication. It fetches the application task dependency graph (CDFG), stores it in its internal memory, and checks the next tasks ready to run to be pre-configured by the MCMU. When the MCMU receives a task pre-configuration demand from the CCP, it configures the shared instruction memory space and allocates the necessary free space, then it fetches the tasks instruction codes from the off-chip DDR3 memory using an internal DMA engine, and finally it creates internally the translation tables. At this stage, the CCP is ready to schedule and dispatch the next tasks to run on available computation units through the *Control bus*.

Each task has serial regions and parallel regions. The parallel regions can be the parallelized loop codes using a fork-join programming model such as OpenMP pragmas. For instance, let us consider the code example shown in Figure 4.9. It consists of 3 serial regions (S1,S2,S3) and 2 parallel regions (P1,P2). The thread execution is processed in 4 steps: 1) executing the serial region 2) forking the child threads 3) executing the child threads in parallel 4) joining the child

threads.

```
void main()
{
const int N = 100000, M = 5000000;
int i, a[N], b[M];
Initialize_vector(&a[0]);          //serial part S1
#pragma omp parallel for           //parallel part P1
for (i = 0; i < N; i++)             //irregular P1
    if ( (a[i] % i) == 1)
        a[i] = 2 * i;
    else if (a[i] % i) == 2)
        a[i] = 4 * i + 1;
    else
        a[i] = a[i] * i;
Initialize_vector(&b[0]);          //serial part S2
#pragma omp parallel for           //parallel part P2
for (i = 0; i < M; i++)             //regular P2
    b[i] = 4 * i + 5;
Print_vector(&a[0], &b[0]);        //serial part S3
return 0;
}
```

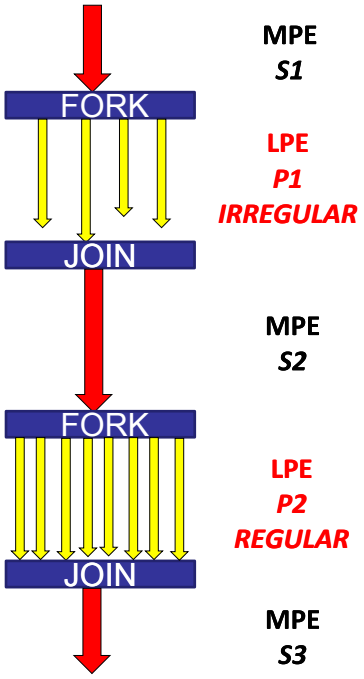


Figure 4.9: A task code example of serial and parallel regions using OpenMP pragmas (fork-join model) allocated by the CCP on a MPE.

Fork: The MPE executes the serial region of the task (S1). When it encounters a loop region using OpenMP pragmas (P1), the MPE executes a scheduling algorithm that uses a heuristic to fork the exact number of child threads in the appropriate Tiles' *Thread Context Pool*. The scheduling algorithm is part of a modified OpenMP runtime. The heuristic determines the maximum number of parallel child threads required to execute the loop as fast as possible based on: 1) the data set size 2) the number of cycles to execute one loop iteration 3) the Tiles' *Thread Context Pool* occupation using the shared *TCP State* memory 4) the cost of forking threads in the local and other Tiles. If possible, the algorithm favors the local *Thread Context Pool* since the fork and join process are done faster by avoiding the access to multiple busses. However, in some cases, the local *Thread Context Pool* is full or not sufficient while the ones in other Tiles are empty. Therefore, the local MPE has the possibility of forking the child threads in others *Thread Context Pool* by verifying their availability using the shared *TCP state* memory. This can be the case for the parallel region P2 in Figure 4.9. Finally, the MPE sends the number of forked child threads to the local *Thread Context Pool* in order to set a counter, then it goes into a 'dormant' mode to reduce the energy consumption.

Execute: Then, each LPE (Loop PE) TC executes one child task instance from the local *Thread Context Pool* until completion. Forked parallel child threads are executed in a *farming model* by the LPEs. As soon as a LPE is idle, it spins on the local *Thread Context Pool* semaphore trying to fetch another child thread context. This type of execution model reduces the thread scheduling time and

improves the LPEs occupation rate. In addition, it optimizes the execution of irregular for-loops. In fact, some for-loops have different execution paths that render their execution highly variable as shown in parallel region P1 in Figure 4.9. This scheduling architecture resembles the SMTC (Symmetric Multi-Thread Context) scheduling model with the MPE playing the role of the global controller. This has been shown to be the best scheduling architecture for multiple multithreaded processors [18] (see section 3.3.3.1).

In AHDAM, we implement a fork-join model with synchronous scheduling: the master thread forks the child threads, then waits to join until all the child threads have finished their execution. Therefore, during the execution of the parallel child threads, the MPE is in a dormant mode and is not preemptable by the CCP. There are 2 advantages from using this execution model: 1) the LPEs have a full bandwidth to the memory and are not disturbed by the MPE execution 2) easier 'join' process.

Join: When a child thread finishes execution, it sends a message to the corresponding *Thread Context Pool*, then flushes its L1 and L2 caches, which implement the write-back cache policy. Then, the *Thread Context Pool* decrements the counter that corresponds to the master thread. When the counter reaches zero, all the child threads have finished their execution and the master thread is ready to join. Thus, the *Thread Context Pool* preempts the MPE. The MPE leaves the dormant mode, flushes its L1 and L2 caches, and continues the execution of the serial region (S2).

4.5 Scalability analysis

AHDAM architecture is designed for the manycore era. But what is the maximum number of cores that can be integrated before experiencing performance drawbacks? To answer this question, we need to analyze each architectural component that might limit AHDAM's scalability.

AHDAM architecture is designed to be scalable horizontally (M Tiles) and vertically (N LPEs). The horizontal scalability is bounded by the control bus bandwidth, the CCP reactivity, and the DDR3 controller bandwidth plus its maximum supported number of master interfaces. On the other hand, the vertical scalability is bounded by the DDR3 controller bandwidth allocated for each Tile and the MUX/DEMUX component. Thus, the maximum number of Tiles and LPEs is bounded by the minimal number of these 4 parameters. In the following sections, we will explore each parameter in details.

4.5.1 Control bus dimensioning

The control bus is the interconnection network where the tasks' execution requests are sent from the CCP to all the Tiles' MPE as shown in Figure 4.10.

Let us assume that the MPEs and CCP have a frequency of 500 MHz (T=2ns) and that the control bus is operating on 250 MHz, that is half the processor's frequency. This implies that a 32-bit MPE (4 Bytes) generates a peak bandwidth of 2 GBps and a 32-bit bus supports 1 GBps peak bandwidth. However, the CCP-MPE communicates normally during the start and the end of a task. And, each communication packet is equal to 64 Bytes, which are the information needed to start the execution of a task. So, the overall communication bandwidth depends heavily on the task granularity. Let us vary the task granularity from 0.01 to 100 μs. The maximum number of MPEs that can be supported by a 32/64/128-bit control bus are shown in Figure 4.11:

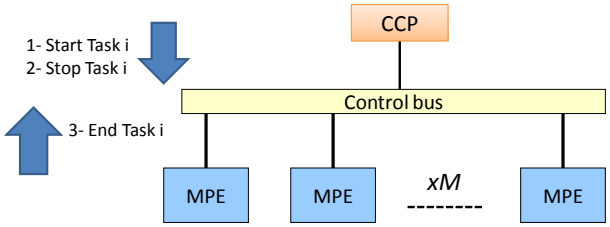


Figure 4.10: AHDAM control bus connecting the CCP with M MPEs.

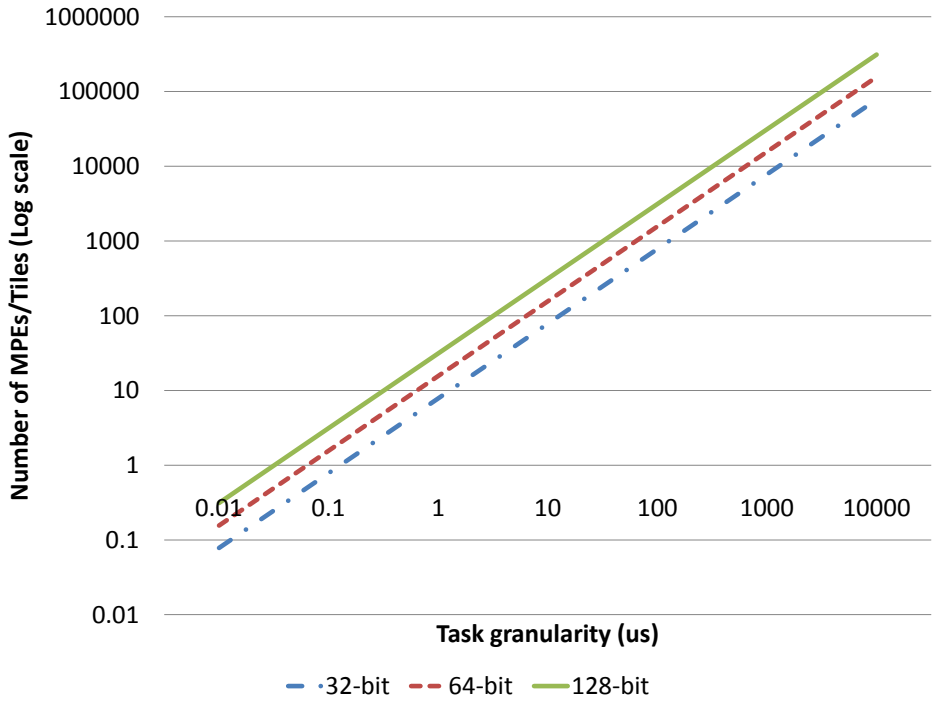


Figure 4.11: Total number of MPEs supported by a 32/64/128-bit control bus. The task granularity (x-axis) varies from 0.01 to 100 μs. The y-axis (log scale) shows the total number of MPEs that can be supported by the control bus.

These results show the optimal case (ideal bus arbiter) that renders the bus overdimensioned. In fact, there are more communications that happen through the control bus, such as MPE-MCMU communication. It is clear from this graph that the task granularity has a major impact on the control bus dimensioning, thus the maximum number of supported MPEs. It should be equal or greater to $5\ \mu s$ in order to support more than 100 MPEs.

4.5.2 CCP reactivity

As stated earlier in section 1.2.2.3, the main drawback of asymmetric architectures is their scalability and the centralized core's reactivity. The centralized control core suffers from contentions when

the number of computing cores and application tasks increases, hence the scheduling overhead of the central core also increases as shown in Figure 4.12. This means that the computing cores are stalled while waiting the scheduling decision of the control core.

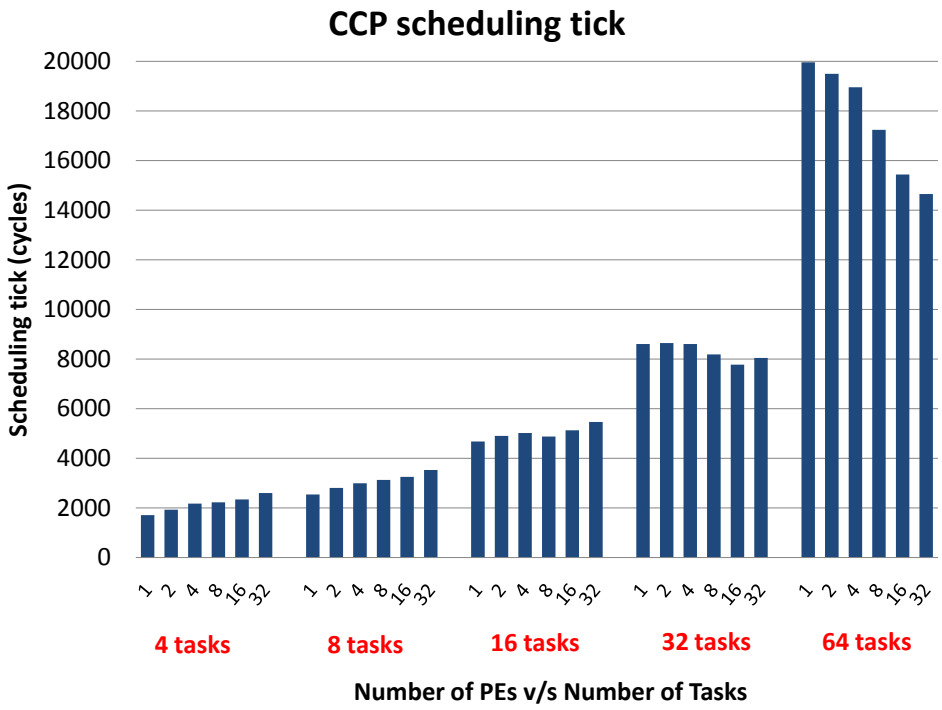


Figure 4.12: CCP scheduling tick length for variable number of PEs (1->32) and tasks (4->64). The application is the connection component labeling with variable levels of parallelism (4->64).

As we can notice, the main factor for a high scheduling overhead is the number of tasks to be scheduled and not the number of cores. Particularly for SCMP, the author [150] has shown that the tasks length should be equal or greater than *10* times the CCP scheduling tick length, in order to guarantee the cores' occupation rate to be greater than 85%. This means that for a CCP running at 500 MHz and an average CCP scheduling tick time of 10000 cycles, the scheduling time is equal to 20 μs. Thus, the minimal task length should be equal or greater to 200 μs (order of 10). Therefore, given the task length constraint and by matching it with the control bus bandwidth constraint in Figure 4.11, the control unit can support more than *1000 computing cores* before starting to suffer from performances degradation. In AHDAM, the CCP core is the same as in SCMP.

This implies that an asymmetric architecture with a central controller should support medium and coarse-grained tasks that have an execution time larger than the 10 times the CCP scheduling tick time. As for fine-grained tasks, they should have a different scheduling model, such as *farming*. In AHDAM, we support coarse-grained tasks on the CCP-MPE level for task-level parallelism and fine-grained tasks on the MPE-LPE level for loop-level parallelism.

In summary, the control bus bandwidth and the CCP reactivity are notlimiting factors for the number of supported Tiles, since the tasks implemented on the CCP level are coarse-grained tasks.

4.5.3 DDR3 controller

AHDAM has to exchange data with the outside world, mainly the DDR3 SDRAMs. To facilitate this communication, we integrate an on-chip DDR3 controller, where all the off-chip memory accesses should pass by. The Tile's memory accesses are serialized via the MUX/DEMUX unit, which is then connected to the DDR3 controller as a host port. For this study, we select the DesignWare Universal DDR Memory and Protocol Controller IP from Synopsys [137]. The DDR memory controller is shown in Figure 4.13.

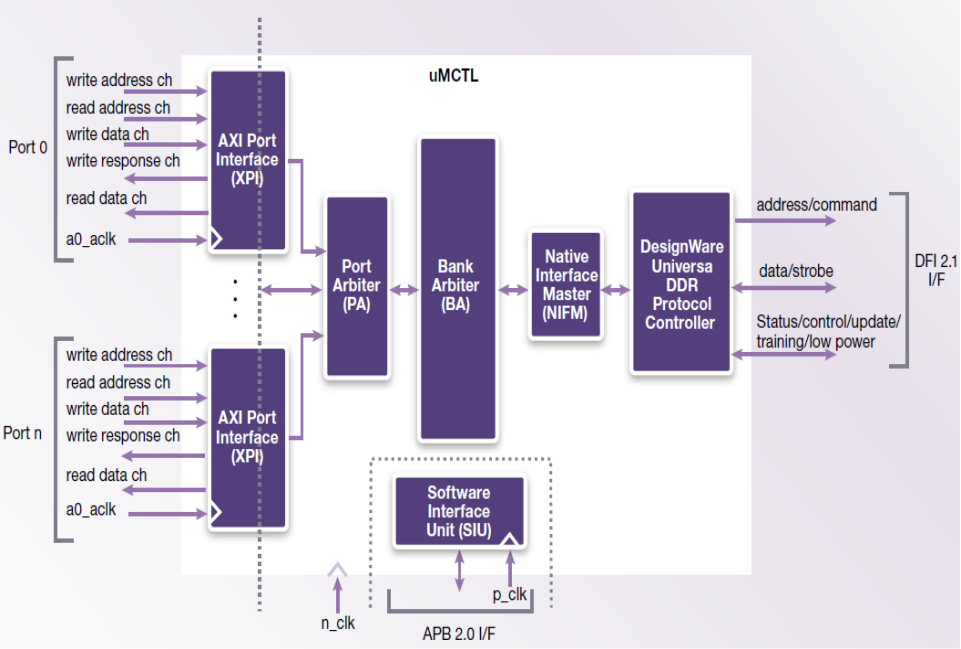


Figure 4.13: DesignWare Universal DDR Memory Controller Block Diagram (Synopsys courtesy).

