l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

THALES

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Romain BRILLU

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'Université de Nantes
sous le label de l'Université de Nantes Angers Le Mans

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Electronique
Spécialité : Système Embarqué
Unité de recherche : Institut d'électronique et de télécommunications de Rennes (IETR)

Soutenue le 12/11/2014
Thèse n° : ED 503-XXX

Efficient design and programming of Multiple Processors System on Chip architectures

JURY

Président :	**M. Alain DARTE**, CNRS research director, Laboratoire de l'Informatique du Parallélisme
Rapporteurs :	**M. Frédéric PÉTROT**, Professor, École nationale supérieure d'informatique et de mathématiques appliquées de Grenoble
	M. Gilles SASSATELLI, CNRS research director, Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier
Examinateur :	**M. Jürgen TEICH**, Professor, University of Erlangen-Nuremberg
Directeur de thèse :	**M. Sébastien PILLEMENT**, Professor, Ecole polytechnique de l'Université de Nantes
Co-directeur de thèse :	**M. Fabrice LEMONNIER**, Research engineer, Thales Research & Technology
Encadrant :	**M. Philippe MILLET**, Research engineer, Thales Research & Technology

À mes papis Pierre et Henri.

Remerciements

Ah les remerciements, dernière étape dans la rédaction de cette thèse qui peut paraître simple mais qui n'en est rien, bien au contraire car résumer en quelques lignes tout ce que vous m'avez apporté au cours de ces trois années est difficile et le plus dur pour moi reste l'impair d'oublier quelqu'un. Si tel est le cas que ces personnes me pardonnent.

Je souhaite avant tout remercier l'ensemble des membres de mon jury, Monsieur Alain Darte pour avoir accepté d'en être le président, Messieurs Frédéric Pétrot et Gilles Sassatelli pour avoir accepté d'en être les rapporteurs et Monsieur Jürgen Teich pour avoir accepté d'en être l'examinateur.

Ensuite comment ne pas citer mon directeur de thèse Sébastien Pillement sans qui cette thèse n'aurait pas été à son terme. Merci pour le soutien, la confiance apportée et pour m'avoir aidé à développer et à formaliser mes idées afin que je puisse mener à bien cette thèse. Comme tu me l'a si bien dit au début de cette aventure "La recherche est avant tout un échange" et ce fût un véritable plaisir d'échanger avec toi tout au long de ces trois années aussi bien professionnellement que personnellement.

Je remercie également mon co-directeur de thèse Fabrice Lemonnier pour m'avoir permis de faire cette thèse et m'avoir accordé une grande confiance et liberté dans le développement et la réalisation de cette dernière.

Un clin d'oeil à Frédéric Falzon pour notre collaboration et à mon encadrant Philippe Millet pour les conseils apportés au cours de ce doctorat.

Merci au Spearou, Michel, Rémi, Paul et Teodora pour leur sympathie, la patience, le temps qu'ils m'ont accordés et pour avoir été présents pour moi dès que j'avais un problème ou des questions.

Ces remerciements ne seraient pas complets si je ne remercie pas ici Christophe pour l'ensemble des connaissances qu'il m'a transmises, le temps et la confiance qu'il m'a accordé et pour la patience qu'il a eu mon égard. Sache que venir à Thales sans nos échanges forts poétiques que ce soit dans les transports ou au bureau n'avaient plus la même saveur.

Au rayon des poètes comment ne pas penser à Lionel pour sa gentillesse, sa disponibilité, sa bonne humeur et son sens inné de la dédramatisation. Merci pour l'aide que tu a pu m'apporter aux différentes étapes de cette thèse et pour les moments partagés.

Je remercie aussi Hai le Zidane vietnamien pour cette année commune passée à Lannion, pour ces parties de baby-foot engagées (Mais tu es quand même particulièrement mauvais) et pour tous les fou rires que nous avons eu. Au plaisir de te revoir dans la magnifique capitale de la Bretagne qu'est Rennes. Cette dernière phrase est particulièrement destinée aux Nantais, je préfère préciser au cas où ils n'auraient pas compris ce qui n'aurait rien d'étonnant.

Vu qu'on parle des Nantais je remercie l'ensemble des membres de l'IETR Nantes. J'ouvre néanmoins ici un aparté destiné aux Nantais récalcitrant à voir la vérité en face (Ils se reconnaîtront) pour leur signaler que le R de IETR signifie Rennes. Ce qui démontre ce qui ne devrait plus être à démontrer la supériorité manifeste de la capitale Bretonne sur sa voisine Française et leur incohérence à vivement critiquer cette magnifique ville et ses clubs sportifs.

Maintenant que ce point à été clarifié je remercie particulièrement au sein de l'IETR la team café pour l'ensemble des bons moments partagés et des conseils plus ou moins pertinents distillés si et là. Distinction particulière à Loïc et Yann pour leurs recettes culinaires. En effet la recette du gâteau cramé sur le dessus et pas cuit au centre et la recette du gâteau cuit que sur les extrémités m'étaient inconnu avant leur rencontre.

Je remercie aussi tous les membres de la team billard pour m'avoir permis de croire que j'étais bon à ce jeu. Merci particulier à Guillaume pour cette victoire écrasante que nous avons connu au tournoi.

Merci à la team football pour l'ensemble des matchs disputés même si cela m'a valu quelques courbatures et une blessure à une cuisse. Les matchs Chine contre le reste du monde vont me manquer.

Je remercie aussi le Justin Bieber indien Chagun pour ces trois années de thèse partagées. Merci pour ta zenitude (qui peut quand même être stressante par moment), ta bonne humeur et ta joie de vivre. Je te remercie moins pour m'avoir arraché quelques larmes en me faisant découvrir la cuisine indienne.

Un grand merci a Joëlle et Sandrine pour leur bonne humeur et disponibilité, vous m'avez simplifié la vie et permis de me concentrer pleinement sur mes travaux.

Afin de n'oublier personne je remercie Philippe Bonnot, Olivier Sentieys, Jean-François Diouris et l'ensemble de leurs équipes pour m'avoir accueilli au sein du laboratoire des systèmes embarqués de Thales Research & Technology, du laboratoire de l'IRISA de Lannion et du laboratoire de l'IETR de Nantes.

En tant que fervent supporter du plus grand club de l'ouest je ne peux que remercier le Stade Rennais FC pour la sublime défaite infligée aux Nantais le dimanche 23 février 2014 qui m'a apporté courage et inspiration dans la phase de rédaction. J'ai une attention particulière pour Ola, Anders et Paul-Georges et leurs trois magnifiques buts plantés en ce beau dimanche après-midi.

Comme le dit si bien le dicton il faut garder les meilleurs pour la fin et donc ma famille pour avoir été toujours derrière moi.

Merci à ma mamie Marie pour s'etre occupé de moi lors de mes jeunes années.

Je remercie challeureusement ma mamie Émile pour avoir veillé sur moi depuis que je suis tout petit et pour m'avoir soutenue jusqu'au bout dans cette thèse alors que ta santé ne te le permettait pas. Je te confirme aussi que j'avais bien besoin d'un doctorat pour faire les fameux "roulé boulé".

Merci à ma filleule et nièce Gaëlla pour avoir su me détendre et me faire rire que ce soit par des tours en LEGO, des dessins, des sorties en vélo, des balades aux parc ou des chants.

Merci à mes soeurs Hélène et Élise pour m'avoir permis chacune à leur manière d'en être là aujourd'hui, pour tous les souvenirs et moments partagées et pour avoir toujours été à mes cotés que ce soit dans les bons ou les mauvais moments.

Un grand merci à ma femme Aurélie pour avoir faire preuve d'une grande patience à mon égard, su me soutenir dans les moments difficiles, m'avoir remonté le moral, fait rire et avoir fait en sorte que je garde le sourire tout au long de cette thèse.

J'adresse une très grande reconnaissance et un très grand merci a mes parents Michel et Paulette. Énumérer la liste de tout ce vous avez fait pour moi en quelques lignes est impossible donc merci pour tout. Si j'en suis là aujourd'hui c'est avant tout grâce à vous et ce doctorat est aussi le votre.

Pour finir je remercie avec émotion "le grand" à qui j'avais promis avant qu'il nous quitte de mener à bien ce doctorat et dont la mémoire m'a permis de surmonter bien des difficultés aux cours de ces trois années.

Romain

Contents

List of Figures

List of Tables

Résumé en français

Contexte et motivations

Problématiques

Les applications embarquées incorporent de plus en plus de fonctionnalités impliquant différents types de traitements à réaliser. L'impact majeur de cette demande est l'évolution croissante des systèmes embarqués que cela soit en termes de performances et de capacité mémoire.

De plus, aujourd'hui les applications deviennent de plus en plus dynamiques. Les temps d'exécution sont alors dépendants de paramètres non prédictibles au moment de la définition de l'application. La détermination du temps d'exécution de l'application devient dès lors impossible, et seule une borne supérieure pour le temps d'exécution peut être définie.

L'évolution des systèmes embarqués entraîne donc des problèmes au niveau de la conception et de la programmation. Ces systèmes doivent en effet trouver un compromis entre leurs capacités (puissance de calcul, dynamicité) et les contraintes du domaine d'application (silicium, consommation).

La problématique, liée à la puissance de calcul, fut la première à être adressée. Dans le cas de processeur monolithique, les principales approches consistent à augmenter la fréquence de fonctionnement du cœur de calcul ou à augmenter la capacité mémoire pour limiter le nombre de cycles inutilisés du processeur. Cependant ces techniques, bien qu'efficaces, augmentent dans le même temps la surface du cœur de calcul le de petits cœurs permet d'atteindre la puissance de calcul d'un seul cœur monolithique tout en réduisant la consommation énergétique et la surface consommée.

Grâce à cette évolution, les architectures MPSoC apparaissent actuellement comme les principaux promoteurs de la révolution industrielle des semi-conducteurs [212]. Ces plates-formes contiennent plusieurs processeurs, généralement hétérogènes, des éléments de traitement avec des fonctionnalités spécifiques reflétant la nécessité du domaine d'application prévu, une hiérarchie mémoire et des composants d'entrées/sorties. Tous ces éléments étant liés les uns aux autres par une interconnexion sur puce (de plus en plus souvent un réseau sur puce ou "NoC").

Grâce à leurs évolutivités, leurs hautes performances, leurs capacités de parallélisme à un très haut niveau d'intégration et leur enveloppe énergétique restreinte ces architectures deviennent de plus en plus populaires. Elles répondent aux besoins de performances des applications multimédia, des architectures de télécommunication, de la sécurité du réseau et de nombreux autres domaines d'application. L'industrie est elle aussi consciente de la nécessité d'utiliser des architectures "MPSoC" dans le but d'augmenter le rapport performance - énergie des systèmes embarqués où les contraintes de consomma-

tion sont plus élevées [138].

Cependant, la conception d'une architecture "MPSoC" faible consommation et supportant les performances requises, n'est pas aisée. Cet équilibre dépend en effet de nombreux paramètres tels que le nombre de cœurs de calcul, l'enveloppe énergétique globale, le type de réseau d'interconnexion, l'architecture de la hiérarchie mémoire, le déploiement de l'application sur le système. En outre, le coût de fabrication de ce type de plateforme est conséquent (surtout dans les technologies modernes) et implique la vérification de manière précise de chaque choix architectural et applicatif.

Le problème de la consommation d'énergie dans les architectures MPSoC vient du fait que l'enveloppe énergétique n'a pas suivi la même évolution que le nombre de cœurs [50]. Les solutions, afin de réduire la consommation d'énergie, ont été pleinement étudiées [86], l'une des principales approches afin de réduire la consommation étant le "voltage scaling".

Les accès mémoires sont aussi un des facteurs critiques de la performance des architectures MPSoC [189], les gains en terme de puissance de calcul outrepassent fortement ceux de la mémoire. En effet, si la puissance de calcul double tous les deux ans, celui de la mémoire double tous les six ans [111, 149]. En conséquence, les accès à la mémoire génère des délais important pour lire ou écrire les données vis-à-vis des temps de calcul.

Dans les architectures MPSoC actuelles, les deux architectures principalement utilisés sont les modèles à mémoire partagée et à mémoire distribuée. Cependant, le choix entre ces deux modèles est complexe car il dépend à la fois de l'application, du réseau d'interconnexion et de la puissance de calcul des cœurs.

Le choix d'un réseau d'interconnexion est un autre point crucial au moment de la définition d'une architecture MPSoC. En effet, le choix d'une topologie est dépendant du nombre de cœurs qui doivent être connectés sur le réseau mais aussi des caractéristiques de l'application. Dans le domaine des systèmes embarqués, trois principales familles pour les réseaux d'interconnexions existent: (1) Les connections point à point, (2) les bus et (3) les NoC.

Une fois, l'architecture MPSoC déterminée, la difficulté réside dans le fait de tirer pleinement parti de cette dernière. À cette fin, le placement d'applications sur des architectures MPSoC à été fortement étudié dans la littérature [175]. En fonction du moment ou sont assignées les tâches sur les coeurs de calcul, les techniques de placement de tâches sont statiques ou dynamiques.

Dans le cas de placement de tâche dynamique, l'assignement et l'ordonnancement des tâches sur les processeurs sont réalisés durant l'exécution de l'application. Le placement de tâche dynamique essaye donc de toujours trouver les goulots d'étranglement de la performance et de répartir la charge de calcul sur l'ensemble des processeurs.