The DesignWare Universal DDR controller family consists of two high performance components, the Universal DDR Protocol Controller (uPCTL) and Universal DDR Memory Controller (uMCTL). Both are capable of controlling JEDEC standard DDR2, DDR3, Mobile DDR and LPDDR2 SDRAMs. The uPCTL is a bridge between a system-on-chip (SoC) application bus and a PHY for a DDR SDRAM, such as the Synopsys DesignWare DDR PHYs (Physical interface). The uPCTL and the DDR PHY together handle the details of the DDR protocol, allowing the application to access memory via simple on-chip bus read/write requests. As for the uMCTL, it is a multi-port memory controller which accepts memory access requests from up to *32* application-side host ports. Therefore, the maximum number of supported Tiles is *32*. In addition, this DDR3 Controller supports data rates up to *2133 MegaTransfer per second* which is equivalent to *34.128 GBps* for a 128-bit data bus. This configuration is for a 533 MHz controller clock. Given this bandwidth, each Tile is allocated approximately *1066.5 MBps*. It is worth to note that some designers might share multiple Tiles per one host port at the expense of reducing the allocated bandwidth per Tile. This is a correct solution, but as we will see in section 4.5.4, it has a great impact on the vertical scalability of the architecture. Therefore, we privilege a simple system with one Tile per host port.

In summary, and based on the CCP reactivity, control bus, and DDR3 controller parameters, AHDAM architecture's maximum horizontal scalability is limited to *32 Tiles*. In the next section, we will see what is the maximum vertical scalability, in other words the maximum number of LPEs per Tile.

4.5.4 Vertical scalability

In this section, we will conduct a per Tile bandwidth analysis to know what is the maximum number of LPEs that can be integrated per Tile, or in other words the vertical scalability limitations. The MPE computation does not interfere with the LPEs, since we adapt a synchronous scheduling execution model as described later in section 4.4.

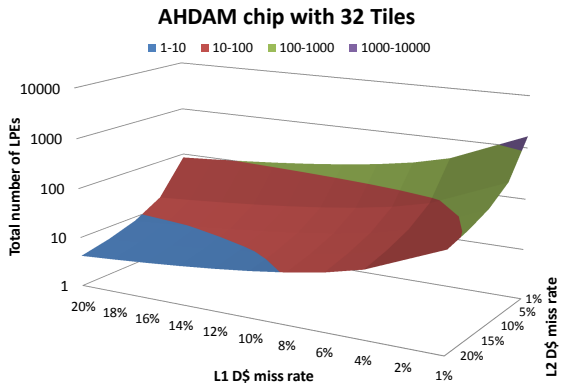
In AHDAM, the memory hierarchy is splitted between instruction and data. The instruction memory hierarchy does not constitute a source of bandwidth limitations, since the L1 I$ size can be over-dimensioned for the multiple classes of applications, thus it generates few cache misses. In addition, the instruction interconnection network is implemented as a multibus to limit simultaneous accesses to the same memory bank. So, the data memory hierachy (L1 D$, L2 D$, DDR3) is the source of vertical scalability limitations.

In our study, we assume the LPE is running at 500 MHz. Therefore, it generates a maximum of 500 MOPS, where each OPS (Operation Per Second) is a 32-bit operation. As a rule of thumb, 20% of these memory accesses are load/store instructions. Therefore, the L1 D$ receives 100 MOPS. Depending on the L1 D$ miss rate, the output throughput varies. For instance, let us assume the L1 D$ miss rate is 10% and the L1 block size is 16 Bytes. Thus, the output throughput that is input to the L2 D$ is equal to 160 MBps. The same reasoning applies for the L2 D$. We assume that the L2 D$ block size is equal to 64 Bytes. At the end, we divide the total allocated bandwidth per Tile by the LPE's L2 D$ output bandwidth, and we get the total number of LPEs that can be integrated in one Tile. The total allocated bandwidth per Tile depends on the number of Tiles. For instance, 32/16/8 Tiles have 1066.5/2133/4266 MBps respectively.

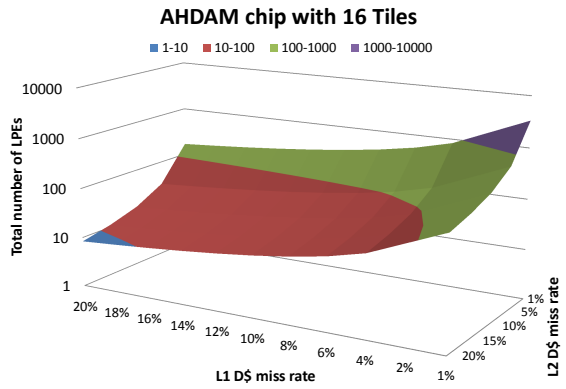
In Figure 4.14, we show the total number of LPEs for 32/16/8 Tiles. We vary the L1 D$ and L2 D$ miss rates between 20% and 1%.

The results show that the total number of LPEs heavily depend on the data cache miss rates. As a system engineer, we dimension the architecture for the worst-case scenario. In our case, we consider the worst-case to be for 20% miss rates for both data caches. Therefore, according to the bandwidth analysis, the maximum number of LPEs is 4/8/16 for 32/16/8 Tiles respectively. Of course, these numbers can be bigger if we are sure that the data cache miss rates will never exceed 20%.

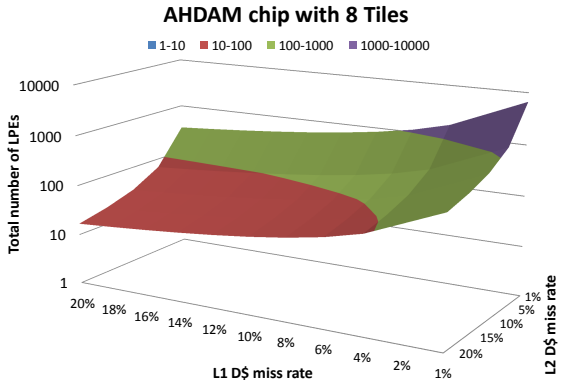
Finally, the MUX/DEMUX unit should guarantee the exact reserved bandwidth for each LPE. Therefore, it is designed as a Time-division multiplexing (TDM) bus. TDM is used for circuit mode communication with a fixed number of channels and constant bandwidth per channel. In a TDM bus, data or information arriving from an input line is put onto specific timeslots on a high-speed bus, where a recipient would listen to the bus and pick out only the signals for a certain timeslot. For case of simplicity, we implement a static TDM bus. However, a possible optimization is to implement a dynamic TDM bus, where the allocated bandwidth depends on the activity of each channel.



(a) 32 Tiles



(b) 16 Tiles



(c) 8 Tiles

Figure 4.14: Maximum number of LPEs per Tile for 32/16/8 Tiles. The x-axis shows the L1 D$ miss rate, the z-axis shows the L2 D$ miss rate, and the y-axis (log scale) shows the total number of LPEs that can be integrated per Tile.

4.6 Discussion

In this chapter, we presented an asymmetric manycore architecture, called AHDAM, that tackles the challenges of future high-end massively parallel dynamic embedded applications. We presented in details its programming model and the functionality of all its components. In addition, we studied the scalability of this architecture and we deduced that it can support 136 processors (8 Tiles x 16 LPEs + 8 MPEs) depending on the application requirement.

In the next chapter, we will evaluate the performance of AHDAM architecture using a real-case application from the telecommunication domain: radio-sensing. In addition, we will compare it to the SCMP architecture.

Chapter 5

Evaluation

Design is not just what it looks like and feels like. Design is how it works. – Steve Jobs, CEO Apple

Contents

We presented in chapter 4 the AHDAM architecture, which is a novel asymmetric manycore architecture for future high-end massively parallel dynamic embedded applications. It has a centralized control core that performs dynamic load-balancing of the coarse-grained tasks (TLP) between multiple Tiles (up to 32). Each coarse-grain task contains loop regions that are parallelized using OpenMP pragmas (fine-grain task) and executed in parallel on multiple dedicated processors. Thus, AHDAM architecture exploits the parallelism at 2 levels: TLP and LLP.

In this chapter, we evaluate the performance and transistor efficiency of AHDAM architecture by using a relevant embedded application. First, in section 5.1, we describe an application from the telecommunication domain called *radio-sensing*. This application has lots of computation requirements, lots of parallelism at the thread and loop levels, a large data set, and is dynamic. The radio-sensing application is parallelized and ported using AHDAM programming model flow. Then, in section 5.2, we show the simulation environment used to model and evaluate the AHDAM functionalities. And finally in section 5.3, we evaluate the transistor efficiency of the architecture and the importance of an asymmetric architecture. We evaluate its performance by running the radio-sensing application on different chip configurations and we compare its performance with respect to the SCMP architecture and a monoprocessor solution. At the end, we estimate the overall chip area in a 40 nm technology for multiple chip configurations.

5.1 An embedded application: Radio-sensing

The radio spectrum sensing application belongs to the telecommunication domain. This component, which can be found in cognitive radios, is developed by Thales Communications France (TCF) within the SCALOPES project.

We describe the application characteristics (purpose, features, scalability and parallelization opportunities) in section 5.1.1. Then, in section 5.1.2, we show how the task parallelism is extracted from the application using *PAR4ALL* tool.

5.1.1 Application description

The Spectrum Sensing is one of the main functions constituting a cognitive radio. A cognitive radio is a system characterized by the ability of a terminal to interact with its environment. It means that this terminal will have skills to sense its surrounding environment (sensing), to decide (cognitive manager) and to reconfigure (software radio) itself. For instance it will be able to detect the available frequencies and use them. Within this thesis, the application case that we will develop and study is more precisely the spectrum sensing step, which occurs at the physical layer as depicted in Figure 5.1:

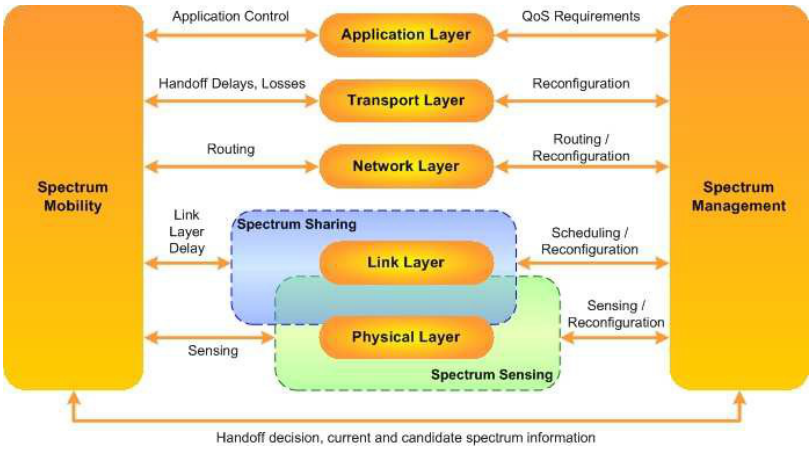


Figure 5.1: Spectrum Sensing description (source EDA/CORSAMA consulting).

The spectrum sensing function [70] aims to detect the unused spectrums and to share them without interference with other users. In other words, the already used spectrums are detected in order to identify spectrum holes. This application is used in the spectrum monitoring devices and the electronic warfare devices. Usually, spectrum monitoring focuses at signal modulation parameters and amplitude and does not require real time constraints. Electronic Warfare searches for real-time output face to regular communication signal or signals available within an identification data base. The sensing function faces a number of challenges in terms of miniaturization, power consumption, and timing response. These constraints are even more severe for mobile terminals. Three main sensing techniques can be used within the scope of spectrum monitoring and sensing: Cooperative context (Data-aided techniques), Blind Context, Semi-Blind Context. In this application, we will

limit the use case to a GSM sensing application (cooperative context).

The GSM sensing is composed of 3 main steps:

1. Digital signal pre-processing consisting in signal wideband filtering and in baseband transposition as well as in the signal channelization.
2. Processing of "each" channelized digital signal consisting in the parameter estimation and in the measurement at signal amplitude bandwidth modulation parameters.
3. Processing of "each" symbol consisting in the demodulation of the signal and in the measurement of the symbol stream code parameter.

The algorithm performing these 3 steps is decomposed into 8 operations. In this use case, we will limit the scope of the GSM sensing application use case to the first three main operations, which constitute the front-end processing. This use case is characterized by a sequential processing made of basic and unoptimized operations (no feed-back, no FFT). It consists in a strong data flow and thus requires strong memory and computational components that can be shared between the operations. It also implies a management of the dynamic of the signal within the processing. Figure 5.2 tries to synthesize these three steps, their flow sizes and related complexities (for GSM wideband input signal at 40 MHz):

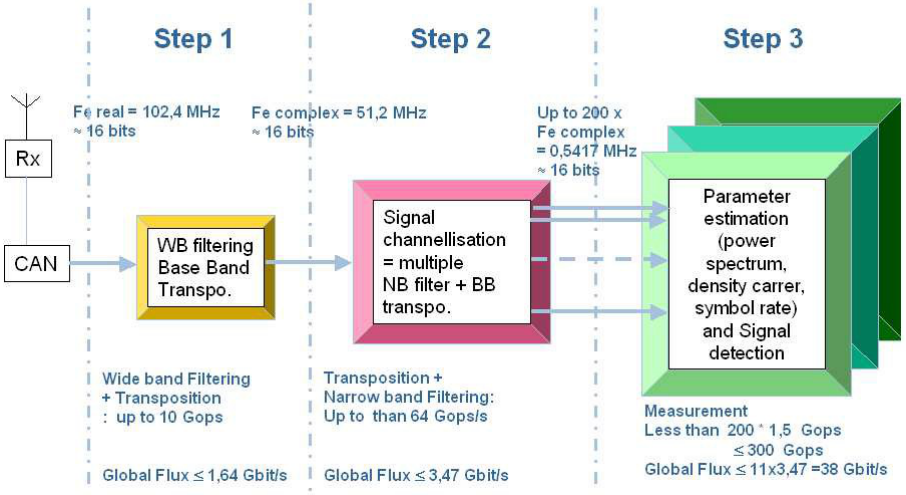


Figure 5.2: Spectrum Sensing main steps, maximal flows and complexities.

This use case contains various parameters that can be modified. As consequence, it leads to scalability opportunities. Modifying them will impact the number of operations (complexity) as well as the flow size and thus will induce different resources (memory, processing units) usage. One of the first obvious parameters on which we can have an influence in this application is the frequency of the wideband input signal. Another parameter is the size of the buffers that varies with respect to the duration length of the stored data. Indeed, all algorithms can be performed with limited duration buffers. For instance, we can select buffers sizes of 100 ms each second. As a consequence, it will limit the complexity and data flow size.

5.1.2 Task decomposition and parallelism

Initially, the radio sensing application is built to run sequentially on a monothreaded processor. The task level parallelism is explicitly expressed by inserting specific pragmas. Then, *PAR4ALL* cuts the application in a set of tasks according to these pragmas, generates communication primitives to implement a double buffer streaming processing, and the corresponding CDFG control graph as shown in Figure 5.3. *PAR4ALL* identified **30 tasks** that can run independently (TLP). Once independent tasks are generated, *PAR4ALL* identifies netloops and inserts OpenMP pragmas.