Dans le cas de placement de tâche statique, le placement des tâches sur les cœurs de calcul est réalisé hors-ligne, avant l'exécution de l'application sur l'architecture. Pour une application donnée et une infrastructure de communication déterminée, les algorithmes essayent de trouver le meilleur placement de tâches au moment de la conception de l'architecture.

Finalement, un des derniers défis lors de la conception d'une architecture MPSoC est le placement des données. Le placement des données est en effet un point clé afin d'être en mesure d'atteindre des hautes performances de calcul et d'avoir une utilisation efficace des ressources matérielles. Comme le placement des données est à la fois dépendant du réseau d'interconnexion, de la taille et de la bande passante des mémoires ainsi que de l'efficacité du processeur, le placement des données à un impact

fort sur le placement des tâches et sur les choix architecturaux. Un placement de données idéal est un placement de données ou le temps de traitement des données est supérieur où au moins égal au temps de transfert des données.

Comme nous pouvons le voir, les architectures MPSoC soulèvent des défis importants que ce soit au niveau de la conception ou de la programmation. De plus, étant donné que ces paramètres (Nombre de cœurs, hiérarchie et taille mémoire, réseau d'interconnexion, placement des tâches et des données) s'influencent mutuellement l'exploration de l'espace de conception des architectures MPSoC devient difficile.

En effet choisir le nombre de cœurs d'une architecture MPSoC doit être fait au regard du parallélisme de l'application mais aussi vis-à-vis de la bande passante de la mémoire et du réseau d'interconnexion. Cependant, le parallélisme pouvant être atteint par l'application est décidé en fonction du placement des tâches et des données, des tailles mémoires et des caractéristiques inhérentes de l'application.

Il apparaît clairement que la modification de l'application ou de l'un des paramètres caractéristiques de l'architecture entraîne la réévaluation complète de la solution. Ce qui dans les premières phases de développements d'une architecture MPSoC est prohibitif et consommateur en terme de temps.

De plus, ces défis de conception deviennent de plus en plus vrai avec l'émergence des architectures MPSoC hétérogène. En effet, ces architectures ne se contentent plus de répliquer plusieurs fois le même cœur de calcul mais incluent des fonctionnalités spécifiques ("Intellectual Property" (IP)) dédiées à un domaine d'application particulier, car elles présentent un meilleur niveau de performance et une meilleure efficacité énergétique [138].

Cette hétérogénéité en dépit des bénéfices apportés augmente la difficulté de concevoir et programmer les architectures MPSoC. En effet, l'architecte doit: (1) Choisir entre plusieurs types de cœurs de calcul, (2) Décider s'il est plus bénéfique de placer des tâches sur des GPP ou des accélérateurs matériels.

La construction de l'architecture est aussi complètement différente étant donnée que l'hétérogénéité de chaque composant doit être abstrait au niveau du réseau d'interconnexion. Ce qui implique de définir entre chaque cœur de calcul et le réseau d'interconnexion un "wrapper" particulier. Le "wrapper" devant être redéfini chaque fois que le réseau d'interconnexion est changé.

En outre, la programmation des architectures MPSoC hétérogènes est difficile étant donné que la manière de programmer un accélérateur matériel ou un processeur généraliste (GPP) est différent. L'utilisateur doit prendre pleinement parti de ces différences afin d'utiliser au maximum ces architectures. Ainsi, les temps de développement évoluent de manière exponentielle ainsi que les temps de mise sur le marché.

Tous ces défis durant la conception des architectures MPSoC mettent en lumière le besoin d'une méthodologie d'exploration d'espace de conception aidant l'utilisateur à définir et à programmer ces architectures. Il devient alors nécessaire de définir un outil d'exploration d'espace de conception (DSE) facilitant l'interaction entre l'application et l'architecture pour réduire efficacement le nombre de solutions s'offrant à l'utilisateur et les temps de mise sur le marché.

De plus comme les architectures MPSoC deviennent de plus en plus hétérogènes, il est nécessaire de développer des modules matériels permettant de faciliter la construction et la programmation des architectures MPSoC hétérogènes.

Contributions

Dans le cadre de cette thèse, notre contribution est la définition d'une méthodologie d'exploration d'espace de conception. Cette méthodologie DSE a pour but de définir à la fois une architecture matérielle et son code binaire exécutable à partir de trois entrées: (1) le code C séquentiel d'une application, (2) une librairie d'architectures, (3) un fichier de contraintes.

Les principales caractéristiques de notre méthodologie DSE est la considération de manière conjointe de toutes les contraintes relatives à la définition d'une architecture matérielle grâce à une collaboration avec l'utilisateur. La méthodologie DSE proposée est en effet capable de: (1) générer une architecture composée de plusieurs cœurs de calcul, (2) définir un placement des tâches et des données ainsi que l'ordonancement associées à partir des contraintes utilisateurs.

Étant donné que ces paramètres s'influencent mutuellement, un outil nommé "Parsimonious ARchitecture Space Exploration" (PARSE) est proposé afin de parcourir l'espace de solution. PARSE est construit autour de mécanismes récursifs permettant depuis des simulations d'adapter l'architecture, le placement des tâches et des données, ceci dans le but d'atteindre le meilleur compromis à la fois pour l'architecture et pour l'application. Pour ce faire la méthodologie de PARSE est semi-automatisée.

En effet, étant donné la taille de l'espace de solution, l'utilisateur aide PARSE en retirant de l'espace de solution l'ensemble des cœurs ne répondant pas aux besoins applicatifs.

PARSE, à partir des choix de l'utilisateur, génère un ensemble d'architectures candidates, accompagnées d'un placemement des tâches et des données. Chaque solution potentielle est évaluée par le biais d'un simulateur SystemC afin de converger de manière efficace vers le meilleur compromis à la fois pour l'application et l'architecture. Cette exploration de l'espace de solution s'effectue de manière automatisée grâce à des algorithmes évolutionnaires. Une fois le meilleur compromis identifié le code VHDL de l'architecture et le code binaire exécutable associés sont générés.

Parce que PARSE à la capacité de générer des architectures matérielles, notre seconde contribution est la définition de deux modules matériels. Le premier module matériel défini une unité de management mémoire (MMU) servant à abstraire la hiérarchie mémoire aux sein d'architectures organisées autour de clusters.

La définition de cette MMU permet de faciliter la programmation des architectures MPSoC et d'optimiser le temps d'exécution de l'application, en réduisant les goulots d'étranglement, en maintenant la localité des données et en limitant les transferts de données à travers la plateforme. Pour ce faire, l'architecture proposée est basée sur des clusters où les processeurs sont regroupés et connectés au travers d'une mémoire partagée. La cohérence de la mémoire étant assurée par le biais d'une MMU matériel.

Le second module matériel défini dans le cadre de cette thèse est l'"accelerator interface" (AI) qui est une interface générique utilisée pour connecter différents types de cœurs de calcul (GPP, DSP[1] ou accélérateurs matériels) sur un réseau d'interconnexion.

L'AI permet de connecter de manière efficace différents types d'accélérateurs et de processeurs sur un même réseau d'interconnexion sans tenir compte de leurs spécificités ou caractéristiques. L'AI propose une apprroche "plug and play", grâce à la définition d'un modèle d'exécution unifié et d'une architecture générique qui permet de ne pas tenir compte du cœur de calcul adressé et de ses spécificités.

1. digital signal processor

De plus, toujours dans le but de tirer pleinement parti de la puissance de calcul induite par les architectures MPSoC, l'AI est autonome une fois initialisé, ce qui permet de réaliser de manière concurrente des traitements en parallèle.

Le reste de ce document présente en chapitre 1 le contexte et les motivations derrières cette thèse. Le chapitre 2 présente l'état de l'art des différentes architectures MPSoC et des outils d'exploration d'espace de conception. Le chapitre 3 quant à lui décrit la méthodologie d'exploration d'espace de conception proposé permettant d'explorer de manière conjointe les aspects architecturaux et applicatifs. Le chapitre 4 présente les contributions apportées au niveau matériels. Pour finir le chapitre 5 dressent les conclusion et les perspectives de cette thèse.

1

Introduction

Abstract: This introductory chapter first presents the context and the motivations behind this thesis. Finally the manuscript organization is given at the end of the chapter.

Contents

1.1 Context and motivations

1.1.1 Problem statement

The embedded applications come up with more and more functionalities inducing various kinds of computations to realize. The major impact of these new application needs is the steadily evolution of the embedded systems performances in terms of computing power and memory capacity. The Figure 1.1 [107] shows the computing power needs depending on the application for various application domains.

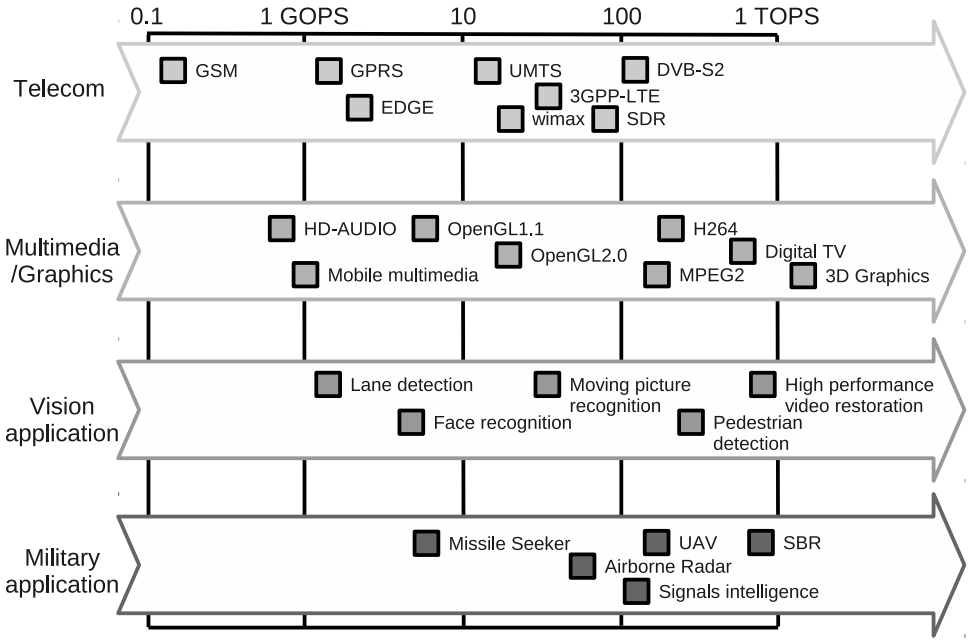


Figure 1.1 – The graph shows the computational requirements of different application grouped by application domains [107].

For example in the telecom domain the application computing power needs to increase to get around 100 giga operation per second (GOPS) for the video broadcasting applications. The same trend is also observed for the multimedia applications, due to the image size and to the algorithm complexity increase. For example the 3D graphics applications require more than 1 tera operation per second (TOPS). The same evolution of the computing power affects all the embedded applications domains whether for the vision applications or the military applications.

Moreover today an embedded system has to handle several kinds of applications at the same time on the same chip. For example a mobile is no longer just used as a phone, but also used to read mails, go on internet, watch movies or play video games. These demands impose the embedded systems to run several applications which come from several applications domains.

Furthermore the applications become more and more dynamic. Indeed the application executions time is dependent of not predictable parameters at the time the program is written. It is then not possible to predict the execution time of an application and it is usually only possible to determine an upper bound for the processing time. This dynamicity is more and more present in the graphic domain where

the algorithms adapt their computation to the data to be processed, as DVS [1] for 3D video games [106] for example. This application dynamicity implies that no optimal scheduling solution can be found off-line, and so the system control as to also be dynamic in order to enable an on-line optimization.

The embedded system evolution leads to a problem at the conception level. Indeed these systems have to find a trade-off between their capacity (computing power, dynamicity) and the embedded system constraints (silicium, consumption). The computing power is one of the major issues.

In order to increase the computing power of a single core architecture the solution is to improve its operation frequency or to raise the size of the memory in order to limit the idle cycle of the processor. These techniques increase, at the same time, the silicium area and the processor power consumption, which then decrease the energy efficiency [50].

Since energy efficiency is a key aspect of an embedded system, the solution is then to use a multi or a manycore architecture. Indeed as stated by [50] it is easier to integrate several little core specialized or not, whose energy and area efficiency are optimized. This allows to reach a computing power equal to the one of a single processor within a reduced power and area budget.

Following this trends, the multiple processors system on chip (MPSoC) architectures (Figure 1.2) appears as a major promoter of the industrial revolution of semiconductors as advised by the international technology road-map for semi-conductors (ITRS) [212]. This relies on the integration on the same chip of several complex functionalities. These platforms contain multiple processors, a memory hierarchy, and I/Os components. All these elements linked to each other by the means of an interconnect infrastructure becoming more and more often a network on chip (NoC).

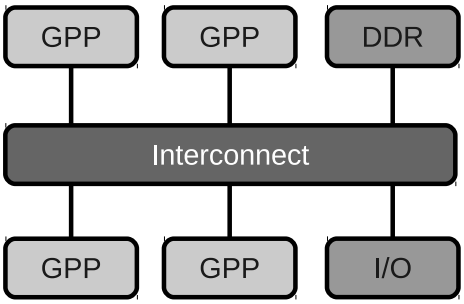


Figure 1.2 – Typical MPSoC architecture based on several GPP connected through an interconnect (Bus or Noc based) along with a DDR memory and all the peripherals needed for a connection with the outside environment. These MPSoC is homogeneous and can target several kinds of computation.