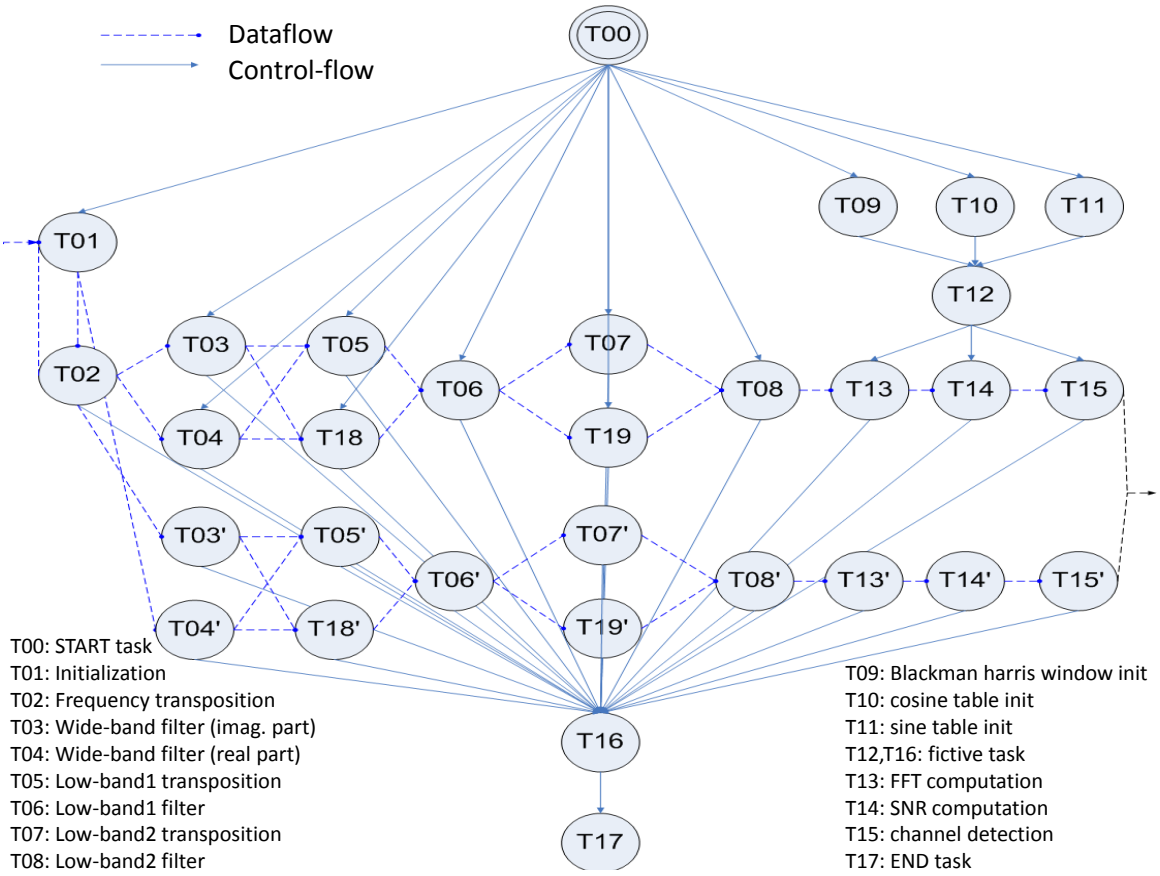


Figure 5.3: CDFG of the radio-sensing application.

The loop parallelism (LLP) is detected at runtime depending on the resources occupation. Also some loops are irregular, which means a variable execution time between the child threads. We profiled the application and we examined the hot spots in the code where most of the application time is spent. We noticed that 99.8% and 95% of the loop regions can be parallelized by OpenMP for the high-sensitivity and low-sensitivity respectively.

In addition, the application execution time varies with respect to the processed input data, hence its *dynamism*. In a real-case scenario, the application might be adaptively reconfigured to different execution modes, which implies different computation requirements.

In particular, the application could be launched in two different modes in order to fit different

user QoS. As a result, we defined two execution modes:

- **high sensitivity**: or high accuracy with a buffer of 100 ms every 1 second, 6 buffers, and a sampling frequency of 102.4 MHz. This gives us a computation requirement of *75.8 GOPS*, a data set of *432 MB*, and a real-time deadline of *6 seconds*.
- **low sensitivity**: or low accuracy with a buffer of 1 ms every 1 second, 6 buffers, and a sampling frequency of 25.6 MHz. This gives us a computation requirement of *328 MOPS*, a data set of *1,025 MB*, and a real-time deadline of *6 seconds*.

The radio-sensing application needs *1.5 MB* of instruction memory for storing all the task codes and stack memories. In a real case scenario, the application implements an adaptive reconfiguration to adjust the input parameters according to the requirements. Thus, it gives an execution behavior that is highly dynamic and with high/variable computation requirements.

In the next section, we will explain the simulation environment for the AHDAM architecture.

5.2 Simulation environment

AHDAM is a complex architecture that has several components with a special execution behavior, such as the Thread Context Pool memory and the multithreaded 3-way VLIW LPE for instance. In addition, the fork-join process inside each Tile necessitates a new and optimized runtime. We do not have yet a complete simulator for the AHDAM architecture. However, by using a combination of currently existing simulators such as SESAM and Trimaran [89, 144], and an analytical model for the AHDAM memory hierarchy architecture, we are able to estimate the AHDAM performance.

The simulation process of AHDAM architecture consists of simulating the serial regions on SESAM and the parallel regions on Trimaran. The performance gain due to multithreading is estimated using the analytical model for the AHDAM memory hierarchy with blocked multithreaded processors. Let us describe the step by step simulation process of AHDAM architecture for the radio-sensing application:

First of all, the radio-sensing application runs on the SESAM simulator that models the SCMP architecture with functional monothreaded MIPS32 ISSes. The monothreaded MIPS32 models the MPE processor. To evaluate the serial regions, we comment out the for-loops with OpenMP pragmas regions for all the tasks, or what we call a 'kernel', from the source code. Thus, the total execution time of each task is the exection time of the serial regions. The serial regions of the tasks are preemptive and can be migrated to other MPEs for dynamic load-balancing.

For the parallel regions, they should be executed on the LPE, which is a 3-way VLIW architecture. In SESAM, we do not have an ISS for a VLIW processor. This is why we use the Trimaran simulator 4.0 [89, 144] to estimate their performance. Trimaran is an integrated compiler and simulation infrastructure for research in computer architecture and compiler optimizations. Trimaran is highly parameterizable, and can target a wide range of architectures that embody embedded processors, high-end VLIW processors, and multi-clustered architectures. Trimaran also facilitates the exploration of the architecture design space, and is well suited for the automatic synthesis of programmable application specific architectures. It allows for customization of all aspects of an architecture, including the datapath, control path, instruction set, interconnect, and instruction/-data memory subsystems [144]. In our case, we use Trimaran to evaluate the VLIW processor.

Trimaran consists of three components as shown in Figure 5.4: the OpenIMPACT compiler, the Elcor compiler, and the Simu simulator.

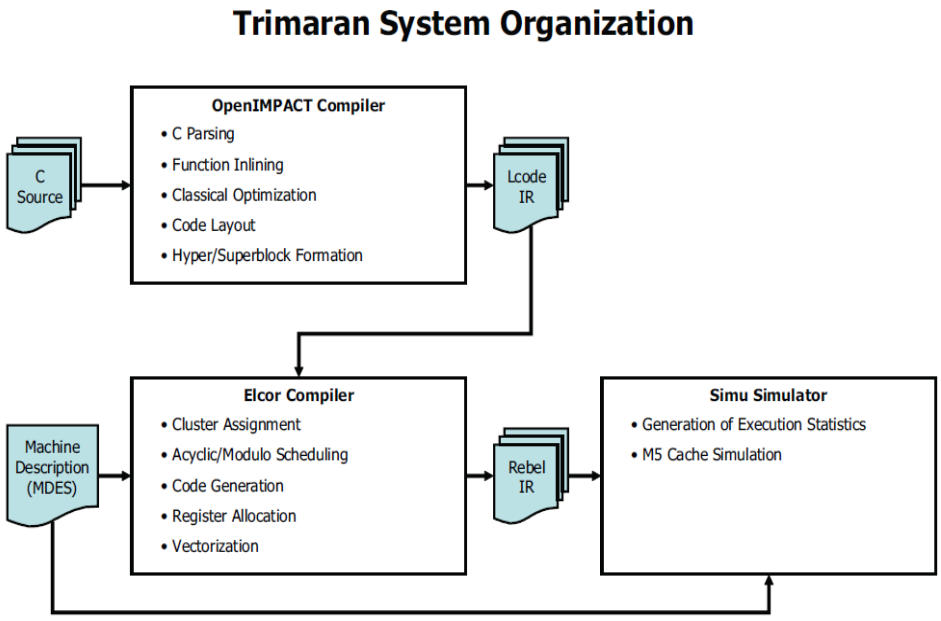


Figure 5.4: Trimaran organization: Trimaran uses OpenIMPACT to compile the original source code into an assembly intermediate representation (IR) called Lcode. The produced Lcode is optimized for ILP, but not for a specific machine. This code is then passed to the Elcor compiler, which is the VLIW compiler, along with a machine description (MDES) that specifies the target machine. Elcor compiles the code for the target machine, producing another IR called rebel. The Trimaran simulator known as Simu consumes the rebel code, executes the code, and gathers execution statistics.

The VLIW architecture has several configuration parameters. To simulate the VLIW in the AHDAM architecture context, we configured it to have a 3-way architecture with 2 ALUs, 1 FPU, a branch unit and 32 registers. The memory configuration reflects the AHDAM memory hierarchy as seen by the LPE: each LPE has a private L1 I$ and D$, and a private L2 D$. Then, the L1 I$ is connected to an on-chip instruction memory with 10 cycles of access time, and the L2 D$ is connected to the off-chip DDR3 memory with access time of 50 cycles. The modeled architecture in Trimaran is shown in Figure 5.5.

Each task of the radio-sensing application is executed 2 times in the Trimaran simulator: the first time with the serial and the parallel regions of the task and the second time with only the serial regions. The difference between both execution times gives the total execution time of the parallel regions on a 3-way VLIW processor. We consider that the parallel regions are non-preemptable, which means that once they are allocated on a LPE, they should run until completion.

The VLIW processor implemented in Trimaran is monothreaded. So, in order to estimate the performance gain of a blocked multithreaded processor, we use our analytical model for BMT in AHDAM memory hierarchy that was described in section 4.3.3. We estimate the performance increase due to a BMT multithreaded processor with 2 TCs. We assume that each LPE is executing 2 child threads from the same parent task. The cache miss rates statistics are extracted from the

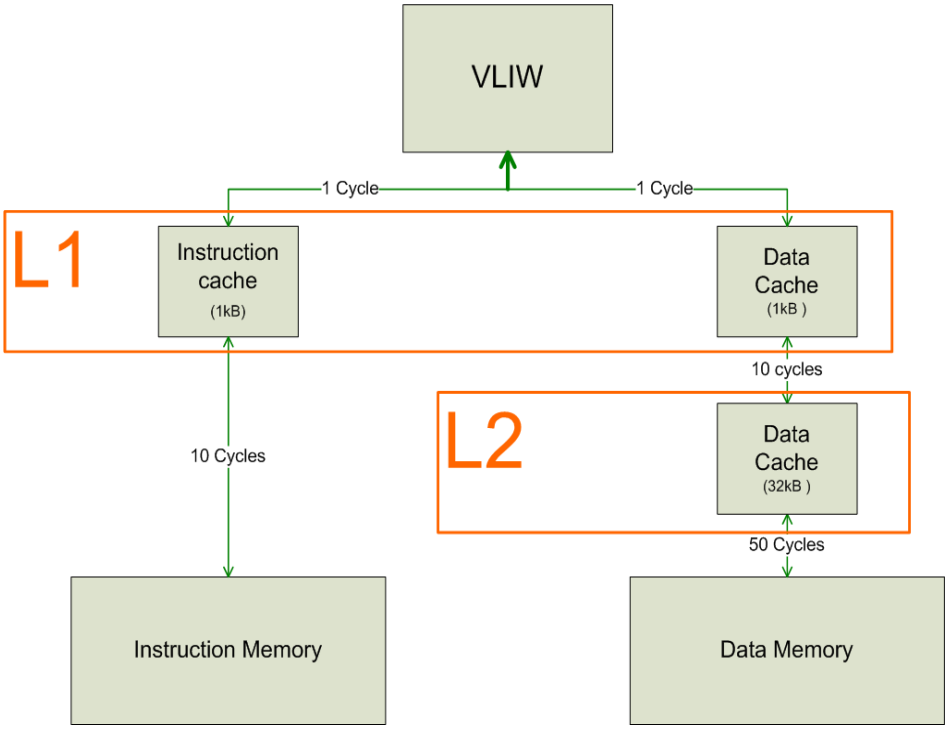


Figure 5.5: Trimaran configuration for modeling the LPE with AHDAM memory hierarchy.

Trimaran simulator.

Then, we inject the new performance results in SESAM simulator and we get the total execution time of all the tasks, where each task has its serial regions executed on a monothreaded MIPS32 processor and its parallel regions executed on a 3-way blocked multithreaded VLIW processor.

Our model assumes that the penalty due to forking and joining threads is negligible compared to the execution time of each thread.

We should note that the Trimaran simulator takes advantage of the host processor resources. For instance, the execution of the trigonometric operations such as sine/cosine are done very fast (fewer than 10 cycles), since they use the host CPU instructions and are executed directly on the microprocessor hardware (if the host CPU supports it). This is not the case of the MIPS simulator. Each sine/cosine function is emulated in software using a polynomial algorithm that takes more than 100 cycles depending on the input data. Therefore, to have a fair comparison between the MPE and LPE execution, we implement these functions in Trimaran as polynomial functions extracted from the generic libc.

5.3 Performance evaluation

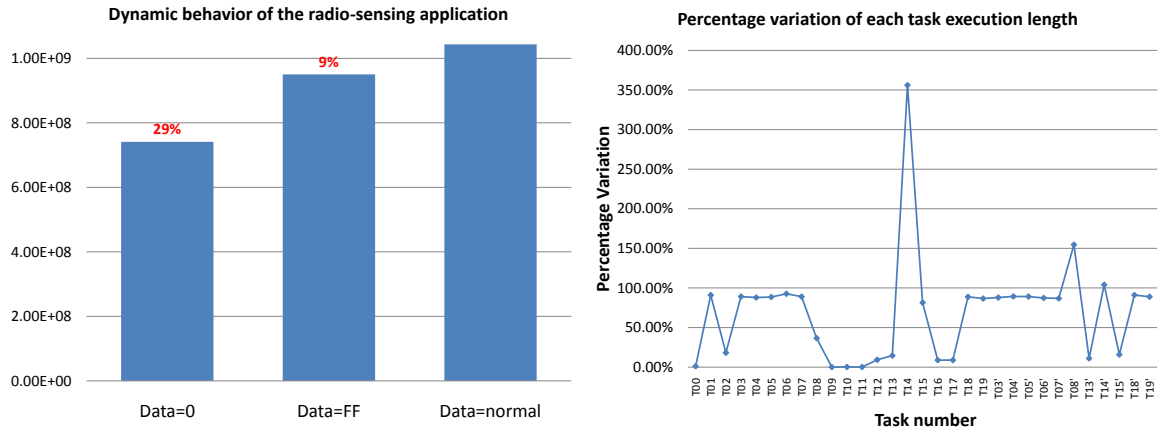
In this section, we will evaluate the performance and transistor efficiency of AHDAM architecture by running the radio-sensing application in 2 modes (see section 5.1): low-sensitivity and high-sensitivity.

The AHDAM architecture is configured with 8 Tiles and 4/8/16 LPEs per Tile. All the cores run at 500 MHz, thus the peak performance of AHDAM is *52 GOPS*, *100 GOPS*, and *196 GOPS* respectively for the 3 configurations. The MPE has a 4-KB L1 I$ and an 8-KB L1 D$, while the LPE has a 1-KB L1 I$ and a 2-KB L1 D$ (1-KB per TC), and a 32-KB L2 D$. The on-chip instruction memory is equal to 1.5 MB, which is the total size of the tasks' instructions and stack memories. The necessary cache memory values are deduced from the profiling results that are conducted on each parallel task in the radio-sensing application. The memory access to the on-chip instruction memory takes 10 cycles, as well as the L2 D$. For the off-chip DDR3 memory, the access time is 50 cycles.

First, we motivate the need for an asymmetric architecture for the radio-sensing application. Then, we evaluate the performance of AHDAM architecture with multithreaded and monothreaded LPEs, in order to see how the multithreading can boost the performance of AHDAM. In addition, we compare the performance of AHDAM architecture with SCMP architecture and with a monothreaded processor. Finally, we estimate AHDAM's area occupation in a 40 nm technology for 8 Tiles and 4/8/16 LPEs per Tile, and we evaluate the transistor efficiency of AHDAM architecture.

5.3.1 Why an asymmetric architecture?

As mentioned earlier, the radio-sensing application is a dynamic application as shown in Figure 5.6 for the low-sensitivity version executed on SCMP with 8 processors.