Due to their performance, MPSoC gain popularity, and are now present in various domains of applications. Industry is aware of the need of using multi-core and shortly manycore chips to raise the performance/power ratio especially in embedded systems when power consumption is one of the main constraints [138].

However the design and the programming of a high performance low power MPSoC architecture is not easy. Having several cores on a single chip raises numerous problems and challenges. Power and temperature management are two concerns that can increase exponentially with the addition of cores. Memory hierarchy and data placement is another challenge, in order to avoid any bottleneck at the

1. Dynamic Voltage Scaling

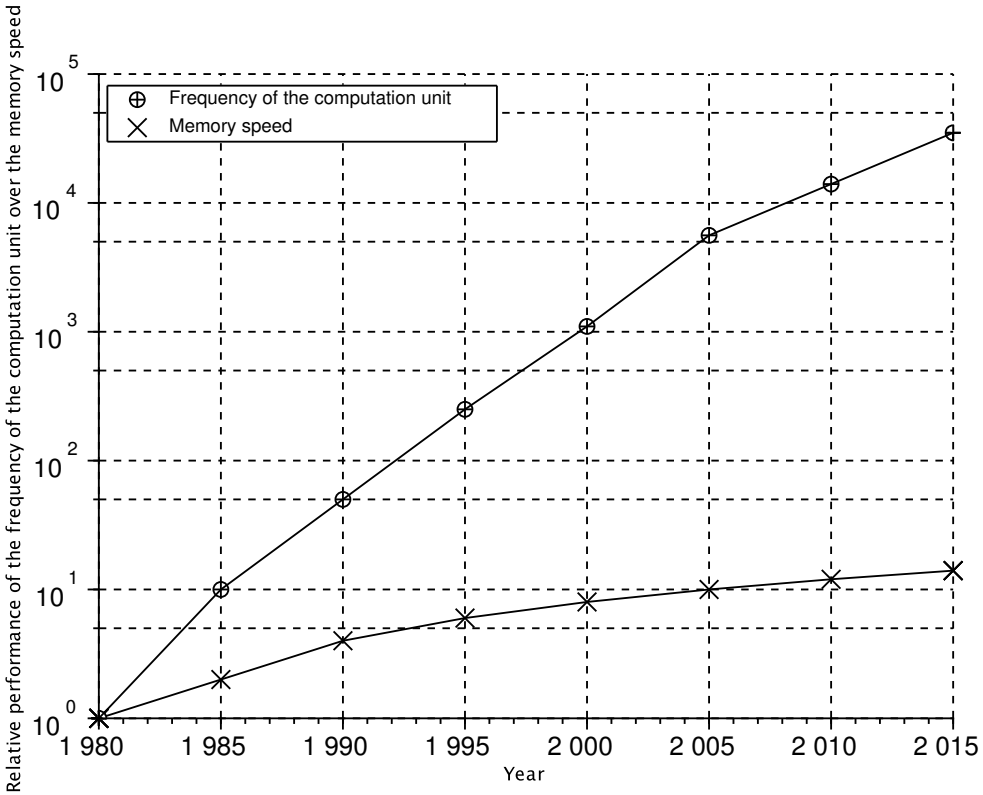


Figure 1.3 – Evolution of the memory performance in regards of the processor performance [111].

memory port and onto the interconnect. The choice of an interconnection network is also deterministic in order to be scalable with the number of cores and to sustain the bandwidth request. Finally, using a MPSoC to its full potential is another issue, if programmers don't write applications that take full advantage of the multiple cores.

The power consumption problem known as the "Power Wall", is due to the fact that the power envelope does not follow the same trend as the number of cores [50]. In order to overcome this "Power Wall" a set of initiative has been proposed [86].

Memory is also a critical factor for overall performance of MPSoC architectures [189], the gain in speed of calculation units far exceeds memory ones. As shown (Figure 1.3), if the speed of calculation units doubles every two years, memory one doubles every six years [111, 149].

In consequence the access time to the memory usually causes important access time to read or write data, compared to the processing time. This divergence in performance between the memories and computation units is called "memory wall". In current MPSoC architectures two main memory models are used depending of the architecture and of the running application:

- The shared memory (SHMEM) model: where a memory is simultaneously accessed by multiple processors for data transfer and for synchronization.

- The distributed memory model: which combines each core with its own local memory (LMEM) and there is no global memory or address space.

A shared memory model eases the programming of the platform since all the processors share the same view of the architectures. However over a certain number of cores a physically shared memory leads to bottlenecks onto the network and at the memory ports. These bottlenecks then limit the scalability of the memory model in context of MPSoC architectures.

The distributed memory model on the other hand is much more scalable. However the difficulty to program this model is increased. Indeed the programmer has to think about the processors synchronizations, the data location and transfers in order to ensure the consistency of the application execution time. To overcome these programming difficulties one possible solution relies on the use of a virtualization layer to expose at the software level a shared memory space.

However choosing between those two memory models is then quite difficult since the choice is fully dependent of the application, the interconnect bandwidth and of the processing cores capabilities.

The choice of an interconnection network is another crucial point when defining a MPSoC architecture. Indeed the choice of a topology is dependent of the number of cores that have to be connected to the interconnect and is also dependent of the application characteristics. In the embedded system field three mains on-chip interconnection families exist: (1) the point to point connection, (2) the bus and (3) the NoC.

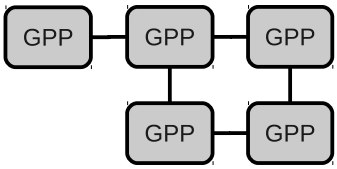


Figure 1.4 – Point to point interconnection network

The point to point connection creates a dedicated communication link between each element to interconnect (Figure 1.4). Thus, when a communication must be made, the sender places the message on the dedicated link and the receiver returns a confirmation of reception. These kinds of connection have the advantages of being extremely efficient since no communication link is limited. However this solution is not scalable. Indeed the number of links between the cores quickly becomes significant and the consumed silicium area unacceptable.

Unlike point to point interconnect, bus based architecture seeks to share communication resources (Figure 1.5). A bus architecture is composed of two major elements an arbitrator and a communication link. So when a core wants to send a message over the bus, it first consults the arbitrator. Once it has received the authorization, it sends the message onto the bus. At all times there is one message on the communication medium, and several arbitration policies exists depending of the needs of the user (FIFO [2], TDMA [3], Priority levels ...).

The main advantage of the bus is its simplicity and the fact that only one link is used to connect all the elements between each other. However over a certain number of cores the scalability of the bus is

2. first in first out
3. time division multiple access

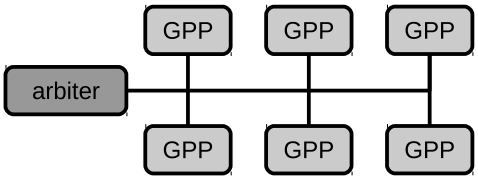


Figure 1.5 – Bus interconnection network

limited due to the latency needed to access the bus.

Due to the limitations of bus and point-to-point interconnection network, respectively regarding the performance, area and scalability, the NoC were proposed. NoC are derived from network between computers. The basic idea is to transfer the information over the network in the form of messages divided in packets. As Ethernet networks, the NoC consist of a set of links and routers enabling communications between the elements (Figure 1.6).

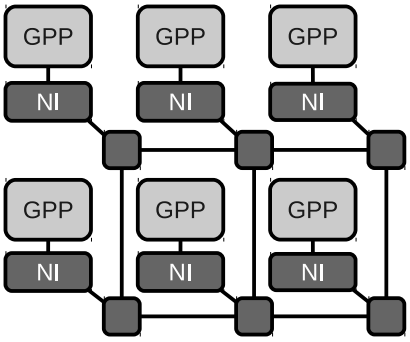


Figure 1.6 – Representation of a NoC

A network interface (NI) is placed between the core and the router to format the message and managed the communication. This modularity allows the addition of new communicating elements without having a significant impact on performance as for the bus. Furthermore, the addition of elements is done with a minimum of links which limits its cost unlike point to point connections. The main difficulty with a NoC relies in the routing policies and the way to ensure a certain quality of services [91].

Once the architecture of the MPSoC determined, the difficulty is to take full advantage of the multiple cores present onto the chip. To that end application mapping is studied by the state of the art [175]. The problem of application mapping is NP-hard [166]. Depending on the time at which the tasks are assigned to the computing cores for processing, the mapping techniques can be classified as dynamic or static mapping.

In case of on-line or dynamic mapping, the assignment and ordering of tasks are performed during the execution of the application. Dynamic mapping always tries to detect the performance bottlenecks and distribute the workload among the processors. As the mapping depends on the current load of the processors, it should result in a better solution. However, the computational overhead of the mapping algorithm may increase the delay and energy consumption of the application at run-time.

On the other hand, in case of static mapping, the mapping of tasks is performed off-line, before

the application runs. For a given application and a target communication infrastructure, static mapping always tries to define the best placement of tasks at design time. As the mapping is completed before execution, the mapping algorithm is executed only once and no overhead implies at run-time.

Finally another challenge when programming a MPSoC architecture is the data placement. The data placement defines based on the memory hierarchy where the data needed by processors are located, how the processors access it and the data transfers needed to ensure the memory consistency and the performance. The data placement is a key point to reach both high performances and to define hardware efficient architectures. Since the data placement is dependent of the interconnect, of the memory size and bandwidth and of the processor efficiency, the data placement has a big impact on the architecture and on the application definition.

An ideal data placement being a placement where the data transfers are minimal and where the time needed to process the data is greater or at least equal to the time needed to transfer them.

The MPSoC architectures, raise significant design and programming challenges in order to take full advantages of their capabilities. Moreover since these parameters (numbers of cores, memory hierarchy and size, interconnect, task mapping, data placement) mutually influence each other the design space exploration (DSE) of MPSoC architecture is becoming cumbersome.

Indeed choosing the numbers of cores of a MPSoC, must be done with regards to the application parallelism but also of the memory and interconnect bandwidth. However the application parallelism that can be achieved is decided in regards of the data placement, the task mapping, the memory size and of the inherent characteristics of the application.

Table 1.1 – Influence of the MPSoC architectures characteristics parameters.

			Impacted parameters					
			Mapping	Data placement	Nb of cores	Type of cores	Interco bw	Mem bw
Impacting parameters	Archi	Interco hierarchy	✓		✓		✓	✓
		Mem hierarchy		✓			✓	✓
		Nb of cores	✓			✓		
	Appli	Type of cores			✓			
		Mapping		✓				
		Interco bw	✓	✓				
		Mem bw	✓	✓				

As it can be seen in table 1.1, it clearly appears that a modification of the application properties or of one of the characteristic parameters of the architecture will force the designer to reevaluate its entire solution. This activity done during the early stages of development of a MPSoC architecture is long and time consuming.

Furthermore these design challenges are becoming more complicated with the emergence of heterogeneous MPSoC architectures (Figure 1.7). Indeed these architectures do not simply replicate several times the same core, but includes specific features reflecting the need of the intended domain of application, because they present better performances and are more power efficient [138].

This heterogeneity despite the performance benefit brought increases the difficulty to design and program MPSoC architectures. Indeed the designer has to: (1) choose between several types of cores, (2) decide if it is more beneficial to map a set of tasks to a general purpose processor (GPP) or to a hardware accelerator (Intellectual Property (IP)).

The construction of the architecture is also more difficult since the heterogeneity of each core have to be hidden at the interconnect level. This implies to create between each IP and the interconnect a particular wrapper. The wrapper has to be redefined each time the interconnect is changed.

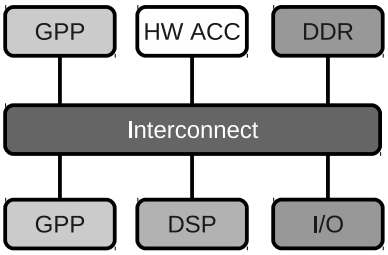


Figure 1.7 – Typical heterogeneous MPSoC architecture.

Finally with heterogeneous architectures, the programming is also more difficult since the way to program a hardware accelerator or a GPP is quite different. Since the user has to take full advantages of these differences in order to fully use its architecture, the developing time is getting more important along with the time to market.

All these challenges during the definition of MPSoC architectures emphasize the needs of an automatic design process to help the user to design and program these architectures. It is then mandatory to define a DSE methodology which aims to define jointly the application and the architecture, to efficiently reduce the time to market and to prune the solution space by providing assistance to the designer. To that end the DSE methodology should be able to handle the:

- Data placement exploration to guide the user to the most efficient data placement.
- Mapping exploration to guide the user to the most efficient task placement.
- Architecture exploration to guide the user in the architecture creation process.
- Heterogeneity of MPSoC architectures.
- Binary code generation from an architecture representation and an application description.
- VHDL code generation from an architecture representation or an application C code.
- Scalability of MPSoC architectures by integrating new architectural or application models into its database.
- Performance evaluation by evaluating the performance of the application onto the architecture.

Moreover since the MPSoC architectures are now becoming more and more heterogeneous it is necessary to develop hardware modules than can connect different cores together and address different applications domains at the same time, in order to ease the definition and the programming of heterogeneous MPSoC architectures.

1.1.2 Contributions

In the context of this thesis our contributions is the definition of a DSE methodology. This DSE methodology aims to define a hardware architecture and its associated executable binary code based on three inputs: (1) an application C code, (2) an architecture library and (3) a constraints file. It is however important to note that the proposed DSE methodology can also start with an already existing architecture and define the associated executable binary code.

The main features of our DSE methodology is the consideration of all the constraints related to the definition of hardware architectures thanks to a collaboration with the user. The proposed methodology is indeed able to: (1) generate an architecture composed of several processing cores potentially heterogeneous, (2) define tasks mapping, a data placement and a scheduling based on the user constraints.

Given that all these parameters mutually influence each other a tool called PARSE (Parsimonious architecture solution space exploration) that explores the design space has also been defined. PARSE is constructed around recursive mechanisms allowing, based on simulations, to adapt the architecture, the tasks mapping, the data placement in order to get the best compromise for both the architecture and the application. To that end, a semi-automated approach is proposed.

Indeed since the design space is huge, the user helps PARSE by removing from the solution space the cores that does not match the needs of the application. Once done the user defines for each application task a set of potential core (processors or IPs) that can run it.

PARSE based on the choices made by the user, generates candidate architectures, along with a task mapping and a data placement. Each potential solution provided being evaluated by the means of a SystemC simulator in order to converge to the best compromise for the application and the architecture. To autonomously go through the solution space PARSE uses evolutionary algorithms [101, 102]. Once the best compromise identified, an executable binary code, along with the VHSIC [4] hardware description language (VHDL) code of the architecture is generated.

Because PARSE aims to explore and generates hardware architecture our second contribution is the definition of two hardware modules. The first one defines a hardware memory management unit (MMU) used to abstract the underlying memory hierarchy in context of cluster based architectures.

The definition of this MMU for cluster based architectures allows to ease the programming of MPSoC architectures and optimize the application execution time by limiting the data transfer among the platform. To that end based on a MPSoC architecture where the processors are grouped into cluster connected to a memory, we propose to manage the consistency of the memory at the hardware level thanks to the use of the hardware MMU.

The second hardware module is the accelerator interface (AI) which is a generic interface used to connect the processing cores (DSP [5] or hardware accelerator) to an interconnect.

The AI allows to easily connect different kinds of accelerators and processors together without taking care of their specificity's or characteristics. The AI then propose a plug and play approach, by defining a unified programming model and a generic architecture, that allows to disregard the IP specificity and the addressed interconnect.

4. very high speed integrated circuit
5. digital signal processor

Moreover still in order to take a full benefit of the computing power induced by MPSoC architectures, the AI behaves autonomously once initialized, which allows to realize concurrent processings in parallel.

1.2 Manuscript organization

The chapter 2 presents the state of the art of MPSoC architectures and of DSE tools. The different characteristics of the MPSoC architectures both at the architectural level and at the programming level are reviewed. This review is completed by a set of concrete examples, demonstrative of these differences.

After this study, the state of the art of available DSE tools is depicted. In this section the tools are split into three families following their abstraction level. Based on this distribution the advantages and the drawbacks of each family along with the capabilities and the needs of each presented DSE tool are given.

After the study of the state of the art, we see that the conception and the programming of MPSoC architectures are difficult, error prone and time consuming. Moreover none of existing DSE methodology is able to handle at the same time the definition of the application and of the architecture which leads to a separation of concerns. To face these limitations the chapter 3 presents a DSE methodology able to simultaneously define the application and the architecture. To enable this simultaneous definition and methodology PARSE tool is proposed. PARSE is based on recursive mechanisms and on a parsimonious representation of the solution space to define autonomously and simultaneously the application and the architecture.

Still with the aims to ease the access to MPSoC architectures the chapter 4 presents the FlexTiles project and the two hardware modules designed in the framework of this project. Indeed the MPSoC architectures present a lot of heterogeneity which greatly increases the difficult to conceive and program these architectures. To that end the first propose hardware module is used to abstract the heterogeneity of the processing core connected onto the interconnect. The second hardware module is used to propose an alternative memory management model to help to overcome the so called memory wall.

Finally, the chapter 5 concludes this manuscript by summarizing the contributions of our works and opportunities beyond this thesis.

2

State of the art

Abstract: The increasing amount of complex applications, has forced the designers to define architectures with high performance constraints. The embedded paradigm introduces new constraints and these architectures have to provide high performance throughput, without using a lot of hardware resources and within a very limited power budget. To reply to these needs the MPSoC architectures appear as a main solution. In order to clearly identify the difference between these architectures this chapter first gives a brief introduction. In a second time a MPSoC architecture classification along with the main example of current MPSoC architectures are described. Finally we present the design space exploration tool flow used to program and design these architectures.

Contents

2.1 Introduction

This chapter presents the state of the art of current MPSoC architectures and DSE tool flow. The first section presents the MPSoC architectures and focus more particularly on their: (1) number of cores and heterogeneity, (2) memory, interconnect, and architecture hierarchy, (3) management and programming.

The second part on the other hand focuses on how existing DSE tools flows help to program and conceive current MPSoC architectures.

2.2 MPSoC architectures

2.2.1 Introduction

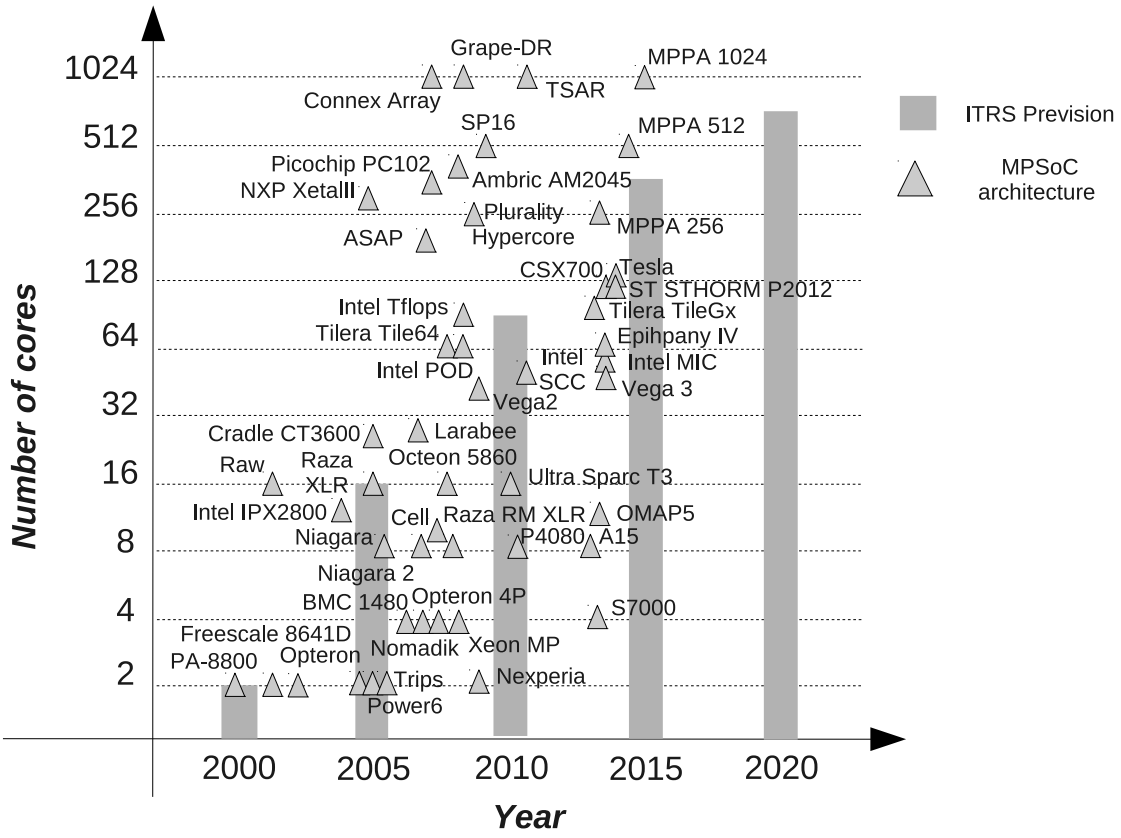


Figure 2.1 – Non-exhaustive list of the evolution of the multiple cores architectures in regards of their number of cores and their year of production [107].

The evolution of the computing power of the MPSoC architectures as stated by the ITRS is tremendous as shown on Figure 2.1. Among all the architectures the goal is to provide a maximum computing power. To that end several execution models (data flow, sequential) and physical structures

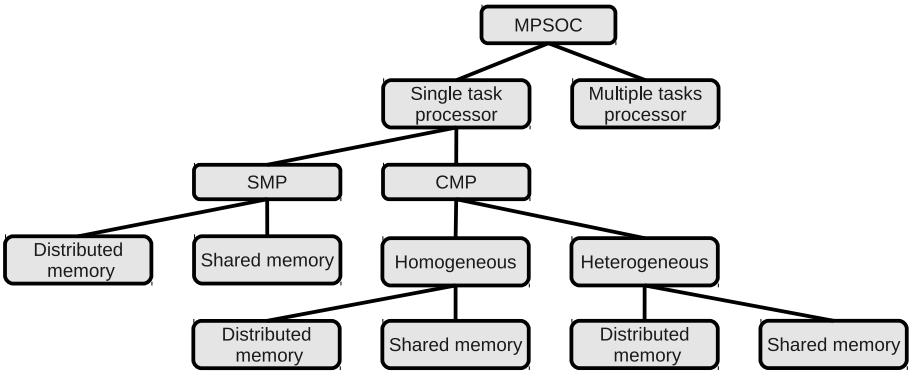


Figure 2.2 – Classification of the MPSoC architectures [74].

(SMP [1],CMP [2],CMT [3]) have been proposed.

These architectures though all based on multiple cores presents fundamental differences at the hardware and programming levels. In order to present a clarified view of the current state of the art we then propose to use the classification of the MPSoC architectures proposed in [74] (Figure 2.2).

This classification is divided into two categories the multiple tasks and the single task processors. The multiple tasks processors regroup the architecture composed of several processors, each one able to execute several tasks at the same time. The single tasks processors regroup the architecture composed of several processors able to execute only one task at a time but several tasks into the time. The single task processors then have an atomic vision of the execution flow while the multiple tasks processor does not.

The single task processors can be either based on a symmetric multiprocessing (SMP) or a chip multiprocessing (CMP) approach, which allow to shows two different execution models. The main difference between these two models come from the fact that within a SMP architectures the same cores are used for control and processing while within a CMP architectures some cores are dedicated to processing and some cores to control.

The CMP architecture can be either homogeneous or heterogeneous. Finally all these single task processors are either based on a distributed memory model or on a shared memory model.

The rest of this section depicts more in details the distinctions made between the several MPSoC architectures and gives concrete examples.

2.2.2 Single task processor

2.2.2.1 Symmetric multiprocessors

The association of several identical single task processors onto a shared interconnect is called a homogeneous architecture. These architectures allocate statically or dynamically the tasks onto the different

1. Symmetric multiprocessor
2. Chip Multiprocessor
3. Chip Multithreading

cores of the architecture.

On the Figure 2.3 for example, the core 0 execute the task 3 and 5, the core 2 execute the task 2 and 7, the core 2 the task 1 and 6 and the core 3 the task 4. This allocation is simple but introduces several problems. Indeed if for example the data produced by the task 3 onto the core 0 are needed by the task 6 onto the core 2 the programmer have to ensure the core synchronization, the data transfers and the memory consistency.

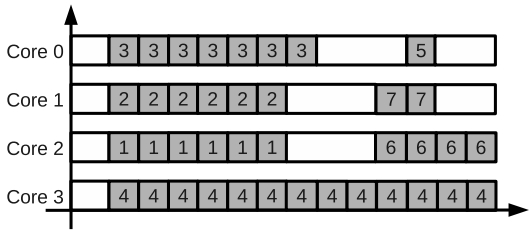


Figure 2.3 – Example of task mapping on a SMP MPSoC.

The main problem of these architectures then relies in their programming and more particularly in the synchronization between the cores and the data access.

Shared memory model:

In a shared memory model all the cores are communicating by the means of a common memory. The main advantage of this model is that all the cores are manipulating the same address space and so the tasks can easily synchronize or exchange data.

However each access to a shared memory resource has to be synchronized thanks to specific mechanisms in order to ensure the data protection, though this leads to a decrease in systems performance. Indeed, the read and write accesses have to be sequential if the memory can only read or write one data at a time. If the memory offers multiple accesses, the write accesses still has to be synchronized in order to avoid concurrent access to the same memory location and ensure the memory consistency.

The communication through a shared memory occur synchronization which can become important with the increase of cores. In order to avoid to limit the processor capabilities, the use of cache memory is becoming mainstream in shared memory architecture. The use of cache memory can partly hide the latency imposed by the access to the shared memory thanks to the prediction mechanism of this kind of memory.

In current MPSoC architecture based on the shared memory model the cooperative caching technique [66] is mainly used and the cache L2 is shared as depicted in Figure 2.4. With this method each processor can write into the shared L2 cache of the other processors. To that end cache to cache transfer, duplication, or replacement of inactive data techniques have to be developed.