(a) Dynamic execution of the radio-sensing application

(b) Average percentage variation of each task execution length

Figure 5.6: Dynamic behavior of the radio-sensing application a) Total application execution time while varying the input data b) Average percentage variation of each task execution length.

In particular, we vary the input data and we plot the total execution time of the application in Figure 5.6(a). There is a variation of 29% between the normal input data and data with all zeros. On the other hand, when plotting the average percentage of variation of each task execution time in Figure 5.6(b), we notice that there is a huge difference between the variation of each task execution length with respect to the input data. For instance, task 14 varies 356% on average while

task 10 varies only 0.2% on average. In fact, each task in the application has different computation requirements.

In addition to its dynamism, the radio-sensing application follows the streaming execution model. Thus, there are lots of synchronizations between the tasks that might cause the processors to stall. We execute the low-sensitivity version of the radio-sensing application on SCMP with a static and dynamic allocation of the tasks. In fact, the central controller of SCMP is similar to that of AHDAM, so the tasks scheduling and load balancing will be the same. In the static scheduler, the tasks are allocated to only one processor with no task migration. While for the dynamic scheduler, the tasks can be migrated between the processors for load-balancing. The results of the static v/s dynamic scheduling are shown in Figure 5.7.

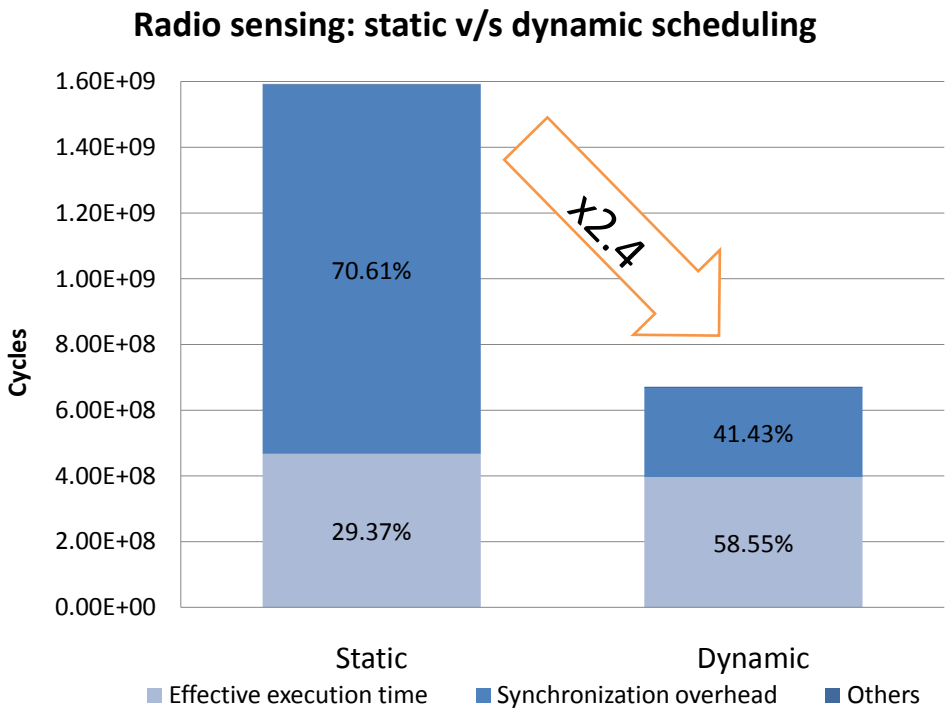
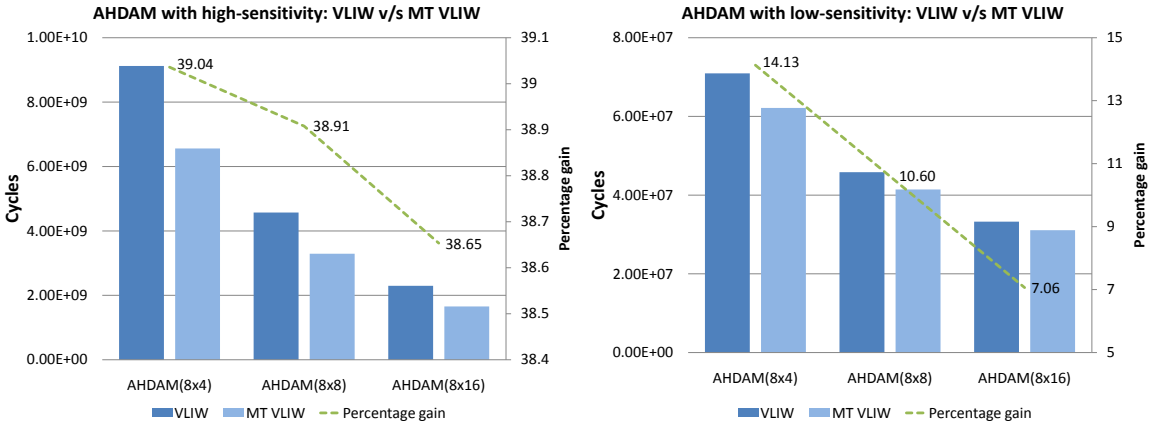


Figure 5.7: Static v/s dynamic scheduling on SCMP for the radio-sensing application. The dynamic scheduling has a speedup of 2.4 over the static scheduling.

The results show that the dynamic scheduler is 2.4 times more performance than the static scheduler. For the static scheduler, the synchronization overhead constitutes more than 70% of the total execution time. However, for the dynamic scheduler, the central controller balances the load dynamically depending on the tasks execution state (active or stalled), thus it is able to find more active tasks to be executed on all the processors. In fact, it is difficult to have an optimal static partitioning prior to the execution of the radio-sensing application because of its dynamism. This implies that the central controller in AHDAM is important for boosting the performance of dynamic applications.

5.3.2 AHDAM: with MT v/s without MT

For this experiment, we evaluate the impact of multithreaded processors on the AHDAM architecture by running the radio-sensing application in 2 modes: low-sensitivity and high-sensitivity. The LPE is implemented as either a monothreaded 3-way VLIW or a blocked multithreaded 3-way VLIW. In Figure 5.8, we plot the performance for the 3 AHDAM configurations with 4/8/16 LPEs per Tile, and for the low-sensitivity and high-sensitivity applications.



(a) Radio-sensing with high-sensitivity

(b) Radio-sensing with low-sensitivity

Figure 5.8: Performance of AHDAM architecture with LPE as monothreaded 3-way VLIW v/s multithreaded 3-way VLIW. AHDAM architecture has 8 Tiles and 4/8/16 LPEs per Tile. Performance results and gain are shown for radio-sensing with a) high-sensitivity b) low-sensitivity.

The results show that for the high-sensitivity application, the impact of multithreading is much higher than that of low-sensitivity. In fact, the high-sensitivity application has a large data set *432 MB* compared to the low-sensitivity *1,025 MB*. This huge difference has a large impact on the memory hierarchy performance. More data cache misses are generated, thus more accesses to the off-chip DDR3 memory. In this scenario, the blocked multithreaded VLIW processor is useful, since it is able to hide the memory access latency by executing another thread. The multithreaded VLIW has a performance gain of *39%* on average for the 3 AHDAM configurations. On the other hand, for the low-sensitivity application, the small data set can fit in the on-chip cache memories, thus there are no frequent accesses to the off-chip memory. Hence, the CPI of the monothreaded and multithreaded VLIW processor are almost identical. This explains the low performance gain due to multithreading, which is *10.5%* on average for the 3 AHDAM configurations. Also note the difference in performance gain the low-sensitivity application when varying the number of LPEs per Tile. This is due to the fact that more threads are executed per LPE, hence more cache contentions between the threads and more accesses to the next level of cache hierarchy. In this case, the multithreaded VLIW has a better performance (14% in the case of AHDAM (8x4)).

5.3.3 AHDAM v/s SCMP v/s monothreaded processor

In this section, we compare the performance of the AHDAM architecture with the SCMP architecture and with a monothreaded processor. The monothreaded processor system is a 1 MIPS32 24K processor with a FPU [91], and a sufficient on-chip memory for data and instructions (432 MB). The memory access time to the on-chip memory is 10 cycles, as well as the L2$ memory. The processor has a 4-KB L1 I$ and an 8-KB L1 D$, and a 32-KB L2 D$. The SCMP system has 8 MIPS32 24K processors with a FPU. Each processor has a 4-KB L1 I$ and an 8-KB L1 D$, and a 32-KB L2 D$. Similarly to the monothreaded system, there are sufficient on-chip memory for data and instructions, while the access time is 10 cycles. For the AHDAM architecture, we consider the 3 configurations that were explained previously in section 5.3. The LPEs are implemented as blocked multithreaded 3-way VLIWs.

The real-time deadline of the radio-sensing application is 6 seconds for both the low-sensitivity and high-sensitivity configurations, which corresponds to 3.10^9 cycles for a 500 MHz processor frequency.

The execution times of the low-sensitivity application on all these systems are shown in Figure 5.9.

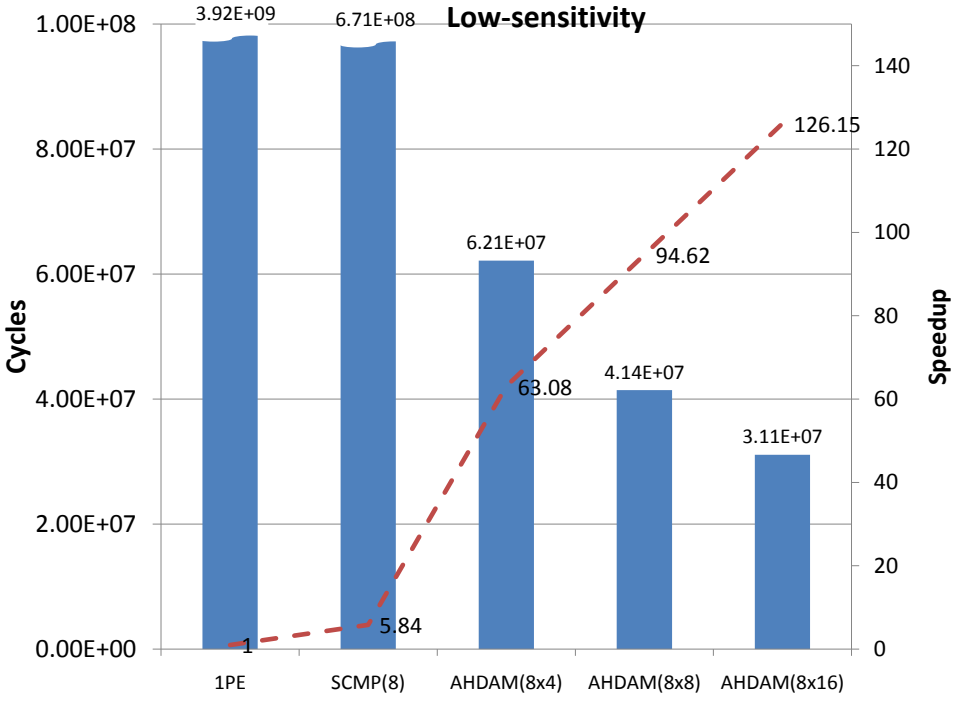


Figure 5.9: Performance of AHDAM v/s SCMP v/s mono for radio-sensing with low-sensitivity. AHDAM architecture has 8 Tiles and 4/8/16 LPEs per Tile. The SCMP architecture has 8 PEs. The real-time deadline of the application is 6 seconds, which corresponds to 3.10^9 cycles for 500 MHz processor frequency (much higher than the y-axis scale.

As we can notice in this figure, both the SCMP and AHDAM were able to meet the real-time requirements, while the monoprocessor was slightly above with an execution time of 7.84 seconds.

The AHDAM(8x16) has a speedup of 126 compared to the monothreaded processor, while the SCMP(8) has only a speedup around 6. For this application requirement, the SCMP performance is sufficient to get the required results since the computation requirements are low (328 MOPS).

However, when running the high-sensitivity application, the performance requirements are much higher as shown in Figure 5.10.

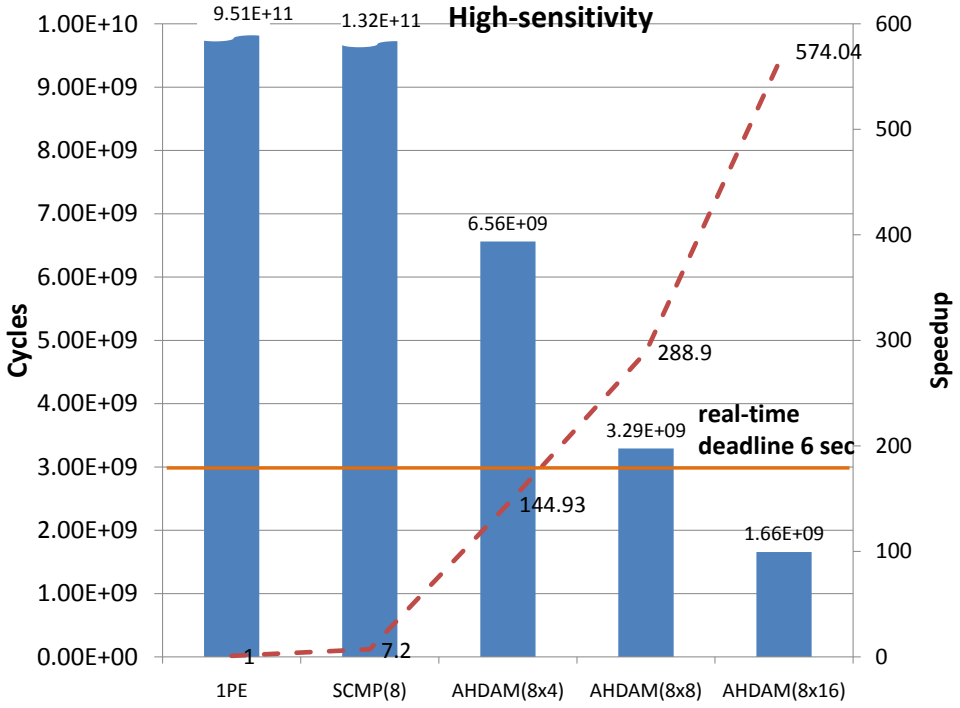


Figure 5.10: Performance of AHDAM v/s SCMP v/s mono for radio-sensing with high-sensitivity. AHDAM architecture has 8 Tiles and 4/8/16 LPEs per Tile. The SCMP architecture has 8 PEs. The real-time deadline of the application is 6 seconds, which corresponds to 3.10^9 cycles for 500 MHz processor frequency.

The results show that none of the architectures is able to meet the real-time requirements under 6 seconds except AHDAM(8x16) architecture, which has a speedup of 574 compared to the monothreaded processor. In fact, the application has a lot of LLP and can be exploited efficiently in AHDAM. Despite the theoretical peak performance of AHDAM(8x8) of 100 GOPS, it only reaches 75.8 GOPS for the high-sensitivity application.

In the next section, we will estimate the overall AHDAM area in 40 nm technology.

5.3.4 AHDAM: chip area estimation

At this stage, AHDAM architecture is not synthesized as a complete chip. However, we are able to estimate the area of the key components in AHDAM architecture in a 40 nm technology, such as the processors (CCP, MPE, LPE) and the memories (instruction memory, Thread Context pools, L1 cache memories, L2 cache memories). For the interconnection networks and busses, we synthesized them in a 40 nm technology and we assumed that the wires are placed above the processors and

caches during the place and route process, thus not occupying more chip area. In Table 5.1, we summarize the area occupation of each component for an 8x16 AHDAM architecture, which implies 8 Tiles and 16 LPEs.