Some example of architecture based on this solution are the Vega 3 processor [3] from Azul systems which is a MPSoC architecture composed of 54 cores design to run virtual machines in highly concurrent environments. The MPCore from ARM proposes architectures dedicated to the mobile market ranging from 4 to 8 cores [2]. The Octeon CN5860 targets intelligent networking, control plane, storage, and wireless applications with a MPSoC architecture composed of up to 16 cores.

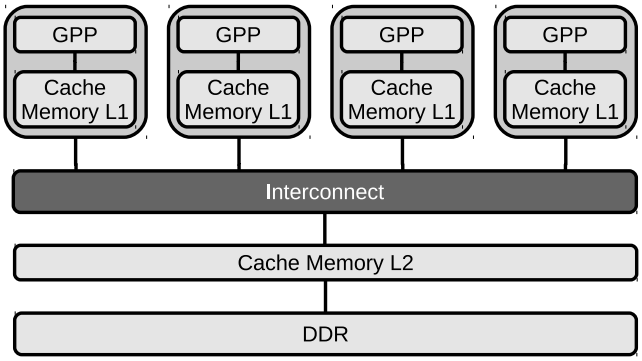


Figure 2.4 – Representation of a MPSoC architecture with a shared L2 cache

However the most demonstrative architecture based on the cooperative caching technique is the TSAR architecture [105] (Figure 2.5). The goal of this research project is to develop a MPSoC architecture composed of 4096 cores. To that end the architecture is based on a set of cluster connected through an interconnect.

Each cluster is composed of 4 cores. The cores are associated with their own L1 cache memory and connected on a shared L2 cache memory. These architectures based on the cooperative caching technique rely on the idea that most of the data access are done in the private cache memory L1. The communications with the external elements are then very detrimental and should be minimized in order to keep a high level of performance.

Moreover the communication with the L2 cache can also be detrimental. In order to limit the impact on the performance of the access to the cache L2 it is necessary to: (1) limit the concurrent access and (2) carefully size the memory in regards of the targeted applications or application domain.

These architectures in order to get a high level of performance are then strongly dependent of the task and data placement. The accesses to the shared memory have to be local in order to avoid any bottleneck onto the interconnect.

Sharing all the memory space can appear as the best solution to reduce the communication cost. However this solution is almost never used since the performance of a processor is dependent of the time needed by the processor to access to its first memory level [111].

In conclusion the shared memory model allows the tasks to communicate in a simple manner and allows to reduce the communication among the platform compared to a distributed model. However in the case of cache memory, the cache miss are very penalizing and can lead to bottlenecks and induce long latencies. Furthermore over a certain number of cores the scalability of the shared memory solution is limited due to important bottlenecks that occur onto the network. This is why most of current MPSoC architectures are based on the distributed memory model.

Distributed memory model:

In a distributed memory model each core is associated with its own memory. The main advantage of this model is to make the accesses to the memory exclusive. Indeed the memory is managed locally by a processor or a hardware controller which ensure that no concurrent accesses occur. In the case where

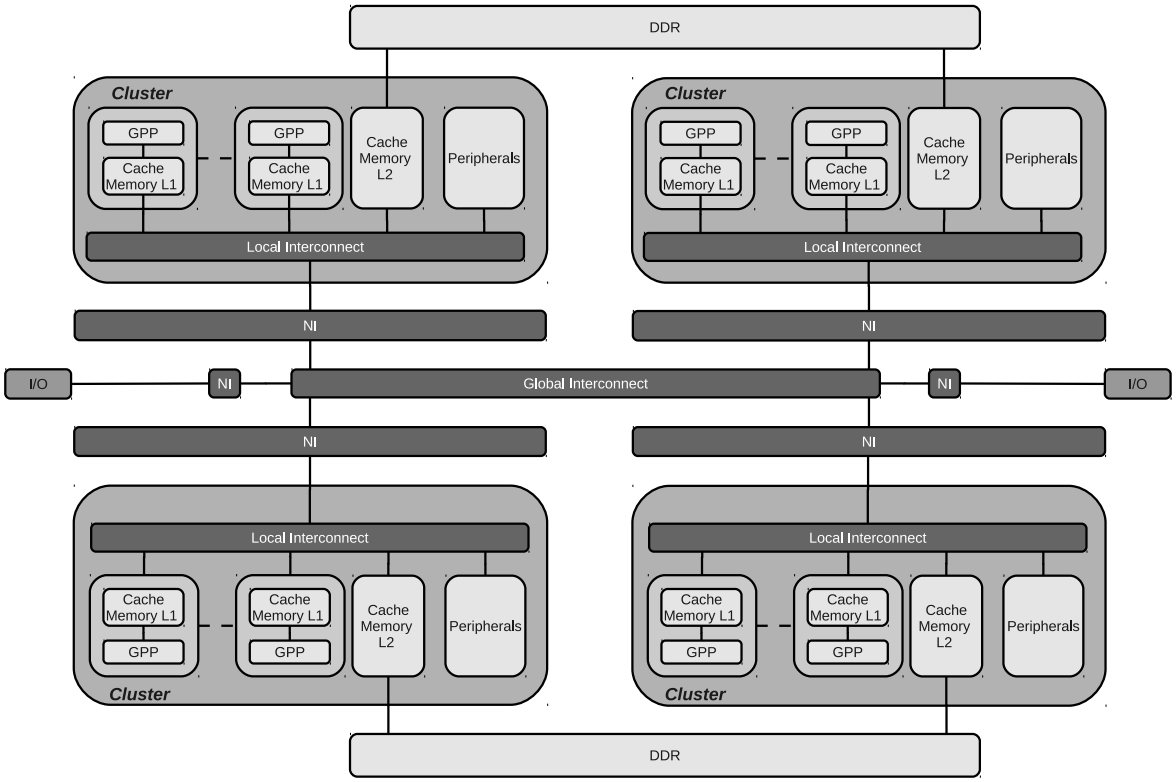


Figure 2.5 – The clustered TSAR architecture using shared memory

a task need the data produced by another task located on a remote core, the two tasks must establish an explicit communication by sending messages over the interconnect.

The architectures based on this model are for example the P4080 from Freescale [10] which connect 8 cores with their own cache memory onto a bus, the Intel Tflops [204] which propose a NoC based architecture connecting 80 cores associated with their own local memory and the Adapteva Epiphany IV [1] which is a low cost MPSoC architecture connecting 64 cores onto a NoC. Two major solutions based on this distributed memory model available commercially are the Tilera TileGx [28] and the single chip cloud computer (SCC) [13, 113] from Intel.

The TileGx is a MPSoC architecture composed of 100 cores connected onto a NoC and organized as a 10*10 grid. Each core is associated with its own cache memory and peripherals for synchronization and data exchange which create what is called a tile (Figure 2.6). In order to ensure the data consistency between all the caches, a distributed cache consistency mechanism is implemented in all the tiles.

The particularity of this chip is it allows the use of the memory in a message passing mode as a shared memory mode. This is made possible thanks to the tool chain provided by Tilera, which ensures from an application based on the shared memory model the generation of the message passing primitives. However the use of the TileGX platform with a shared memory model decreases the performances of the platform.

The SCC on the other hand proposes a NoC based architecture composed of 48 cores organized in a grid of 6*4. Within the SCC the cores are regrouped by pairs with their own cache memory and

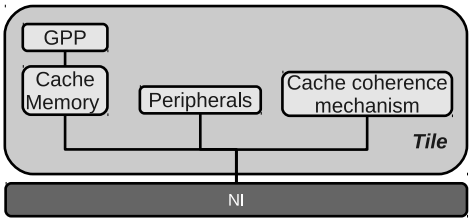


Figure 2.6 – Architecture of a TileGX tile organized around a GPP, a cache memory, a cache coherence mechanism, peripherals for synchronization and data transfers and a NI for network connection.

peripherals for synchronization with other tiles and connection onto the NoC (Figure 2.7). The SCC has also access to four on-die external memory controllers and to one system interface for I/O management.

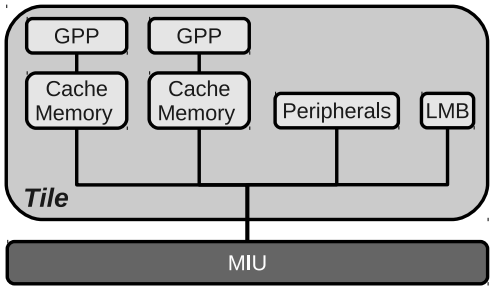


Figure 2.7 – Architecture of a SCC tile organized around two GPP along with their cache memory, peripherals for synchronization and data transfers, a LMB and a MIU.

The singularity of this chip is due to an on-chip message passing implementation, which is made possible thanks to specific instruction and hardware components (MIU [4], LMB [5]), which allow to reduces the latency induced by the message passing protocol [65] when managed at the software level.

Even with specific hardware modules the MPSoC architectures based on the distributed memory model increase the data transfer latency and synchronization among the platform to maintain the data consistency compared to a shared memory approach. Moreover with this model the user have to explicitly describe the communication between the tasks by the means of the message passing paradigm. The task of the programmer is greatly increased since he has to deal with data copying and consistency issues, by specifying explicit communications between the tasks into its code.

However the distributed memory model is mainly used in current MPSoC, because it is much more scalable than a traditional shared memory model and reduce the bottlenecks issue met with the shared memory model when the number of cores increases.

Nevertheless some approaches have proposed new programming model to get rid of the message passing paradigms drawbacks. With this programming model an application is represented as a set of tasks synchronized by the means of the data and organized under a data dependency graph. Example of these type of architectures are the Picochip PC102 [79] and Am2045 from Ambric [108].

4. mesh interface unit
5. local memory buffer

With these architectures, each core is always doing the same computation on different data and the data are transmitted from core to core until the end of the computation. The main advantage of these architectures is their simplicity of programming.

However these architectures are limited to data flow or computing intensive applications, which does not allow their use with time changing workload applications or with applications which requires specific memory movements between two tasks executions.

2.2.2.2 Asymmetric multiprocessors

In all the MPSoC architectures studied so far the processes dominated by conditionals computation are run with the same resources than the processes that are doing intensive computation. The use of the same core for different kinds of computation reduce the potential optimization of these SMP architectures.

The CMP architectures on the other hand proposes to define MPSoC architectures where specific and dedicated cores are used for the control and the synchronization, while the other cores of the architecture which are either homogeneous or heterogeneous are used for the computation.

As shown on Figure 2.8, this model shares some similarities with the super-scalar processor. However the selection, the synchronization and the allocation of the tasks onto the cores is realized by one or several remote control cores. These cores have a global view of the architecture or part of the architecture if several cores are responsible for the control.

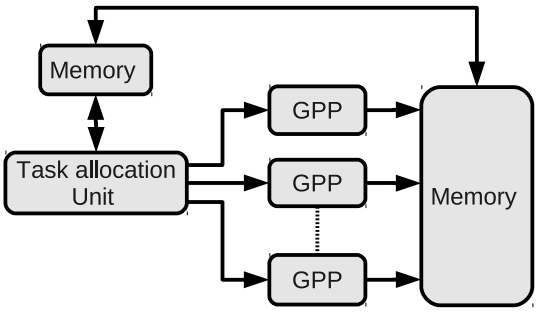


Figure 2.8 – Logical representation of a asymmetric architecture.

Homogeneous architecture:

The homogeneous CMP are constructed around identical processing cores. This is the case of the CELL processor [164] which proposes an architecture composed of a main core for control and synchronization, and 8 secondary cores for computing intensive tasks. With the same philosophy the CSX700 propose an architecture composed by 96 processing elements [5]. All these architectures are built around a distributed memory.

Examples of architectures constructed around the shared memory model also exist. This is the case of the plurality hypercore [20] with a processor composed of 256 processing elements (PE).

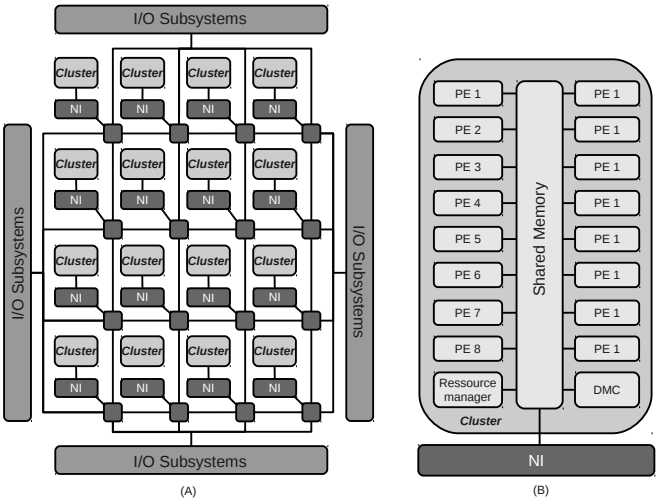


Figure 2.9 – (A) Overview of the MPPA 256 architecture. (B) Architecture of a MPPA cluster.

Two innovative architectures in their management of their memory space are the MPPA 256 from Kalray [75] and the SCMP from the CEA [205].

The MPPA 256 is an architecture composed by 288 very long instruction word (VLIW) PE (256 processing cores and 32 control cores) (Figure 2.9.A). This architecture is organized around 16 clusters connected through a synchronous NoC. The NoC is divided into two NoC, one for the data and one for the control in order to ensure different quality of service (QoS). Finally four I/O subsystems constructed around a processor to manage the connection with the outside environment are present onto the chip.

The particularity of this architecture being that the memory is distributed into the different clusters but shared by the PEs inside of the cluster (Figure 2.9.B). In order to take full advantage of this specificity the data locality have to be kept high in order to avoid remote data transfers between clusters. The MPPA architecture takes advantage of both memory models while maintaining the level of performance and the scalability.

The SCMP on the other hand [205] is a computing intensive resource seen by the central processing unit (CPU) as a co-processor. The SCMP is organized around 8 PE connected through a network to a set of distributed memory and I/Os (Figure 2.10).

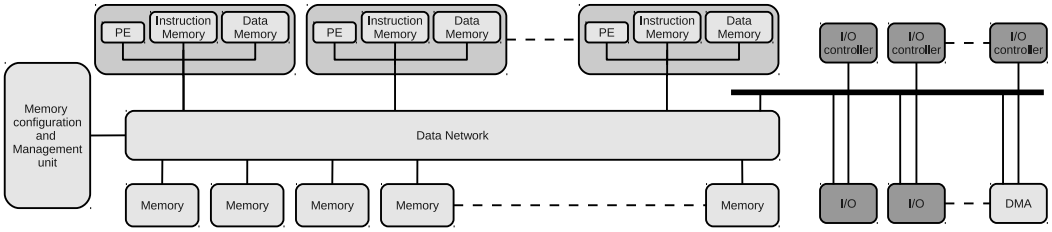


Figure 2.10 – The SCMP architecture.

The main innovation of this architecture is the management of the access to the memory. Indeed the address space is physically distributed and logically shared at the end of a task execution. With this approach when a task becomes eligible, this one is allocated to a PE and the access path to the memory which contains the data to be processed is modified.

This means that the data are not moved around the architecture but the access paths to the memory are changed at each task allocation, which reduces the impact of the data transfer onto the performance. However this solution is not scalable since the cross-bar needed to switch the path of the PE to the memories can be of significant size.

Heterogeneous architecture:

On the other hand the heterogeneous architectures propose solutions optimized for an intended domain of application. The energetic performances of these architectures are then improved. However the difficulties with these heterogeneous MPSoC architectures lie in:

- The abstraction of the heterogeneity of all the elements at the interconnect level.
- The programming issue.

A concrete example of a heterogeneous architecture is the OMAP 5 [27] for the telecommunication domain which propose to embed one ARM cortex A15 [2], two cortex M4, two graphical processing unit (GPU) and a DSP. The S7000 for video and image processing application is also another example and is constructed around an ARM 9 processor along with configurable hardware accelerators [24]. These two architectures are constructed around a distributed memory model.

Based on a shared memory the Nomadik [214] and the Nexperia [214] architectures propose to target the image and video processing applications. These architectures are organized around a standard processor (MIPS and ARM A9 respectively) along with a set of dedicated IP.

One of the main heterogeneous MPSoC architecture is the STHORM from ST Microelectronics [45] which is composed of 69 cores including 64 for the computing operation. This architecture is organized around four clusters connected onto a NoC, and a set of I/O bridges for a connection with the environment (Figure 2.11.A).

Each cluster is composed of 16 PE connected to a shared memory (Figure 2.11.B). The memory as for the MPPA is distributed at the MPSoC level but shared in the cluster. Moreover each cluster is composed by a set of 32 hardware accelerators used to execute specific functions inside the cluster. The function supported by these hardware accelerators are defined at design time. Finally a cluster controller is implemented within each cluster to control and synchronize the PE and the hardware accelerators. All these elements are connected thanks to two interconnects (The local interconnect and the cluster interconnect).

This approach as for the MPPA architecture allows to takes advantages of both memory models while maintaining the level of performance and the scalability. Moreover the use of dedicated hardware accelerators allows to increases the platform computing power and decreases the energy consumption. At the cost of an increase in programming and control.

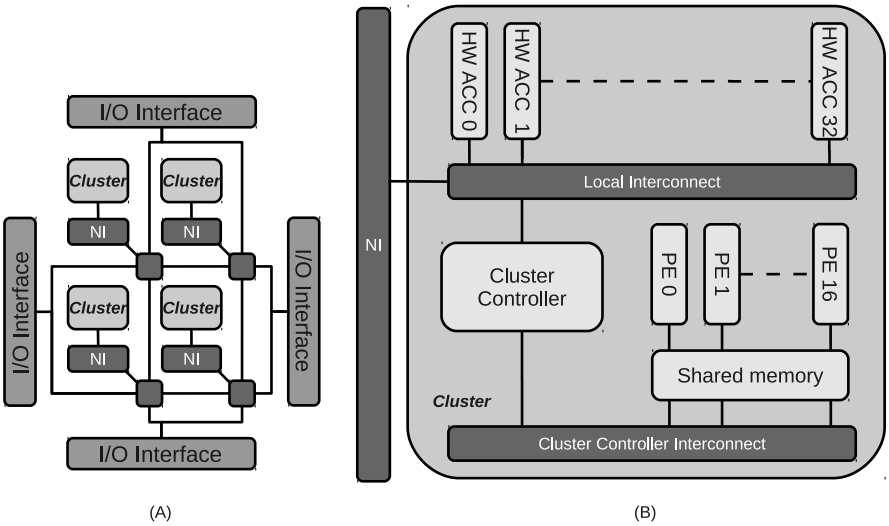


Figure 2.11 – (A) Representation of the STHORM architecture. (B) Architecture of a STHORM cluster.

2.2.3 Multiple tasks processor

A multiple tasks processors or a CMT architecture is a processor able to execute simultaneously several tasks onto the same core. Generally these architectures are similar to the VLIW or superscalar processors. However some pipelines stages are more complex in order to be able to execute at the same time several data and instruction flows. As shown Figure 2.12, several instructions are selected in parallel into different memory location thanks to several program counters.

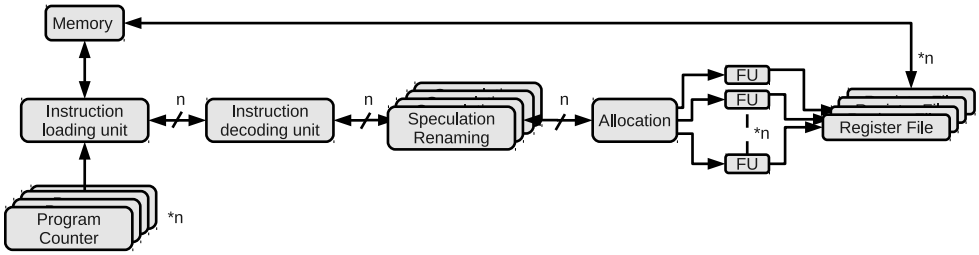


Figure 2.12 – Logical representation of a CMT architecture.

To ensure the access to different flows and supply the pipeline with enough instructions, several instructions files are used. Moreover the remaining and speculation functionalities are duplicated. Finally the registers are also duplicated in order to save the obtained results.

In the literature there exist a lot of multiple tasks processors [195]. They are either based on an explicit or implicit task parallelism. The implicit execution consists in the dynamic creation of tasks from sequential program thanks to the speculation mechanism. This approach then generates complex architectures, which are not suitable for the embedded systems domains, due to an important power consumption.

On the other hand the explicit task execution consists into the execution of multiple tasks onto shared

computing resources. The manner of running these tasks in parallel can be classified into three categories:

- Interleaved multithreading.
- Blocked multithreading.
- Simultaneous multithreading.

The interleaved multithreading processor changes the task to execute at each clock cycle (Figure 2.13.A). The advantages of this solution are that it is no more necessary to implement complex speculation mechanism since the control and data dependency are eliminated. Moreover the pipeline occupation rate can be high and the context changes have a low impact onto the execution time. However these multithreaded processors have to execute at least as many tasks as the number of pipeline stages.

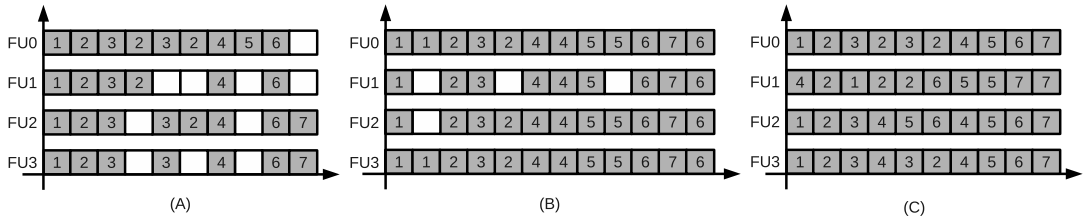


Figure 2.13 – (A) Example of an interleaved execution, (B) Example of a blocked execution, (C) Example of a simultaneous multithreaded execution.

The blocked multithreading on the other hand propose to execute each task until the task is blocked (Figure 2.13.B). For example this can happen during the access to a shared memory resource, or in case of data dependencies. In regards of the interleaved multiprocessor a reduced number of tasks is needed to fill the pipeline and a task can run without any interruption if no synchronization is needed.

In contrast with the two others execution models which are used with pipelined or VLIW processors, the simultaneous multithreading is used with superscalar processors. The main difference comes from the fact that several instructions which come from different tasks can be executed in the same clock cycle (Figure 2.13.C). The advantages of these simultaneous multithreaded architectures are that they fully exploit the task parallelism and offer a high level of flexibility and of performance. However the number of access port to the memory, to the registers and the number of pipeline stages have to be duplicated in regards of the number of instructions to be executed in parallel.

The processors currently present on the market are mainly based on a simultaneous multithreading approach. This is the case of the Power6 from IBM [93] which is constructed around two cores with their own L1 cache and shared L2 cache. Another example is the UltraSparc T3 from Sun Microsystems [202] which features up to 16 cores with their own L1 cache and shared L2 cache.

The main example of CMT architectures are the GPU which are based either on interleaved or simultaneous multithreading as the Tesla from Nvidia [142] (Figure 2.14). The Tesla architecture is based on a scalable processor array composed by 128 streaming processor (SP) organized as 16 streaming multiprocessor (SM) in 8 independent processing units called Texture/Processor cluster (TPC) (Figure 2.14).

The advantages of the CMT over the SMP and the CMP solutions are due to better resource occupancy. Indeed these models allow the simultaneous execution of several tasks onto several distributed processors. The CMT architectures then generate a high computing power and are mainly dedicated for

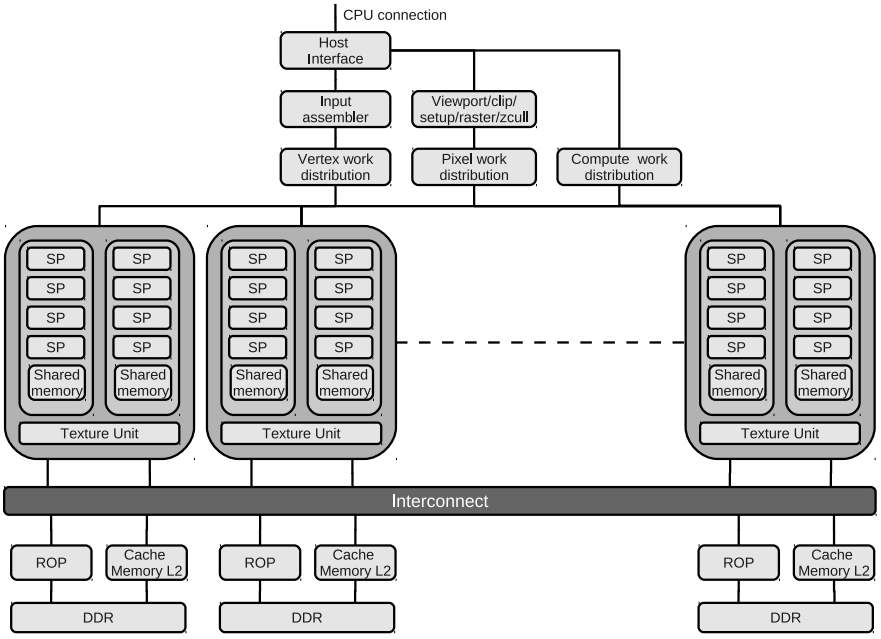


Figure 2.14 – Nvidia Tesla architecture.

high performance computing applications.

Indeed these solutions lead to very complex architectures with an important energy consumption which is unacceptable in the embedded systems domains which are highly constrained. Furthermore these architectures are dedicated to specific applications with regular processing patterns to be able to fully use the pipeline stages and the architecture resources.

2.2.4 Summary

The embedded applications complexities are becoming more and more important and the embedded systems have now to handle the simultaneous execution of several tasks at the same time. In order to answer to those needs the MPSoC architectures are becoming mainstream because they reach a high level of performances and are more power efficient.

The analysis of these MPSoC architectures have shown that three mains families currently exist: (1) the SMP, (2) the CMP, (3) the CMT.

The CMT architectures are complex and dedicated to high performance computing applications. These architectures are not well suited for some applications domains since their energy consumption is quite high. Moreover these architectures are dedicated to computing intensive applications to fully use the architecture capabilities.

The choice between a SMP or CMP architecture is much more difficult. Indeed the conception, the design and the programming of a CMP architecture is much more complex than a SMP architecture. However a CMP solution increase the energy efficiency of MPSoC architecture by using computing re-

sources adapted to the computation to be done or specific hardware resources for computing intensive processing in case of heterogeneous architecture. The choice between these two models is then highly dependent of the targeted application domains, the implemented processors, interconnect, memory hierarchies.