AHDAM unit	Component name	Area of component (um²)	Number of components	Total area (um²)	Source of results
	AntX	11400	1	**11400**	Synthesis TSMC
	1-KB L1 I$	6292.32	1	**6292.32**	CACTI 6.5
	2-KB L1 D$	11743.3	1	**11743.3**	CACTI 6.5
Control	**Control bus**	2412.5	1	**2412.5**	Synthesis TSMC
	MIPS24K	360000	8	**2880000**	www.mips.com
	4-KB L1 I$	18650.7	8	**149205.6**	CACTI 6.5
	8-KB L1 D$	36288.2	8	**290305.6**	CACTI 6.5
	3-way MT VLIW	87053.47	128	**11142844.16**	Synthesis TSMC
	1-KB L1 I$	6292.32	128	**805416.96**	CACTI 6.5
	2-KB L1 D$	11743.3	128	**1503142.4**	CACTI 6.5
	32 KB L2 D$	116531	136	**15848216**	CACTI 6.5
	TCP (32-KB scratchpad)	86638.8	8	**693110.4**	CACTI 6.5
Tile	**Tile NoC**	13237.5	8	**105900**	Synthesis TSMC
	Instruction memory (1.5 MB SRAM)	19674031.36	1	**19674031.36**	CACTI 6.5
Instruction memory	**Instruction network**	113125	1	**113125**	Synthesis TSMC

Table 5.1: AHDAM components area occupation in 40 nm technology and for 8 Tiles and 16 LPEs per Tile.

For the 8x16 AHDAM architecture, there are 1 CCP implemented using the AntX processor, 8 MPEs implemented as MIPS24K with FPU [91], and 128 LPEs implemented as a 3-way blocked multithreaded VLIW with FPU. The CCP and the LPEs have a 1-KB L1 I$ and a 2-KB L1 D$, while the MPEs have a 4-KB L1 I$ and 8-KB L1 D$. Both the MPEs and the LPEs have a 32-KB L2 D$. Each Tile has a 32-KB scratchpad memory for the Thread Context Pool. Also, as mentioned earlier in section 5.1.1, the radio-sensing application necessitates 1.5 MB of on-chip instruction memory. The CCP, the LPE and the interconnection networks are synthesized in a 40 nm TSMC technology, while the MPE area value is taken from MIPS website [91] for a 40 nm TSMC technology. The cache memories and SRAM memories areas are estimated with the CACTI 6.5 tool. The technology used by CACTI tool is based on ITRS roadmap [125], but it is not similar to TSMC technology. Therefore, the processor system is not synthesized with the same technology, but this gives us an idea of the relation between the cache size and the processor size. So, all these components are synthesized/estimated in a 40 nm technology.

In Figure 5.11, we show the surface repartition of the 8x16 AHDAM architecture based on the 3 main components: processors, cache memories, SRAM memories, and interconnection networks. The total area is estimated to be around 53 mm^2 excluding the on-chip DDR3 controller, the MCMU, and the MUX/DEMUX units.

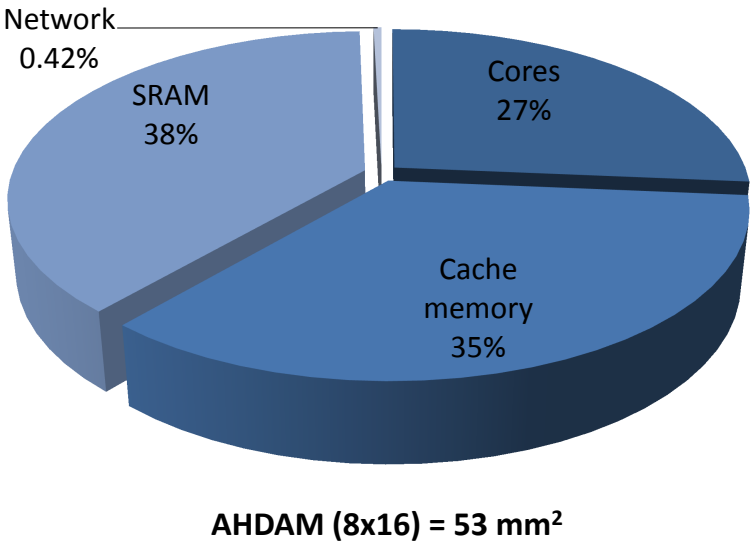


Figure 5.11: AHDAM architecture surface repartition in 40 nm technology for 8 Tiles and 16 LPEs per Tile. The total estimated area is equal to 53 mm^2 excluding the interconnection networks, the on-chip DDR3 controller and the MCMU.

We can notice that the computing cores take 27% of the overall die area, which is quite a good number compared to recent MPSoC architectures. In fact, the key design parameter taken in AHDAM design is to reduce the size of the on-chip memory and integrate more efficient processors for computation. The interconnection networks occupy only 0.4% of the overall die area.

The AHDAM architecture can be used for different application requirements, having different needs in thread parallelism. Therefore, in Figure 5.12, we compare the area of different configurations of AHDAM architecture with 8 Tiles and 4/8/16 LPEs per Tile.

As depicted in this figure, the area difference between AHDAM architecture with 136 cores and 40 cores is only 67% for more than 3 times the number of cores. Therefore, it would be advantageous to select an AHDAM architecture with a higher number of cores, hence a higher peak performance, for only a small increase in chip area.

Finally, we evaluate the impact of multithreaded VLIW on the overall AHDAM area. In Figure 5.13, we show the AHDAM architecture surface with monothreaded VLIWs and multithreaded VLIWs for 8 Tiles and 4/8/16 LPEs per Tile.

The estimated chip area results show that the multithreaded processors have a small impact on the overall chip area. It is only 2.8% more for AHDAM(8x4), while it reaches 7% for AHDAM(8x16). As we saw in section 5.3.2, the performance gain due to multithreading is much higher than the overall chip area increase, hence a high transistor efficiency of the AHDAM architecture.

5.4 Discussion

In this chapter, we evaluated the performance of the AHDAM architecture with respect to a massively parallel application from the telecommunication domain called radio-sensing. The high-

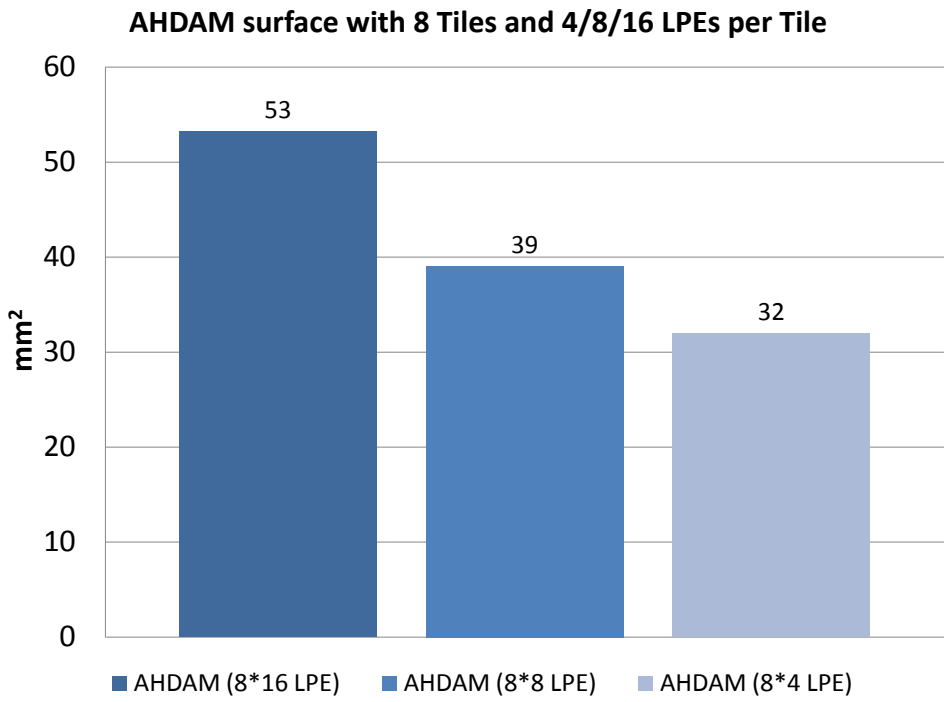


Figure 5.12: AHDAM architecture surface with 8 Tiles and 4/8/16 LPEs per Tile.

sensitivity configuration of this application is characterized by its large data set of *432 MB* and its high computation requirement of *75.8 GOPS*. In addition, 99.8% of its execution time is spent in loops that can be parallelized using OpenMP pragmas.

AHDAM architecture is simulated using a combination of simulator tools such as SESAM and Trimaran, and using the analytical model of the BMT processor described in section 4.3.3.

We conducted several experimentations that lead to interesting conclusions:

1. The asymmetric property of the AHDAM architecture is essential for the dynamic applications to increase their performance. The dynamic scheduling gave a speedup of 2.4 over the static scheduling for the radio-sensing application.

2. A multithreaded VLIW LPE boosts the performance of the AHDAM architecture for only a small area increase. For instance, the AHDAM(8x16) with multithreaded VLIWs gives a performance gain of *39%* for only 7% area increase, as compared with monothreaded VLIWs. Hence, multithreading is a transistor efficient solution for the AHDAM architecture.

3. The VLIW architecture significantly increases the performance of the architecture since it exploits the ILP of the application with only a small area increase. This is why we tend to see VLIW architectures in lot of MPSoC solutions such as Tilera TILE64 [142, 19].

4. Exploiting the loop-level parallelism in hardware boosts significantly the performances, since a large portion of the execution time is spent in loops.

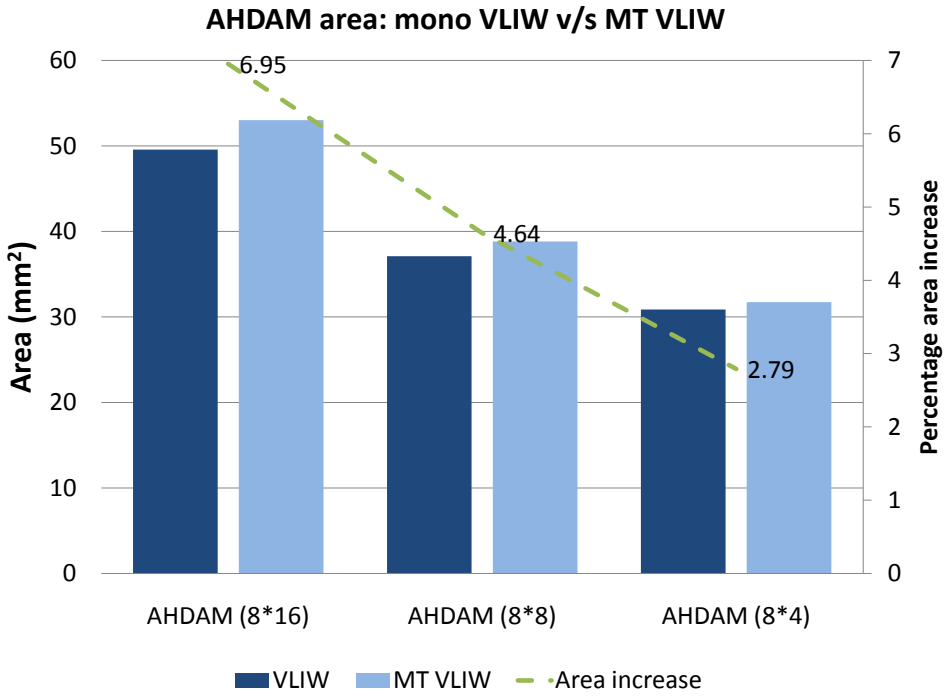


Figure 5.13: AHDAM architecture surface with monothreaded VLIW and multithreaded VLIW for 8 Tiles and 4/8/16 LPEs per Tile.

5. By splitting both instruction and data memories, and implementing a cache architecture for the data, we improved the programmability of the architecture. New applications are easily ported on AHDAM.

6. AHDAM architecture can efficiently and effectively meet the performance requirements of future high-end massively parallel dynamic applications.

Conclusions and Perspectives

If you have a lemon, make Lemonade. – Warren Hinckle, journalist

Contents

To tackle the challenges of future high-end massively-parallel dynamic embedded applications, we have designed the AHDAM architecture, an asymmetric manycore architecture. Its architecture permits to process applications with large data sets by efficiently hiding the processors' stall time using multithreaded processors. Besides, it exploits the parallelism of the applications at the thread and loop levels. AHDAM architecture tackles the dynamism of these applications by dynamically balancing the load between its execution resources using a central controller to increase their utilization rate.

Synthesis of this work

In chapter 1, we defined the context of our study: *massively-parallel dynamic embedded applications*. These applications are highly parallel. The parallelism can be extracted at the thread level (TLP) and at the loop level. So an application might have more than 1000 parallel threads to be processed in parallel. Therefore, *manycore* architectures are natural solutions for these applications. In addition, the dynamism of those applications requires an efficient MPSoC solution to manage the resources occupation and balance the loads in order to maximize the overall throughput. *Asymmetric homogeneous MPSoC* architectures are the best solution for fast and reactive load-balancing. They are also highly transistor and energy efficient because of the separation between control and computing cores. However, these architectures have shown some limitations that prevent them from being scalable to the manycore level and efficient for the long latency memory accesses. Therefore, we have chosen the SCMP architecture as the architecture of reference for experimentations, in order to propose a design improvement of its performance. In particular, we explored two types of architectural improvements: hardware multithreading and scalability.

First of all, we investigated the advantages/disadvantages of hardware multithreading for embedded systems in chapter 2. We started by designing two small footprints, scalar, in-order multithreaded processor for the embedded systems based on a monothreaded AntX processor: Interleaved Multithreading (IMT) and Blocked Multithreading (BMT). The synthesis results in a 40 nm TSMC technology showed that the register file occupies more than 38% of the overall core area, thus it is not area efficient to integrate more than 2 thread contexts (TC) per multithreaded processor. Therefore, we have chosen to implement a multithreaded processor with 2 TCs. Both multithreaded

processors were synthesized in a 40 nm TSMC technology. The results shows that the IMT and BMT processors have 73.4% and 61.3% increase in core area versus the monothreaded core. Thus, the BMT has a smaller area. Finally, we compared the performances and transistor efficiency of both MT cores using a bubble sort application, while varying the L1 data cache size and the data memory latency. The results have shown that there is a trade-off between the data cache memory size, the data memory latency, and the core area overhead. Choosing the best processor highly depends on the system designer specifications and the application requirements.

Based on this conclusion, we explored in chapter 3 the performance impact of the multithreaded processor in the SCMP architecture. For this reason, we developed a new multithreaded ISS in SystemC language and integrated it in SESAM, which is the simulation environment for SCMP. The new SCMP architecture with multiple multithreaded processors has been called MT_SCMP. We conducted several benchmarks based on control-flow and streaming applications in order to choose which multithreaded processor suits best for MT_SCMP (IMT v/s BMT), which global thread scheduling for multiple multithreaded processors gives the best performance (VSMP v/s SMTC), and which asymmetric MPSoC architecture is the most performing and transistor efficient (SCMP v/s MT_SCMP). The results have shown that the blocked multithreaded processor (BMT) and the SMTC scheduler suits best for MT_SCMP [18], and thus are adapted as fixed system design parameters for this architecture. Finally, we compared the performances and transistor efficiency of SCMP and MT_SCMP by running 2 types of embedded applications: connected component labeling (control-flow) and WCDMA streaming. The MT_SCMP had better peak performance, but less transistor efficiency than SCMP. Whether to choose multithreaded processors for SCMP or not, depends on the system designer. If peak performance is a key parameter, then multithreaded processors are an interesting solution. However, for transistor efficiency, monothreaded processors remain a more efficient solution. As for high-end massively-parallel dynamic embedded applications with large data sets, there are lots of parallelism at the thread level (TLP) and at the loop level (LLP) that should be exploited by the architecture. The SCMP architecture has shown scalability, extensibility, programmability, and parallelism limitations for such applications.

Therefore, we proposed a novel solution that target the manycore era in chapter 4. The proposed architecture is called AHDAM, which stands for *Asymmetric Homogeneous with Dynamic Allocator Manycore architecture*. AHDAM has been designed to tackle the challenges of future high-end massively parallel dynamic embedded applications. It is used as an on-chip accelerator and it exploits the parallelism at the thread level (TLP) and loop level (LLP). We presented in details its programming model and the functionality of all its components. In addition, we studied the scalability of this architecture and we deduced that it can support 136 processors depending on the application requirement; hence AHDAM has reached the manycore level.

Finally in chapter 5, we evaluated the performance of AHDAM architecture with respect to a massively parallel dynamic application from the telecommunication domain called radio-sensing. The high-sensitivity configuration of this application is characterized by its large data set of *432 MB*, its high computation requirement of *75.8 GOPS*, and its dynamism. In addition, 99.8% of its execution time is spent in loops that can be parallelized using OpenMP pragmas. AHDAM architecture was simulated using a combination of simulator tools such as SESAM and Trimaran, and using the analytical model of the BMT processor that we have developed. After we have conducted several experimentations, we concluded that the asymmetric property of the AHDAM architecture is essential for the dynamic applications to increase their performance. The dynamic scheduling

gave a speedup of 2.4 over the static scheduling for the radio-sensing application. In addition, multithreading boosts the performance of AHDAM architecture and is a transistor efficient solution. Finally, AHDAM architecture is a powerful improvement over SCMP and can meet the performance requirements of future high-end massively parallel dynamic applications.

My PhD works and results contribute today to the CEA LIST roadmap and the LCE laboratory activities. However, there are still some important validations to the proposed architectural concepts in AHDAM architecture that should be done before industrializing the solution, and which we summarize them in the next section.

Perspectives

Short term

Despite the evaluations we have conducted in this thesis, there are lots of proposed concepts in AHDAM architecture that still need explorations, developments and improvements. The short term perspectives can be divided into three main steps: development of a simulator, building a prototype, and comparison with other manycore architectures.

In the first step, we need to develop a simulator for AHDAM, mainly an extension of the SESAM simulator environment. New components should be developed in SystemC that did not exist previously for the SCMP architecture, such as the L2 cache memory and its protocols, the Thread Control Pool and the TCP state scratchpad memories. In addition, a multithreaded VLIW ISS should be developed. Then, we need to encapsulate all the Tile's units in one module in order to look as one PE for the CCP, and validate all the Tile functionalities. In particular, the Tile NoC architecture should be investigated. Finally, the L2 instruction and data memories should be split.

After building the AHDAM simulator environment, the proposed runtime environment for fork-join threads should be developed. This runtime is a critical part of the AHDAM functionality and the intra-Tile and inter-Tile management. The heuristic behind finding the optimal number of threads to be forked should be investigated in more details, since it is an important parameter for the overall loop-regions acceleration. In addition, the farming execution model should be validated. At this stage, we can experiment new features in global scheduling, such as the possibility to execute more than one task on each Tile by allowing the preemption of the MPEs and the LPEs in order to dynamically adapt the resources depending on the application requirements. Furthermore, new concepts of memory management can be tested, such as the dynamic allocation of data buffers in the off-chip DDR3 memory, and implementing a data prefetcher from the DDR3 to the L2$ memories.

Having the AHDAM simulator and runtime environment in place, it would be interesting to continue the development of the automatic programming toolchain that we started in chapter 4. It could be based on the PAR4ALL tool. This will allow us to port any legacy code easily to AHDAM architecture.

The second main step consists of building a prototype of the AHDAM architecture on a hardware emulation board. This prototype will be the proof of concept of the architecture. Having such a test chip prototype, we can estimate to an accurate value the transistor and energy efficiency of the AHDAM architecture as well as the multithreaded processors. In particular, we can render the AHDAM chip more energy efficient by exploring new load-balancing strategies inside each Tile and

between the Tiles, and integrate the strategies in the runtime environment. Other energy efficient techniques such as DVFS can be implemented on FPGA, but it would be more accurate on a final ASIC solution. We can image that each Tile is running on a different frequency level and can be controlled by the CCP depending on the application requirements.

Finally, the third main step consists in comparing AHDAM architecture with other relevant manycore solutions such as Tilera TILE64 [142, 19], ST Microelectronics P2012 [84], and Kalray MPPA [72]. For this reason, we need to port several relevant dynamic embedded applications from several domains that have lots of parallelism and computation requirements. These applications should run on all these chips and a fair comparison would be conducted. At this stage, we are ready to conduct a technological transfer of the AHDAM chip solution to industries and national/European projects. In particular, we can develop two versions of AHDAM chip: low-end and high-end. The first version targets the embedded market, while the last one targets the server market, and especially cloud computing. What would differentiate both chips is the number of Tiles, the number of LPEs per Tile, and the load-balancing strategies utilized in the chip that would target performance or energy efficiency.

Long term

On the long term, there are several architectural improvements that we imagine for AHDAM architecture.

As the process technology improves, there are more concerns about the reliability of the AHDAM architecture. AHDAM could be used in critical domains such as military, nuclear and space applications, where fault tolerance is a non-negligible architectural decision. We can imagine that AHDAM chip would be fault-tolerant on the Tile, MPE and LPE levels by integrating spare components.

In addition, as we are experiencing nowadays, there is a huge gap between the processor and memory speed. This does not seem to change in the future unless a new technological breakthrough has been found for the memory technologies. Assuming this is not the case, there should be an architectural solution for keeping the LPE multithreaded processors from stalling. One solution would be to increase the number of hardware threads per LPE. But as we saw previously in chapter 2, this is not a transistor efficient solution for small footprint processors, a new technique would be to use a N out of M static interleaving multithreading architectures. This technique implies that a multithreaded processor has N foreground threads (hardware thread contexts) and M virtual threads stored in a special scratchpad memory close to the multithreaded processor. In this way, we are increasing the number of supported child threads per LPE.

AHDAM chip is a manycore architecture. But as we saw in chapter 4, there are also limitations to the scalability of the architecture. One solution would be to integrate more DDR3 controllers on-chip, thus increasing the number of Tiles. Another solution to the scalability problem is to consider AHDAM architecture as an optimized cluster in a multi-cluster environment. Then, by using a hierarchical solution, we can increase the number of cores dramatically (more than 1000 cores). At this stage, we could imagine that the AHDAM programming model is extended to support MPI communication between the different AHDAM clusters. Thus, AHDAM would support OpenMP + MPI.

Finally, the on-chip SRAM and cache memories can be stacked on top of the cores using a 3D

stacking technology [50]. This would be a dramatic improvement to the chip size, since 73% of the chip estimated area is occupied by the cache and SRAM memories. Thus, more cores could be integrated and the memory access times would be faster. This will improve the performance of AHDAM chip and perhaps new architectural improvements should be proposed when using a 3D stacking technology.

Glossary

ADL Architecture Description Language. 51, 56–58, 124

AHB Advanced High-performance Bus. 41, 124

AHDAM Asymmetric Homogeneous with Dynamic Allocation Manycore. i, v, vii–x, xviii, xix, xxi, 3, 4, 80, 82–89, 92, 93, 95, 97–103, 107–124

ALU Arithmetic Logic Unit. 28, 33, 124

BMT Blocked MultiThreading. vi–viii, xvii, xviii, 3, 4, 26, 30, 31, 33, 35–45, 48, 53, 56, 63, 65, 66, 68–71, 74, 75, 79, 93–95, 108, 117, 119, 120, 124

CCP Central Controller Processor. xviii, 18, 19, 52, 53, 74–76, 78, 80, 85, 87, 88, 95–99, 101, 114, 115, 121, 122, 124

CDFG Control Data Flow Graph. xviii, 20–22, 66, 67, 69, 80, 83–85, 87, 95, 106, 124

CMT Chip MultiThreading. 30, 57, 124

CPI Cycle Per Instruction. xviii, 89–95, 112, 124

DDR Double Data Rate. xviii, 87, 88, 100, 124

DDR3 Double Data Rate type 3. x, 82, 86–89, 95, 97, 100, 101, 108, 110, 112, 115, 116, 121, 122, 124

DLP Data-Level Parallelism. 7, 124

DMA Direct Memory Access. 19, 23, 50–52, 80, 85, 87, 95, 124

DSP Digital Signal Processor. 19, 124

DVFS Dynamic Voltage and Frequency Scaling. ix, 122, 124

EX EXecute. xvii, 33, 36, 59, 60, 124

FFT Fast Fourier Transform. 105, 124

FPU Floating Point Unit. 28, 51, 59, 108, 113, 115, 124

FSM Finite State Machine. xvii, xviii, 33, 36–38, 40, 63–65, 124

GOPS Giga Operations Per Second. i, viii, 7, 107, 110, 114, 117, 120, 124

GSM Global System for Mobile communications. 105, 124

HPC High-Performance Computing. 14, 83, 124

HW Hardware. 8, 13–15, 18, 19, 124

I/O Input/Output. 14, 19, 23, 36, 37, 45, 47, 52, 53, 61, 62, 74, 80, 85, 124

ID Instruction Decode. 33, 34, 59, 61, 124

IF Instruction Fetch. 33, 36–39, 59–61, 63–65, 89, 124

ILP Instruction-Level Parallelism. vi, 5, 7, 10, 88, 108, 117, 124

IMT Interleaved MultiThreading. vi, vii, xvii, xviii, 3, 4, 26, 29–31, 33, 35, 37–45, 48, 53, 56, 63–66, 68–71, 79, 119, 120, 124

IP Intellectural Property. 8, 14, 19, 30, 56, 57, 60, 100, 124

IPC Instruction Per Cycle. 11, 15, 23, 24, 28, 29, 34, 44, 47, 82, 124

ISA Instruction-Set Architecture. 8, 33, 48, 56, 58, 59, 124

ISS Instruction-Set Simulator. vii, xvii, 4, 48–51, 53, 56–66, 70, 78, 107, 120, 121, 124

KIPS Kilo Instructions Per Second. 57, 124

LLP Loop-Level Parallelism. iv, v, 80, 83, 103, 106, 114, 120, 124

LPE Loop Processing Element. viii–x, xviii, xix, 88, 89, 95–97, 99, 101, 102, 107–110, 112–118, 121, 122, 124

MCMU Memory Configuration and Management Unit. 19, 21–23, 52, 70, 74, 78, 83–85, 87, 95, 98, 115, 116, 124

MEM MEMory. 33, 34, 36–38, 59, 62, 63, 65, 89, 124

MOPS Million Operation Per Second. 101, 107, 114, 124

MPE Master Processing Element. viii, ix, xviii, 88, 96–99, 101, 102, 107, 109, 110, 114, 115, 121, 122, 124

MPI Message Passing Interface. x, 13, 122, 124

MPSoC Multi-Processor System-On-Chip. i, iv–vii, x, xvii, xxi, 2–4, 6, 8–17, 21, 23, 24, 27, 30, 33, 45, 47–50, 52, 55–58, 66, 78, 79, 81, 116, 117, 119, 120, 124

MT MultiThreaded. vi, vii, xviii, 14, 16, 40, 44, 45, 48, 65, 66, 68, 70, 71, 73–80, 120, 124

NFS Network File System. 50, 124

NoC Network-On-Chip. 6, 8, 50–52, 88, 121, 124

OS Operating System. 10, 13, 15, 17, 27, 54, 55, 124

OSoC Operating System accelerator On Chip. 18, 19, 23, 52, 124

PC Program Counter. 33, 35–39, 50, 59, 124

PE Processing Element. viii, ix, xviii, xxi, 18, 19, 21–23, 40, 52–56, 62, 71–80, 85, 88, 90–96, 99, 113, 114, 121, 124

RF Register File. 37–39, 124

RTL Register Transfer Language. vi, 3, 26, 33, 35, 37, 39, 56, 57, 124

RTOS Real-Time Operating System. 18, 19, 87, 124

SCC Single-chip Cloud Computer. 13, 124

SCMP Scalable Chip MultiProcessor. v–viii, xvii, xviii, 3, 4, 6, 15–24, 45, 48, 52, 53, 56, 65, 66, 73–81, 84–87, 89, 91–93, 99, 102, 103, 107, 110, 111, 113, 114, 119–121, 124

SDRAM Synchronous Dynamic Random Access Memory. 100, 124

SESAM Simulation Environment for Scalable Asymmetric Multiprocessing. vii, viii, xvii, 4, 48–54, 56, 66, 78, 107, 109, 117, 120, 121, 124

SIMD Single Input Multiple Data. 7, 10, 28, 124

SMP Symmetric MultiProcessing. 13, 54, 55, 124

SMT Simultaneous MultiThreading. 29, 57, 124

SMTC Symmetric Multi-Thread-Context. vii, xvii, 54–56, 66, 71–73, 79, 97, 120, 124

SoC System-On-Chip. xvii, 6, 14, 45, 58, 60, 61, 100, 124

SPMD Single Program Multiple Data. 83, 124

SRAM Static Random Access Memory. x, 86, 115, 122–124

SW Software. 13, 54, 73, 124

TC Thread Context. vi, vii, 4, 26, 31, 34–38, 41–43, 45, 53–56, 62–64, 68, 70, 72, 73, 75–77, 88, 93–96, 110, 119, 124

TCP Thread Context Pool. viii, 88, 96, 121, 124

TDM Time Division Multiplexing. 101, 124

TLB Translation Lookaside Buffer. 19, 23, 50, 52, 53, 65, 70, 124

TLM Transaction Level Modeling. 48–52, 56–58, 61–64, 124

TLP Thread-Level Parallelism. iii–v, vii, 2, 6–8, 11, 16, 23, 25–27, 80, 83, 103, 106, 119, 120, 124

TLS Thread-Level Speculation. 14, 27, 124

TOPS Tera Operations Per Second. i, iv, 2, 6, 7, 9, 124

VHDL Very-High-Speed-Integrated-Circuits Hardware Description Language. vi, 3, 18, 26, 39, 124

VLIW Very Long Instruction Word. iv, vi, viii, xviii, xix, 2, 8, 19, 25, 88, 89, 107–109, 112, 115–118, 121, 124

VSMP Virtual Symmetric Multi-Processing. vii, xvii, 54–56, 66, 71–73, 79, 120, 124

WAR Write-After-Read. 34, 124

WAW Read-After-Write. 34, 124

WAW Write-After-Write. 34, 124

WB Write-Back. 33, 34, 59, 61, 124

WCDMA Wideband Code Division Multiple Access. vii, xviii, 4, 68, 69, 75, 78, 79, 120, 124

WCET Worst Case Execution Time. 29, 124

Bibliography

[1] The GNU GDB project. http://www.gnu.org/software/gdb/. (Cited on page 50.)

[2] Hot Topic II - Power Supply and Power Management in Ubicom. page 1, april 2007. (Cited on page 30.)

[3] B. Ackland, A. Anesko, D. Brinthaupt, S.J. Daubert, A. Kalavade, J. Knobloch, E. Micca, M. Moturi, C.J. Nicol, J.H. O'Neill, J. Othmer, E. Sackinger, K.J. Singh, J. Sweet, C.J. Terman, and J. Williams. A single-chip, 1.6-billion, 16-b MAC/s multiprocessor DSP. *IEEE Journal of Solid-State Circuits*, 35(3):412 –424, mar 2000. (Cited on page 6.)

[4] A. Agarwal, R. Bianchini, D. Chaiken, F.T. Chong, K.L. Johnson, D. Kranz, J.D. Kubiatowicz, Beng-Hong Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine. *Proceedings of the IEEE*, 87(3):430 –444, mar 1999. (Cited on pages 30 and 32.)

[5] A. Aiken and A. Nicolau. Optimal loop parallelization. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 308–317, New York, NY, USA, 1988. ACM. (Cited on page 7.)

[6] H. Akkary and M.A. Driscoll. A dynamic multithreading processor. In *MICRO-31: Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 226 –236, nov-2 dec 1998. (Cited on page 27.)

[7] Gene Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference*, pages 483–485, 1987. (Cited on page 7.)

[8] C. Araujo, M. Gomes, E. Barros, S. Rigo, R. Azevedo, and G. Araujo. Platform designer: An approach for modeling multiprocessor platforms based on SystemC. *Design Automation for Embedded Systems*, 10(4):253–283, 2005. (Cited on page 58.)

[9] ARM. Cortex A-9 Processor. http://www.arm.com/products/processors/cortex-a/cortex-a9.php. (Cited on page 13.)

[10] Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams, and Katherine A. Yelick. The landscape of parallel computing research: a view from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006. (Cited on pages 24 and 26.)

[11] T. Austin, E. Larson, and D. Ernst. SimpleScalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, Feb. 2002. (Cited on page 57.)

[12] Christopher Batten, Ajay Joshi, Jason Orcutt, Anatoly Khilo, Benjamin Moss, Charles Holzwarth, Milos Popovic, Hanqing Li, Henry Smith, Judy Hoyt, Franz Kartner, Rajeev Ram, Vladimir Stojanovic, and Krste Asanovic. Building Manycore Processor-to-DRAM

Networks with Monolithic Silicon Photonics. In *Proceedings of the 2008 16th IEEE Symposium on High Performance Interconnects*, pages 21–30, Washington, DC, USA, 2008. IEEE Computer Society. (Cited on pages 23 and 47.)