The choice between a homogeneous and a heterogeneous architecture is also difficult but some guidelines can be drawn. Indeed if the application needs are unknown it is then preferable to design a homogeneous architecture based on complex processor to target several domains of applications. It is preferable to know the application domain or the application to design a heterogeneous solution with dedicated processing unit able to speedup the intensive processing.

The study of each of these solutions to reply to the new applications needs did not show a single solution (Table 2.1). Indeed, the design of a low power MPSoC architecture supporting the required performance is not easy. This balance depends mainly on the application domain. This would influence many parameters such as the number of processing cores, the overall energy envelope, the type of interconnection network, the memory hierarchy, the number of access port to the local memory, the number of external memory, the deployment of the application on the system, etc. In addition, the manufacturing costs of this kind of platform are important (especially in modern technologies) and involve checking accurately each architectural and application choice.

Finally the programming issue is another key point to choose a MPSoC architecture. Indeed the mapping and data placement of the application onto the architecture have a big impact on the overall performance reach by the system. Thus to fully take benefit of the underlying architecture, the choice between a SMP, a CMP or a CMT architecture is then dependent of the application domain and of the user constraints.

Table 2.1 – Summary of the MPSoC architectures capabilities.In this table the MPSoC are evaluated following their capacities to ease the mapping and the data placement along with their ability to be scalable and target several applications with a reduce power budget.

			Mapping	Data placement	Scalability	Application domain	Power consumption
SMP	Hom	Shared	+++	+++	+	+	++
		Distributed	++	++	+++	++	++
CMP	Hom	Shared	++	++	+	+	++
		Distributed	+	+	+++	++	++
	Hetr	Shared	++	++	+	++	+++
		Distributed	+	+	+++	+++	+++
CMT			+++	+++	+++	+	+

2.3 Design space exploration tool flow

2.3.1 Introduction

Different kinds of DSE can be carried out during the whole system design process (from the initial specifications to the final design implementation). The classification of the DSE approaches proposed in [125] and depicted in Figure 2.15 shows the different possibilities made available to the designer during the development process of an architecture.

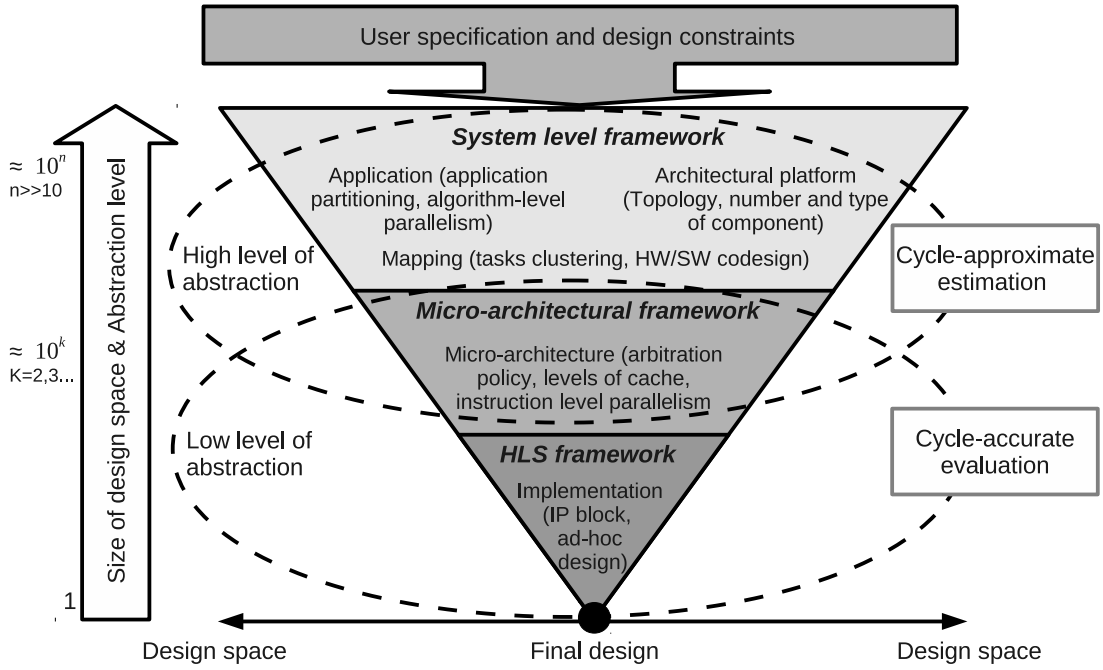


Figure 2.15 – Classification for DSE approach [125].

This classification is composed of three categories: (1) the system level framework, (2) the micro-architecture framework and (3) the high level synthesis (HLS) framework. The system level frameworks are based on a model approach and allow the exploration of the solution space in the earlier steps of development. This exploration may either concern:

- The application specification (level of parallelism, partitioning)
- The architecture definition (topology, memory hierarchy, number and type of components).
- The mapping of the application onto the architecture.

On the other hand the micro-architectural frameworks focus on the architecture definition and propose to model more accurately the behavior of the various components implemented within the platform. These frameworks are then preferably used when both the application and architectural constraints are identified.

Finally the HLS frameworks are the only frameworks that propose to generate an architecture (RTL [6]

6. register transfer level

code) based on an application specification.

2.3.2 System level framework

System-level frameworks allow to model and evaluate architectures and applications on different levels of abstraction using various models of descriptions. Most of the existing methods [98] use as a starting point of the exploration analysis, two abstract specifications, one describing the application and one defining the architecture available resources and constraints. The uses of abstract inputs specification make solvable complex design processes.

The main differences between these tools are the formalism used to represent the application. Indeed the tools are either based on a meta-model, kahn processor network (KPN) [81], synchronous data flow graph (SDFG) [162] or more recently on Array-ol [52] or Marte [7] formalism [103].

The meta-model representation is used in the Metropolis approach. Metropolis [39] is a framework targeting embedded systems development, which support simulation, formal analysis and synthesis. The Metropolis infrastructure relies on a meta-model to capture the application, the architecture and the mapping of the application onto the architecture. The metropolis meta-model capture the applications as a set of processes which communicates by the means of a media (a media is equivalent to a channel in SystemC) [15]. An architecture is represented by the means of performance models. Finally the mapping is done thanks to another model which maps the application model on the architecture model. This approach is attractive because several applications domains can be targeted. However the developments that are the responsibility of the designer are still too important [37].

The KPN on the other hand is a distributed model of computation (MoC) where a group of deterministic sequential processes are communicating through FIFO channels. The resulting process network exhibits deterministic behavior that does not depend on the various computation or communication delays.

Spade [140] proposes a methodology based on the KPN to explore signal processing architectures at system level. Applications are modeled as a network of concurrent communicating processes. Communication happens via channels that are bounded to processes ports. When executing, each process produces a trace of application events, which represent the application workload. Traces are then passed as an input of corresponding architectural models, which associate a defined latency to each trace event.

Based on the idea from Spade, Artemis [165] proposes a system-level co-simulation performed by using symbolic instruction traces generated and interpreted at run-time by abstract performance models. It adds facilities to explore and to refine architecture models.

Still based on KPN, Disydent [37] propose to go through the solution space of MPSoC platform design for shared memory multiple inputs multiple data (MIMD) architectures. In Disydent a design problem is a triplet (system, application, constraints) where the system is both an operating system and a hardware template that can be enhanced with dedicated co-processor. The main steps of the design flow are KPN modeling, functional validation, design space exploration, high-level synthesis and temporal validation.

7. modeling and analysis of real-time and embedded systems

The main advantage of using a KPN, is due to the determinism of this model [61]. Moreover the synchronization processes are done thanks to blocking read which is a very simple protocol than can be used either in software or in hardware. The control is also distributed to each process and the partitioning of a KPN in the context of a MPSoC architecture is then simple. Finally since the data exchange are made through data FIFO there is no notion of global memory and no contentions appears. However the KPN cannot be used to model all the applications which limit the use of these tool flows to specific domains [37].

The SDFG are a restriction of the KPN and allow compile-time scheduling. They are based on the ideas that each process reads and writes a known number of tokens each times it fires. System Studio [80] which is a SystemC simulator and specification environment for DSE of MPSoC architectures is based on SDFG for application description. Within System Studio applications are captured, verified and optimized using SDFG and finite state machines for control mechanism. The architectures are described based on models that can be tuned to represent the behavior of the targeted processors or hardware modules.

The Milan framework [152] still based on a SDFG approach, proposes a two phases DSE. This framework aims to create energy efficient design for signal processing application. To that end Milan uses a hierarchical methodology for embedded system design, estimation and design space exploration. The Milan methodology is based on the following step: (1) the application is described using synchronous data flow (SDF) graph(2) the architectural model is created, (3) the DSE is performed with the aims of improving the energy efficiency. The DSE is divided into two steps: (1) the solution space is explored automatically and the best set of solution is extracted, (2) the user with the help of the tool selects the best available solution.

Syndex [135] is a tool used for the implementation of signal processing algorithms on parallel calculators. It performs a matching between architecture and application based on the given input graphs. The applications are represented thanks to the SDFG model, while architecture graphs are simply described by a graphic representation of processor network showing how they are connected between each other. Within Syndex no help is provided to the user to define or refine its architecture or application models. The goal of Syndex is to produces based on the two inputs models an executable code of the application onto the platform. To that end Syndex is based on heuristics to find the best mapping and scheduling.

The SDFG however are not well suited to efficiently to describe data oriented loop-transformation [100].

To face the SDFG limitations the Array-OL formalism was proposed, since it is more suitable to describe data-oriented loop transformations [100]. The Array-OL is a high-level specification language dedicated to the definition of intensive signal processing applications.

In the Array-OL formalism, a program is a network of processes which communicate through shared arrays. A process is made of one or more parallel loops. At each iteration of these loops, a task is executed. The task may contain one or more loops, which are executed sequentially. The execution of a task is decomposed into three steps:

- Move portions of the input arrays to the local memory of the processor executing the task.
- Execute the task and generate portions of the output arrays.
- Move the results to the output arrays.

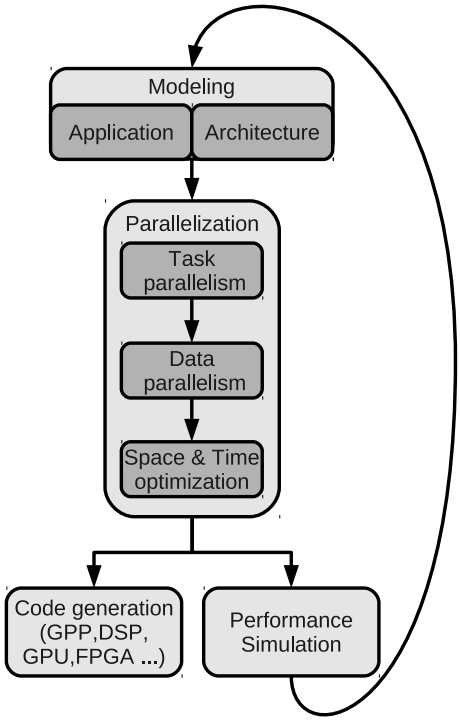


Figure 2.16 – Possibilities provided by SpearDE tool.

SpearDE [139] tool use the Array-OL formalism to represent the application and is designed to support an implementation flow (Figure 2.16) of an application on a MPSoC architecture written using specific models. The architecture model highlight, the structure of the targeted machine under an appropriate description. From these models and thanks to the tool, the mapping of the application on the architecture is realized by defining a placement of the application on the architecture resources. Once this placement obtained, SpearDE allows on one hand to generate a SystemC simulator to evaluate the performance of the various targeted architectures and on the other hand to trigger code generators in order to provide a parallel low level code executable on the targeted machine.

CoFluent Studio [6] proposes a visual embedded system modeling and simulation toolset where models (application and architecture) are captured in graphical diagrams using a domain specific language (DSL) or standard unified modeling language (UML) [180] notations which is a combination of system modeling language (SysML) [92] and of the Marte profile.

The Marte profile is an extension of the UML language to support model-driven development of real-time and embedded application. It consists mainly of four parts:

- A core framework defining the basic concepts required to support real-time and embedded domain.
- A first specialization of this core package to support pure modeling of applications.
- A second specialization of this core package to support quantitative analysis of UML models and performance analysis.
- A last part gathering all the MARTE concept.

Cofluent studio tool can be use to model and simulate the behavior, timing requirements, architecture and performance estimates (loads, power, memory, and cost) of any electronic system (HW IP, embedded SW application, mixed HW/SW multiprocessor system) (Figure 2.17). Use cases of the system are also modeled so the automatically generated transaction-level SystemC code can be used as verification test-bench. Behaviors are described with intuitive graphical notations and ANSI C/C++ code, although algorithms can be left undefined and abstracted to their sole execution time. Platforms are built by assembling generic models of universal components like processors, integrated circuits, memories, interconnects. Each generic model provides variable parameters to easily adjust its behavior and performance characteristics.