[13] V. Baumgarte, G. Ehlers, F. May, A. Nückel, and M. Vorbach. PACT XPP - A Self-Reconfigurable Data Processing Architecture. *Supercomputing*, 26:167–184, 2003. (Cited on page 19.)

[14] C. Bechara, A. Berhault, N. Ventroux, S. Chevobbe, Y. LHuillier, R. David, and D. Etiemble. A small footprint interleaved multithreaded processor for embedded systems. In *18th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, Beirut, Lebanon, December 2011. (Cited on pages 35 and 36.)

[15] C. Bechara, N. Ventroux, and D. Etiemble. Towards a Parameterizable cycle-accurate ISS in ArchC. In *IEEE International Conference on Computer Systems and Applications (AICCSA)*, Hammamet, Tunisia, May 2010. (Cited on pages 51, 56, 59, 61 and 62.)

[16] C. Bechara, N. Ventroux, and D. Etiemble. A TLM-based Multithreaded Instruction Set Simulator for MPSoC Simulation Environment. In *3rd Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO 2011), Held in conjunction with the HiPEAC conference*, Heraklion, Greece, January 2011. (Cited on pages 56 and 62.)

[17] C. Bechara, N. Ventroux, and D. Etiemble. AHDAM: an Asymmetric Homogeneous with Dynamic Allocator Manycore chip. In *Facing the Multicore Challenge II*, Karlsruhe, Germany, September 2011. (Cited on pages 4 and 82.)

[18] C. Bechara, N. Ventroux, and D. Etiemble. Comparison of different thread scheduling strategies for Asymmetric Chip MultiThreading architectures in embedded systems. In *14th Euromicro conference on Digital System Design (DSD 2011)*, Oulu, Finland, September 2011. (Cited on pages 55, 71, 79, 97 and 120.)

[19] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - Processor: A 64-Core SoC with Mesh Interconnect. In *ISSCC'08: IEEE International Solid-State Circuits Conference*, pages 88 –598, feb. 2008. (Cited on pages 13, 117 and 122.)

[20] G. Beltrame, C. Bolchini, L. Fossati, A. Miele, and D. Sciuto. ReSP: A non-intrusive Transaction-Level Reflective MPSoC Simulation Platform for design space exploration. In *ASPDAC'08: Proceeding of the Asia and South Pacific Design Automation Conference*, pages 673–678, 2008. (Cited on page 58.)

[21] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino. Legacy SystemC co-simulation of multi-processor systems-on-chip. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 494–499, 16–18 Sept. 2002. (Cited on page 57.)

[22] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability Analysis of Global Scheduling Algorithms on Multiprocessor Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 20(4):553–566, April 2008. (Cited on page 8.)

[23] G. Blake, R. G. Dreslinski, and T. Mudge. A survey of multicore processors. *IEEE Signal Processing Magazine*, 26(6):26–37, October 2009. (Cited on page 9.)

[24] H. Blume, J.v. Livonius, L. Rotenberg, T.G. Noll, H. Bothe, and J. Brakensiek. Performance and Power Analysis of Parallelized Implementations on an MPCore Multiprocessor Platform. In *IC-SAMOS'07: International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 74 –81, july 2007. (Cited on page 13.)

[25] B. Boothe and A. Ranade. Improved Multithreading Techniques for Hiding Communication Latency in Multiprocessors. In *The 19th Annual International Symposium on Computer Architecture*, pages 214 –223, 1992. (Cited on page 31.)

[26] S. Borkar. Microarchitecture and Design Challenges for Gigascale Integration. *MICRO-37: 37th International Symposium on Microarchitecture*, page 3, dec. 2004. (Cited on page 14.)

[27] F. R. Boyer, Liping Yang, E. M. Aboulhamid, L. Charest, and G. Nicolescu. Multiple SimpleScalar processors, with introspection, under SystemC. In *Proceedings of the IEEE International Symposium on Micro-NanoMechatronics and Human Science*, volume 3, pages 1400–1404 Vol. 3, 2003. (Cited on page 57.)

[28] J.D. Brown, S. Woodward, B.M. Bass, and C.L. Johnson. IBM Power Edge of Network Processor: A Wire-Speed System on a Chip. *IEEE Micro*, 31(2):76 –85, march-april 2011. (Cited on page 14.)

[29] M. Butler, Tse-Yu Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single instruction stream parallelism is greater than two. In *18th Annual International Symposium on Computer Architecture*, pages 276 – 286, 1991. (Cited on page 11.)

[30] S. Byna, Yong Chen, and Xian-He Sun. A Taxonomy of Data Prefetching Mechanisms. In *I-SPAN 2008: International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 19 –24, may 2008. (Cited on page 14.)

[31] G.T. Byrd and M.A. Holliday. Multithreaded processor architectures. *IEEE Spectrum*, 32(8):38 –46, aug 1995. (Cited on page 26.)

[32] Lukai Cai and Daniel Gajski. Transaction level modeling: an overview. *First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 19–24, 2003. (Cited on pages 48 and 56.)

[33] ClearSpeed. CSX700 Processor. http://www.clearspeed.com/products/csx700.php. (Cited on page 14.)

[34] Codesourcery. http://www.codesourcery.com. (Cited on page 13.)

[35] J. Cong, K. Gururaj, G. Han, A. Kaplan, M. Naik, and G. Reinman. MC-Sim: An efficient simulation tool for MPSoC designs. In *ICCAD'08: IEEE/ACM International Conference on Computer-Aided Design*, pages 364–371, 2008. (Cited on page 48.)

[36] S. Cordibella, F. Fummi, G. Perbellini, and D. Quaglia. A HW/SW co-simulation framework for the verification of multi-CPU systems. In *HLDVT '08: IEEE International High Level Design Validation and Test Workshop*, pages 125–131, 2008. (Cited on page 57.)

[37] David E. Culler. *Multithreading: Fundamental limits, potential gains, and alternatives. In Multithreaded Computer Architecture*. Kluwer Academic Publishers, 1994. (Cited on page 31.)

[38] R. David, S. Pillement, and O. Sentieys. *Low Power Electronics Design*, volume 1 of *Computer Engineering*, chapter Energy-Efficient Reconfigurable Processors. CRC Press, 2004. (Cited on page 19.)

[39] M. R. de Schultz, A. K. I. Mendonca, F. G. Carvalho, O. J. V. Furtado, and L. C. V. Santos. Automatically-retargetable model-driven tools for embedded code inspection in SoCs. In *Proceedings of the 50th Midwest Symposium on Circuits and Systems MWSCAS 2007*, pages 245–248, 5–8 Aug. 2007. (Cited on page 58.)

[40] T.C. Deepak Shekhar and K. Varaganti. Parallelization of Face Detection Engine. In *39th International Conference on Parallel Processing Workshops (ICPPW)*, pages 113 –117, sept. 2010. (Cited on page 13.)

[41] R. Dimond, O. Mencer, and W. Luk. Application-specific customisation of multi-threaded soft processors. *IEE Proceedings on Computers and Digital Techniques*, 153(3):173 – 180, may 2006. (Cited on page 26.)

[42] Marc Duranton, Sami Yehia, Bjorn De Sutter, Koen De Bosschere, Albert Cohen, Babak Falsafi, Georgi Gaydadjiev, Manolis Katevenis, Jonas Maebe, Harm Munk, Nacho Navarro, Alex Ramirez, Olivier Temam, and Mateo Valero. *The HiPEAC Vision*. HiPEAC network of excellence , 2010. (Cited on pages 1 and 5.)

[43] Ali El-Moursy. *Highly Efficient Multi-Threaded Architecture*. VDM Verlag, Germany, 2009. (Cited on page 29.)

[44] Eleven Engineering. XInC wireless multithreaded processor. http://www.elevenengineering.com/products/chips/XInC.php. (Cited on page 30.)

[45] C.J. Chen F. Chang and C.J. Lu. A Linear-Time Component-Labeling Algorithm Using Contour Tracing Technique. *Computer Vision and Image Understanding*, 93(2):206–220, 2004. (Cited on pages 8 and 66.)

[46] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *EDTC '95: Proceedings of the 1995 European conference on Design and Test*, page 503, Washington, DC, USA, 1995. IEEE Computer Society. (Cited on page 57.)

[47] Karl filip Faxén, Christer Bengtsson, Mats Brorsson, Erik Hagersten, Bengt Jonsson, Christoph Kessler, Björn Lisper, Per Stenström, and Bertil Svensson. Multicore computingthe state of the art. http://soda.swedish-ict.se/3546/, 2008. (Cited on page 9.)

[48] J.A. Fisher, P. Faraboschi, and C. Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers, and Tools*. Morgan Kaufmann Publishers, 2005. (Cited on pages 2 and 88.)

[49] David Fotland. Ubicom's MASI Wireless Network Processor. www.hotchips.org/archives/hc15/3_Tue/7.ubicom.pdf, 2003. (Cited on page 30.)

[50] Philip Garrou, Christopher Bower, and Peter Ramm. *Handbook of 3D Integration: Technology and Applications of 3D Integrated Circuits (2nd edition)*. Wiley-VCH, 2008. (Cited on page 123.)

[51] Frank Ghenassia. *Transaction-Level Modeling with Systemc: TLM Concepts and Applications for Embedded Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. (Cited on pages 48 and 56.)

[52] A. Gonzalez-Escribano and D.R. Llanos. Speculative Parallelization. *Computer*, 39(12):126 –128, dec. 2006. (Cited on page 14.)

[53] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future. *IEEE Micro*, 31(2):86 –95, march-april 2011. (Cited on page 14.)

[54] Thorsten Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002. (Cited on page 49.)

[55] M. Gschwind, H.P. Hofstee, B. Flachs, M. Hopkin, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10 –24, march-april 2006. (Cited on page 15.)

[56] A Guerre. *Approche hiérarchique pour la gestion dynamique des tâches et des communications dans les architectures massivement parallèles programmables*. PhD thesis, Université Paris-Sud 11, September 2010. (Cited on pages 7, 80, 85 and 86.)

[57] A. Guerre, N. Ventroux, R. David, and A. Merigot. Approximate-Timed Transaction Level Modeling for MPSoC Exploration: a Network-on-Chip Case Study. In *12th Euromicro Conference on Digital System Design*, 2009. (Cited on pages 48, 49 and 52.)

[58] Yao Guo, P. Narayanan, M.A. Bennaser, S. Chheda, and C.A. Moritz. Energy-Efficient Hardware Data Prefetching. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(2):250 –263, feb. 2011. (Cited on page 14.)

[59] J. Gustafson. Re-evaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988. (Cited on pages 7 and 8.)

[60] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An Instruction Set Description Language For Retargetability. In *Proceedings of the 34th Design Automation Conference*, pages 299–302, Jun 1997. (Cited on page 57.)

[61] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. EXPRESSION: a language for architecture exploration through compiler/simulator retargetability. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, page 100, New York, NY, USA, 1999. ACM. (Cited on page 57.)

[62] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. (Cited on pages 33, 59 and 61.)

[63] Jan Hoogerbrugge and Andrei Terechko. Transactions on high-performance embedded architectures and compilers III. chapter A multithreaded multicore system for embedded media processing, pages 154–173. Springer-Verlag, Berlin, Heidelberg, 2011. (Cited on pages 15 and 16.)

[64] R. M. Hord. *The Illiac-IV, The First Supercomputer*. ISBN: 0914894714. Computer Science Press, May 1982. (Cited on page 7.)

[65] J. Howard, S. Dighe, S.R. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V.K. De, and R. Van Der Wijngaart. A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling. *IEEE Journal of Solid-State Circuits*, 46(1):173 –183, jan. 2011. (Cited on page 13.)

[66] M. Howard and A. Kopser. Design of the Tera MTA integrated circuits. In *19th Annual Symposium on Gallium Arsenide Integrated Circuit (GaAs IC) Symposium*, pages 14 –17, oct 1997. (Cited on page 29.)

[67] Kai Hwang. *Advanced Computer Architecture: Parallelism,Scalability,Programmability*. McGraw-Hill Higher Education, 1st edition, 1992. (Cited on page 32.)

[68] Infineon. TriCore 2. http://www.infineon.com/. (Cited on pages 26 and 31.)

[69] INTEL. RCCE specification for Intel SCC. http://techresearch.intel.com/spaw2/uploads/files/RCCE_Specification.pdf. (Cited on page 13.)

[70] International Telecommunications Union (ITU). *National Spectrum Management*. 2005. (Cited on page 104.)

[71] I. Horiba K. Suzuki and N. Sugie. Linear-time connected-component labeling based on sequential local operations. *Computer Vision and Image Understanding*, 89(1):1–23, 2003. (Cited on page 66.)

[72] Kalray semiconductors. Multi-Purpose Processor Array: MPPA. http://www.kalray.eu/. (Cited on page 122.)

[73] N. Kavvadias and S. Nikolaidis. Elimination of Overhead Operations in Complex Loop Structures for Embedded Microprocessors. *IEEE Transactions on Computer*, 57(2):200–214, Feb. 2008. (Cited on page 58.)

[74] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, 2005. (Cited on pages 11, 14 and 26.)

[75] D. Koufaty and D.T. Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2):56 – 65, march-april 2003. (Cited on page 29.)

[76] Janusz S. Kowalik, editor. *on Parallel MIMD computation: HEP supercomputer and its applications*, Cambridge, MA, USA, 1985. Massachusetts Institute of Technology. (Cited on page 29.)

[77] Lionel Lacassagne and Bertrand Zavidovique. Light speed labeling: efficient connected component labeling on RISC architectures. *Journal of Real-Time Image Processing*, pages 1–19, 2009. 10.1007/s11554-009-0134-0. (Cited on pages 8 and 66.)

[78] M.S. Lam and R.P. Wilson. Limits of Control Flow on Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46 –57, 1992. (Cited on page 11.)

[79] Y. Le Moullec, C. Leroux, E. Baud, and P. Koch. Power consumption estimation of the multi-threaded xinc processor. *Proceedings of the Norchip Conference*, pages 210 – 213, nov. 2004. (Cited on page 30.)

[80] A.S. Leon and D. Sheahan. The U1traSPARC T1: A Power-Efficient High-Throughput 32-Thread SPARC Processor. *ASSCC'06: IEEE Asian Solid-State Circuits Conference*, pages 27 –30, nov. 2006. (Cited on page 29.)

[81] Rainer Leupers and Peter Marwedel. Retargetable Code Generation based on Structural Processor Descriptions. In *Design Automation for Embedded Systems*, pages 1–36. Kluwer Academic Publishers, 1998. (Cited on page 57.)

[82] Linux kernel. www.kernel.org. (Cited on page 13.)

[83] Linux on ARM. http://www.linux-arm.org. (Cited on page 13.)

[84] Luca Benini. Programming Heterogeneous Many-core platforms in Nanometer Technology: the P2012 experience. http://www.artist-embedded.org/docs/Events/2010/Autrans/talks/PDF/Benini/BeniniAutrans10.ppt.pdf. (Cited on page 122.)

[85] D. Madon, E. Sanchez, and S. Monnier. A Study of a Simultaneous Multithreaded Processor Implementation. In *International Euro-Par Conference*, pages 716–726, Toulouse, France, August 1999. (Cited on page 57.)

[86] H. Mair, A. Wang, G. Gammie, D. Scott, P. Royannez, S. Gururajarao, M. Chau, R. Lagerquist, L. Ho, M. Basude, N. Culp, A. Sadate, D. Wilson, F. Dahan, J. Song, B. Carlson, and U. Ko. A 65-nm Mobile Multimedia Applications Processor with an Adaptive Power Management Scheme to Compensate for Variations. *IEEE Symposium on VLSI Circuits*, pages 224 –225, june 2007. (Cited on page 14.)

[87] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins. Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture:A Case Study. In *Design, Automation and Test in Europe (DATE)*, Paris, France, February 2004. (Cited on page 19.)

[88] Mentor. ModelSim. http://www.model.com/. (Cited on page 49.)

[89] Bhuvan Middha, Anup Gangwar, Anshul Kumar, M. Balakrishnan, and Paolo Ienne. A Trimaran based framework for exploring the design space of VLIW ASIPs with coarse grain functional units. *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 2–7, 2002. (Cited on page 107.)

[90] MIPS. MIPS 1004K. http://www.mips.com/products/cores/32-64-bit-cores/mips32-1004k/. (Cited on pages 13 and 14.)

[91] MIPS. MIPS 24K. http://www.mips.com/products/cores/32-64-bit-cores/mips32-24k/. (Cited on pages 113 and 115.)

[92] MIPS. MIPS 34K. http://www.mips.com/products/cores/32-64-bit-cores/mips32-34k/. (Cited on page 15.)

[93] MIPS. Programming the MIPS32® 34K Core Family. Technical report, MIPS Technology, 2005. (Cited on pages 26 and 30.)

[94] MIPS. Single Chip Coherent Multiprocessing: The Next Big Step in Performance for Embedded Applications. Technical report, MIPS Technology, 2005. (Cited on page 34.)

[95] Takashi Miyamori. Venezia: a Scalable Multicore Subsystem for Multimedia Applications. http://www.mpsoc-forum.org/2008/slides/3-2 Miyamori.pdf, 2008. (Cited on page 15.)

[96] MOBILEYE. EyeQ2 and EyeQ3 vision system-on-chip. http://www.mobileye.com. (Cited on page 14.)

[97] Bren C. Mochocki, Kanishka Lahiri, Srihari Cadambi, and X. Sharon Hu. Signature-based workload estimation for mobile 3D graphics. In *Proceedings of the 43rd annual Design Automation Conference*, DAC '06, pages 592–597, New York, NY, USA, 2006. ACM. (Cited on page 8.)

[98] Simon W. Moore. *Multithreaded Processor Design.* Kluwer Academic Publishers, Norwell, MA, USA, 1996. (Cited on page 31.)

[99] R. Moussali, N. Ghanem, and M.A.R. Saghir. Microarchitectural Enhancements for Configurable Multi-Threaded Soft Processors. In *International Conference on Field Programmable Logic and Applications*, pages 782 –785, aug. 2007. (Cited on page 30.)

[100] Naveen Muralimanohar and Rajeev Balasubramonian. CACTI 6.0: A Tool to Model Large Caches. http://www.hpl.hp.com/techreports/2009/HPL-2009-85.html, 2009. (Cited on pages 40 and 86.)

[101] NetLogic Microsystems. XLR732 Processor. http://www.netlogicmicro.com/Products/ProductBriefs/MultiCore/XLR700.htm. (Cited on page 14.)

[102] Y. Nishikawa, M. Koibuchi, M. Yoshimi, K. Miura, and H. Amano. Performance Improvement Methodology for ClearSpeed's CSX600. In *ICPP'07: International Conference on Parallel Processing*, page 77, sept. 2007. (Cited on page 16.)

[103] Erik Norden. A Multithreaded RISC/DSP Processor with High Speed Interconnect. www.hotchips.org/archives/hc15/2_Mon/4.infineon.pdf, 2003. (Cited on page 31.)

[104] Daniel Nussbaum, Alexandra Fedorova, and Christopher Small. An overview of the sam cmt simulator kit. Technical report, Mountain View, CA, USA, 2004. (Cited on page 57.)

[105] Open SystemC Initiative. http://www.systemc.org. (Cited on page 49.)

[106] OpenMP. http://www.openmp.org. (Cited on pages 7 and 83.)

[107] M. Paganini. Nomadik: A Mobile Multimedia Application Processor Platform. In *ASP-DAC '07: Asia and South Pacific Design Automation Conference*, pages 749 –750, jan. 2007. (Cited on page 14.)

[108] PAR4ALL. http://hpc-project.com/pages/par4all.htm. (Cited on page 83.)

[109] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. LISA—machine description language for cycle-accurate models of programmable DSP architectures. In *DAC '99: Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 933–938, New York, NY, USA, 1999. ACM. (Cited on page 57.)

[110] Plurality. Hypercore processor. http://www.plurality.com/hypercore.html, 2009. (Cited on page 15.)

[111] Plurality. Plurality Ltd announces the worlds first scalable 256 multicore processor for wireless infrastructure. http://www.plurality.com/Plurality-Press-Release-wireless-ESC-26-April-2010.pdf, 2010. (Cited on page 16.)

[112] Valer Pop, Ruben de Francisco, H. Pflug, J. Santana, H. Visser, Ruud J. M. Vullers, Harmke de Groot, and Bert Gyselinckx. Human++: Wireless autonomous sensor technology for body area networks. In *ASPDAC'11: Proceedings of the 16th Asia and South Pacific Design Automation Conference*, pages 561–566, 2011. (Cited on page 6.)

[113] Wei Qin, Subramanian Rajagopalan, and Sharad Malik. A formal concurrency model based architecture description language for synthesis of software development tools. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 47–56, New York, NY, USA, 2004. ACM. (Cited on page 57.)

[114] M. Bartholomeu G. Araujo C. Araujo R. Azevedo, S. Rigo and E. Barros. The ArchC Architecture Description Language and Tools. *Parallel Programming*, 33(5):453–484, 2005. (Cited on pages 51, 57 and 58.)

[115] B.R. Rau, D.W.L. Yen, W. Yen, and R.A. Towle. The Cydra 5 departemental supercomputer: Design philosophies, decisions, and trade-offs. *IEEE Computers*, 22(1):12–34, January 1989. (Cited on pages 2 and 88.)

[116] J. Renau, K. Strauss, L. Ceze, W. Liu, S.R. Sarangi, J. Tuck, and J. Torrellas. Energy-Efficient Thread-Level Speculation. *IEEE Micro*, 26(1):80 –91, jan.-feb. 2006. (Cited on page 14.)

[117] Andrew Richardson. *WCDMA Design Handbook*. Cambridge University Press, New York, NY, USA, 2004. (Cited on page 68.)

[118] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. ArchC: a systemC-based architecture description language. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing SBAC-PAD 2004*, pages 66–73, 2004. (Cited on pages 58 and 61.)

[119] Sandro Rigo, Marcio Juliato, Rodolfo Azevedo, Guido Araújo, and Paulo Centoducatte. Teaching computer architecture using an architecture description language. In *WCAE '04: Proceedings of the 2004 workshop on Computer architecture education*, page 6, New York, NY, USA, 2004. ACM. (Cited on page 58.)

[120] M. W. Riley, J. D. Warnock, and D. F. Wendel. Cell Broadband Engine processor: Design and implementation. *IBM Journal of Research and Development*, 51(5):545 –557, sept. 2007. (Cited on pages 15 and 87.)

[121] N.J. Rohrer, M. Canada, E. Cohen, M. Ringler, M. Mayfield, P. Sandon, P. Kartschoke, J. Heaslip, J. Allen, P. McCormick, T. Pfluger, J. Zimmerman, C. Lichtenau, T. Werner, G. Salem, M. Ross, D. Appenzeller, and D. Thygesen. PowerPC 970 in 130 nm and 90 nm technologies. In *ISSCC'04: IEEE International Solid-State Circuits Conference*, pages 68 – 69 Vol.1, feb. 2004. (Cited on page 28.)

[122] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. Trace processors. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, pages 138–148, Washington, DC, USA, 1997. IEEE Computer Society. (Cited on page 27.)

[123] J. Schutz. A 3.3V 0.6 mu;m BiCMOS superscalar microprocessor. In *ISSCC: 41st IEEE International Solid-State Circuits Conference*, pages 202 –203, feb 1994. (Cited on page 13.)

[124] SEIKO-EPSON. Realoid SoC. http://www.epson.com. (Cited on page 14.)

[125] Semiconductor Industry Association. International Technology Roadmap for Semiconductors. www.itrs.net, 2003. (Cited on pages 40, 74 and 115.)

[126] O. Serres, A. Anbar, S. Merchant, and T. El-Ghazawi. Experiences with UPC on TILE-64 processor. In *IEEE Aerospace Conference*, pages 1 –9, march 2011. (Cited on page 13.)

[127] M. Shah, J. Barren, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, B. Sana, D. Sheahan, L. Spracklen, and A. Wynn. UltraSPARC T2: A highly-treaded, power-efficient, SPARC SOC. *ASSCC '07: IEEE Asian Solid-State Circuits Conference*, pages 22 –25, nov. 2007. (Cited on page 30.)

[128] J. J. Sharkey, D. Ponomarev, and K. Ghose. M-Sim: A Flexible, Multithreaded Architectural Simulation Environment. Number CS-TR-05-DP01, October 2005. (Cited on page 57.)

[129] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4.5):505 –521, july 2005. (Cited on page 29.)

[130] M. Sjalander, A. Terechko, and M. Duranton. A Look-Ahead Task Management Unit for Embedded Multi-Core Architectures. In *DSD '08: 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, pages 149 –157, sept. 2008. (Cited on page 8.)

[131] Angela Sodan, Jacob Machina, Arash Deshmeh, Kevin Macnaughton, and Bryan Esbaugh. Parallelism via Multithreaded and Multicore CPUs. *Computer*, 99(PrePrints), 2009. (Cited on page 9.)

[132] G.S. Sohi, S.E. Breach, and T.N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414 –425, june 1995. (Cited on page 27.)

[133] G.S. Sohi and A. Roth. Speculative multithreaded processors. *Computer*, 34(4):66 –73, apr 2001. (Cited on page 27.)

[134] G.P. Stein, E. Rushinek, G. Hayun, and A. Shashua. A Computer Vision System on a Chip: a case study from the automotive domain. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition - CVPR Workshops*, page 130, june 2005. (Cited on page 14.)

[135] SuperESCalar simulator. http://www.sesc.sourceforge.net/. (Cited on page 57.)

[136] S. Sutanthavibul and S.K. Perabala. First Intel Low-Cost IA Atom-based System-On-Chip for Nettop/Netbook. *ASQED'09: 1st Asia Symposium on Quality Electronic Design*, pages 88 –91, july 2009. (Cited on page 29.)

[137] Synopsys. DesignWare Universal DDR Memory and Protocol Controllers. http://www.synopsys.com/dw/ipdir.php?ds=dwc_ddr_universal_ctls. (Cited on page 100.)

[138] K. Tanaka. PRESTOR-1: a processor extending multithreaded architecture. In *Innovative Architecture for Future Generation High-Performance Processors and Systems*, page 8 pp., jan. 2005. (Cited on page 31.)

[139] Andrew S. Tanenbaum. *Structured Computer Organization*. Prentice Hall, 5 edition, 2006. (Cited on page 7.)

[140] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5 –25, jan. 2002. (Cited on page 6.)

[141] Tensilica. Xtensa LX processor. http://www.tensilica.com/. (Cited on page 14.)

[142] TILERA. TILE64. http://www.tilera.com/products/processors/TILE64. (Cited on pages 13, 117 and 122.)

[143] Hitoshi Toyoda. MIPS Multi-Threaded and Multi-core:. https://www.mips.jp/mips_jp/materials/PDFs/14thSCseminar_Toyoda.pdf, June 2010. (Cited on page 45.)

[144] Trimaran. http://www.trimaran.org/. (Cited on page 107.)

[145] Jenn-Yuan Tsai, Jian Huang, C. Amlo, D.J. Lilja, and Pen-Chung Yew. The superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9):881 –902, sep 1999. (Cited on page 27.)

[146] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392 –403, june 1995. (Cited on page 29.)

[147] Theo Ungerer, Bortu Robic, and Jurij Silc. Multithreaded Processors. *The Computer Journal*, 45:320–348, 2002. (Cited on pages 11 and 26.)

[148] Mark Utting, Mark Utting, Peter Kearney, and Peter Kearney. Pipeline specification of a mips r3000 cpu. Technical report, 1994. (Cited on page 59.)

[149] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. *SIGARCH Computer Architecture News*, 38:205–218, March 2010. (Cited on page 14.)

[150] N Ventroux. *Contrôle en ligne des systèmes multiprocesseurs hétérogènes embarqués: élaboration et validation d'une architecture*. PhD thesis, Université de Rennes 1 (MATISSE), September 2006. (Cited on pages 17, 18, 19 and 99.)

[151] N. Ventroux and R. David. The SCMP architecture: A Heterogeneous Multiprocessor System-on-Chip for Embedded Applications. *Eurasip*, 2009. (Cited on pages 3, 6, 9, 15 and 16.)

[152] N. Ventroux, A. Guerre, T. Sassolas, L. Moutaoukil, G. Blanc, C. Bechara, and R. David. SESAM: An MPSoC Simulation Environment for Dynamic Application Processing. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 1880–1886, Washington, DC, USA, 2010. IEEE Computer Society. (Cited on pages 48, 50, 51 and 53.)

[153] N. Ventroux, T. Sassolas, R. David, G. Blanc, A. Guerre, and C. Bechara. SESAM extension for fast MPSoC architectural exploration and dynamic streaming applications. In *VLSI System on Chip Conference (VLSI-SoC), 2010 18th IEEE/IFIP*, pages 341 –346, sept. 2010. (Cited on pages 48, 50, 51 and 53.)

[154] P. Viana, E. Barros, S. Rigo, R. Azevedo, and G. Araujo. Exploring memory hierarchy with ArchC. In *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing*, pages 2–9, 10–12 Nov. 2003. (Cited on page 58.)

[155] Ro Won W., Yi Jaeyoung, Park Joon-Sang, and Park Joonseok. Simultaneous thin-thread processors for low-power embedded systems. *IEICE Electronics Express*, 5(19):802–808, 2008. (Cited on page 29.)

[156] D.W. Wall. Limits of instruction-level parallelism. In *International Conference on Architectural Support for Programming Languages and OperatingSystems (ASPLOS)*, Santa Clara, USA, April 1991. (Cited on pages 5, 11 and 25.)

[157] Panit Watcharawitch and C Panit Watcharawitch. MulTEP: MultiThreaded Embedded Processors. In *International Symposium on Low-Power and High-Speed Chips (Cool Chips) IV*, 2003. (Cited on page 31.)

[158] W. Wolf, A.A. Jerraya, and G. Martin. Multiprocessor System-on-Chip (MPSoC) Technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1701 –1713, oct. 2008. (Cited on pages 6 and 9.)

[159] N. Yamasaki. Responsive Multithreaded Processor for distributed real-time control. In *AMC '04: The 8th IEEE International Workshop on Advanced Motion Control*, pages 457 – 462, march 2004. (Cited on page 29.)

[160] J. J. Yi and D. J. Lilja. Simulation of computer architectures: simulators, benchmarks, methodologies, and recommendations. *IEEE Transactions on Computers*, 55(3):268–280, March 2006. (Cited on page 48.)

Personal publications

International conferences

1. AICCSA2010 : Charly Bechara, Nicolas Ventroux, Daniel Etiemble, *Towards a Parameterizable Cycle-Accurate ISS in ArchC*, 2010 IEEE/ACS International Conference on Computer Systems and Applications (AICCSA) , pp.1-7, May 2010
2. ICESS2010 : Nicolas Ventroux, Alexandre Guerre, Tanguy Sassolas, Larbi Moutaoukil, Charly Bechara and Raphaël David, *SESAM : an MPSoC Simulation Environment for Dynamic Application Processing*, in the IEEE International Conference on Embedded Software and Systems (ICESS), Bradford, July 2010
3. VLSISOC2010 : Nicolas Ventroux, Tanguy Sassolas, Raphaël David, Guillaume Blanc, Alexandre Guerre, Charly Bechara, *SESAM extension for fast MPSoC architectural exploration and dynamic streaming applications*, in 18th IEEE/IFIP Conference on VLSI System on Chip (VLSI-SoC), Madrid, September 2010
4. RAPIDO2011 : Charly Bechara, Nicolas Ventroux, Daniel Etiemble, *A TLM-based Multithreaded Instruction Set Simulator for MPSoC Simulation Environment*, 3rd Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO 2011), Held in conjunction with the HiPEAC conference, January 2011
5. DSD2011 : Charly Bechara, Nicolas Ventroux, Daniel Etiemble, *Comparison of different thread scheduling strategies for Asymmetric Chip MultiThreading architectures in embedded systems*, in 14th Euromicro conference on Digital System Design (DSD 2011), Oulu, Finland, September 2011
6. MULTICORECHALLENGE2011 : Charly Bechara, Nicolas Ventroux, Daniel Etiemble, *AHDAM: an Asymmetric Homogeneous with Dynamic Allocator Manycore chip*, in Facing the Multicore Challenge II, Karlsruhe, Germany, September 2011
7. ICECS2011 : Charly Bechara, Aurélien Berhault, Nicolas Ventroux, Stéphane Chevobbe, Yves Lhuillier, Raphaël David, Daniel Etiemble, *A small footprint interleaved multithreaded processor for embedded systems*, in 18th IEEE International Conference on Electronics, Circuits and Systems (ICECS) 2011, Beirut, Lebanon, December 2011