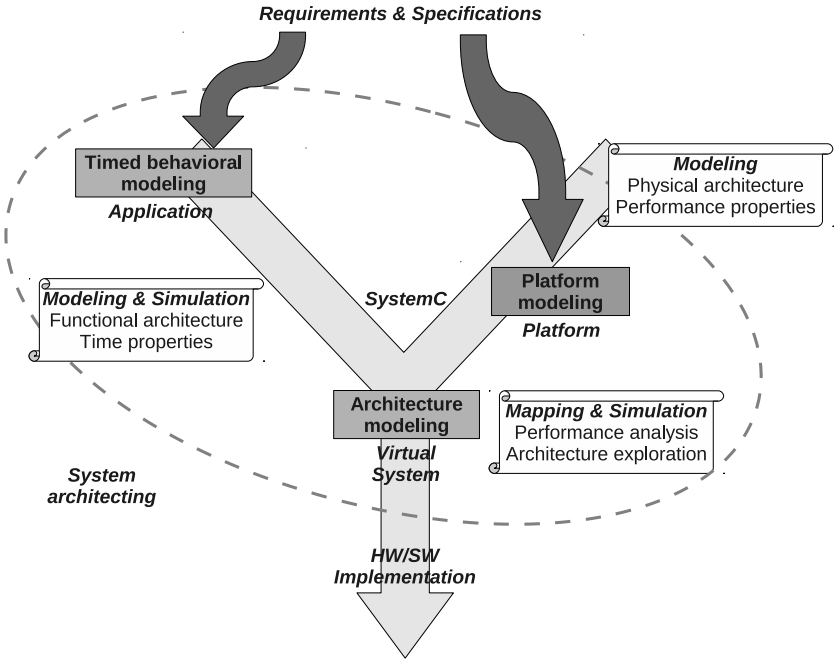


Figure 2.17 – Overview of Cofluent studio tool [6].

Space Codesign [157] on the other hand does not propose to help the user to parallelized its application and directly starts with a parallelize application C code (Figure 2.18). With this framework the first step is to define a set of potential architectures. These architectures vary the number of processors, the type of interconnect, the HW/SW partitioning of tasks, the mapping of software tasks to processor cores and the configuration of hardware components.

For each potential architecture and mapping, Space Codesign automatically generates a SystemC TLM-2.0 virtual platform of the system's hardware components, and a binary code for each processor in the platform. Then an architecture performance evaluation is done along with estimation of the used resources and of the power consumption. Finally, one architecture is selected based on the user constraints. The selected architecture can then be implemented manually in RTL, or refined automatically thanks to Space Codesign GenX [38, 156].

In conclusion the system level frameworks are tools that ease the exploration of the solution space. Moreover thanks to the models approach these tools are able to help the user exploring the solution space in a more efficient manner.

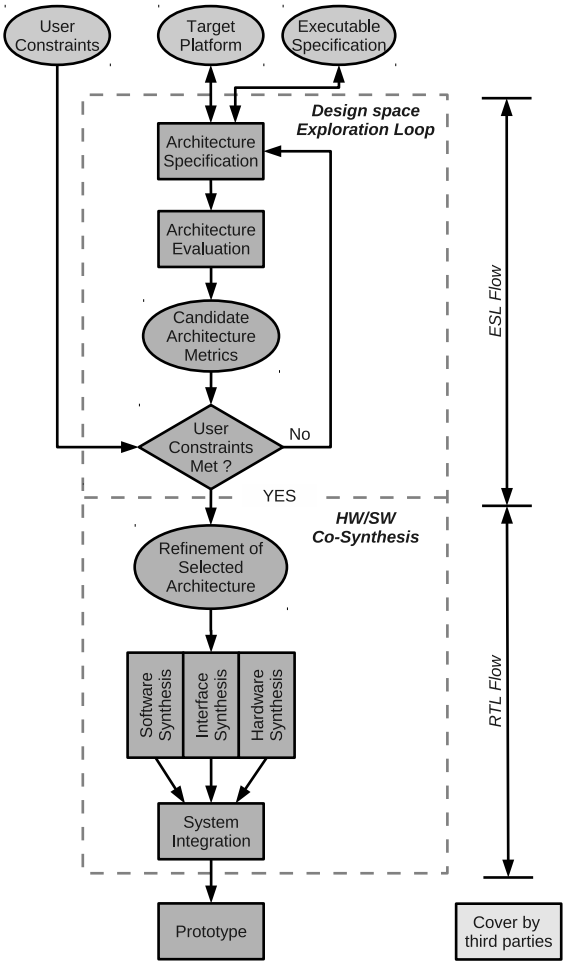


Figure 2.18 – Overview of Space codesign tool [23].

However the models approach do not allow to model in an accurate manner the behavior of the architecture which can leads to incorrectly sized solution. Furthermore most of exploration is done by hand which is time consuming and difficult, especially in context of MPSoC architectures where the solution space is huge.

2.3.3 Micro-architectural framework

Micro-architectural frameworks are used to model the design and behavior of a MPSoC platform and its components. These frameworks allow to model with accuracy the internal architecture of the platform and allow the performance evaluation and efficiency of these platforms.

For modeling purpose most of these frameworks are based on the SystemC language [120]. Indeed the SystemC language allows an accurate modeling of the architecture without being as restrictive as the hardware description language like VHDL. This then allow to explore more accurately the solution space, compared to the system level framework.

The SoCLib environment [22] is one of the major contributions in order to explore the micro-architecture of a MPSoC platform. The goal of this platform is to provide the designer with a tool to fully analyze and go through the architecture solution space. To that end performance analysis are performed as soon as possible in the development process. Moreover in order to follow the project status the SoCLib environment proposes several levels of simulation: cycle accurate, transaction level with time and untimed transaction level.

The SoCLib simulation engine is based on a set of models described in SystemC following the VCI/OCP communication protocol [32], which allows the integration of several IPs together. In order to be compliant with the VCI/OCP protocol all the IPs are represented by state machines. This approach allows the description of the IPs under two combinatorial functions, one for the transitions and one for the outputs. The simulation of the entire platform is equivalent to the creation of a network of state machines. The discrete event algorithm run by the SystemC kernel is then not overloaded and very efficient. However the state machine approach implies a linear evolution of the simulation time, and limits the scalability of the SoCLib framework. Moreover the application aspect is not taken into account which creates a separation of concerns.

Still based on a SystemC approach, GreenSoC [12] propose a simulator based on the integration of quick emulator (QEMU) [42] with a SystemC interface [78]. QEMU provides an open source emulation platform, which can be modified to suit the needs of several modeling platforms. QEMU is then used to emulate the software, while the hardware modules are described using SystemC (Figure 2.19).

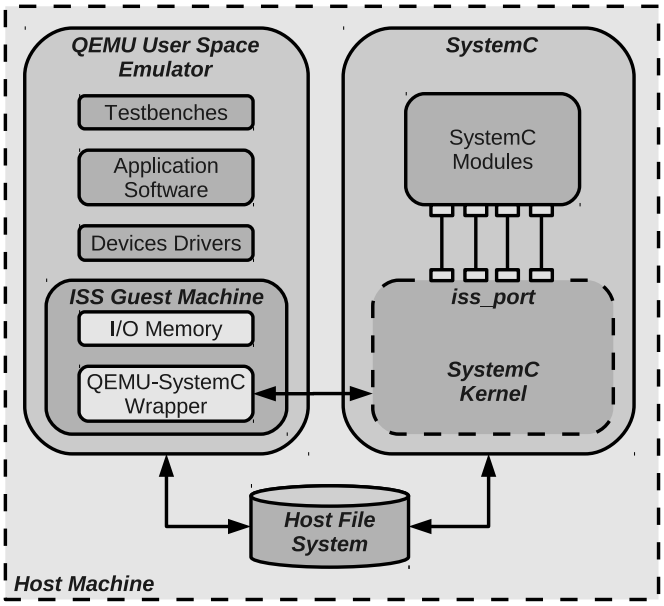


Figure 2.19 – Qemu-SystemC framework.

The code execution is based on a dynamic translation of the binary code which allows QEMU to speedup the execution time compared to the traditional instruction set simulator (ISS). The interactions between the two environments are based on synchronous transactions initiated by QEMU. The instructions executed into QEMU are simulated at the cycle level in SystemC. Qemu forwarding to the SystemC model all the information necessary to execute the instruction (kind of instruction, registers value, memory access).

QEMU-SystemC allows to efficiently get performance evaluation and validation of architectural principles. However the modeling of a complete MPSoC, in terms of simulation and performance estimates still has to be demonstrated [44].

Gem5 tool [47] results of combined efforts from industries (ARM, AMD, HP, MIPS) and academics (Princeton, MIT, Universities of Michigan) and propose a discrete event modular platform based on the merge of M5 [48] and GEMS [151] tools. M5 is used to provide a highly configurable simulation tool based on several instruction set architecture (ISA), while GEMS is used to provide a flexible and dedicated memory system, including several interconnection, memory and cache protocols.

This merge offers to the tool a high level of flexibility both at the modeling and the simulation levels. It is indeed possible to configure each model present into the library in regards of the current status of the project. The simulation can be run at the cycle accurate level or at the system level depending on the project progress status.

Moreover one of the main advantages of this framework is the architecture accurate temporal estimations which are made possible thanks to the discrete events simulation core. However the ability of GEM5 to integrate new architectural models is limited and reduces the use of this tool to the models present into the library.

One of the major micro-architectural exploration frameworks for MPSoC architectures is proposed by Imperas. Imperas offers to model MPSoC platform at the software level thanks to the OVP simulation platform [18]. This platform is structured around three main axes:

- A rich library of models composed by a large panel of processors, peripherals and MPSoC platforms.
- A set of modeling API.
- A simulation platform called OVPSim.

These features enable the OVP platform to answer three types of distinct needs (Figure 2.20): (1) the validation and estimation of performance of an application, (2) the description and validation of hardware devices, (3) the modeling and the simulation of MPSoC architectures. Moreover as done by QEMU the OVPSim simulation core is based on a dynamic binary code translation to reach a high level of performance. Moreover OVPSim is composed of a set of wrappers which enables to encapsulate the simulation core within other environment.

OVP thanks to its rich library of models, its API and documentation is one of the major tools in the domain. However one of the main drawbacks of this tool is the lack of temporal estimations which does not allow its use in the more advanced stages of the project development.

Finally, Mescal [150] proposes a framework where the designer has only to think about the data path of the design. All possible primitive operations that the data path supports are extracted automatically, i.e., the designer does not have to specify any op-codes or control logic elements. The designer can then restrict the set of operations and define more complex instructions from the set of primitive instructions. Cycle-accurate simulators and synthesizable verilog descriptions of the architecture can be generated. However this framework is limited to purely data-flow application.

In conclusion, the micro-architectural frameworks are tools that allow to model in a more accurate manner the behavior of a hardware peripherals, processors or MPSoC architectures. However this accuracy comes with a price. Indeed the exploration time is important since the modeling of each component

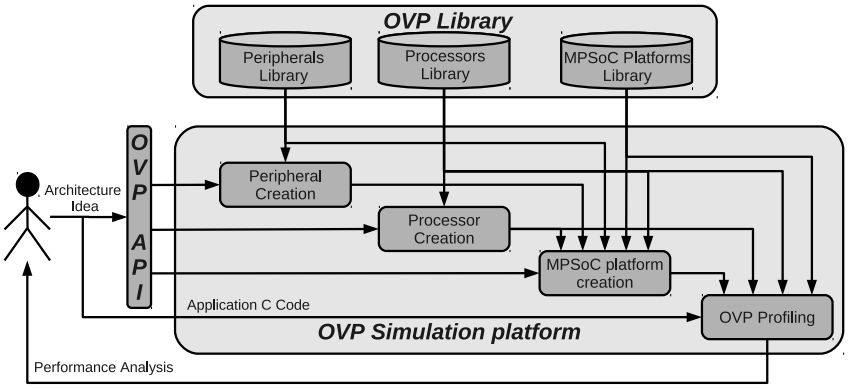


Figure 2.20 – Overview of the OVP platform.

requires an accurate definition. Moreover the modeling and the exploration is only done by hand which is time consuming.

Finally some of the micro-architectural frameworks do not take into account the application aspects. This is a main drawback since the application and the architecture mutually influence each other. This separation of concerns can then lead to an incorrectly sized architecture.

2.3.4 HLS framework

The HLS frameworks are able from a high level application specification (C, C++, OpenCL, SystemC, Matlab) to produces a hardware description of this application (RTL code). A typical HLS conception flow is depicted in Figure 2.21.

In such a flow, the first step is to produces a high level code, the second step is the compilation and the production of a hardware description. Finally the last step is a traditional hardware description language (HDL) code compilation. The compilation of the application high level representation is also done into three phases implemented within the front-end, the middle-end and the back-end. The front-end generates an intermediate representation on which it is simpler to apply transformations. The necessary transformations such as the scheduling, the optimization and the addition of variable are performed during the middle end. Finally the code generation is realized during the back end step.

The HLS tools are based on a set of internal operators [161] providing information about the area, the latency and other used hardware resources. So with a relative accuracy it is possible to quickly get hardware resources estimations. With these estimations the user may impose transformations, like loop transformations to guide the tool to the most promising regions of the solution space.

There exist many HLS tools both in the academics and in the industrial domains. Catapult C [104] and Cadence-C-to-silicon [4] are two of the major tools of the domains. They use as an entry point the C, C++ and SystemC language and produce in output a HDL code for several FPGA (Field Programmable Gate Array) families.

HDL coder [182] which uses as entry point the Matlab language is another example. HDL Coder generates portable, synthesizable Verilog and VHDL code from MATLAB functions, Simulink models,

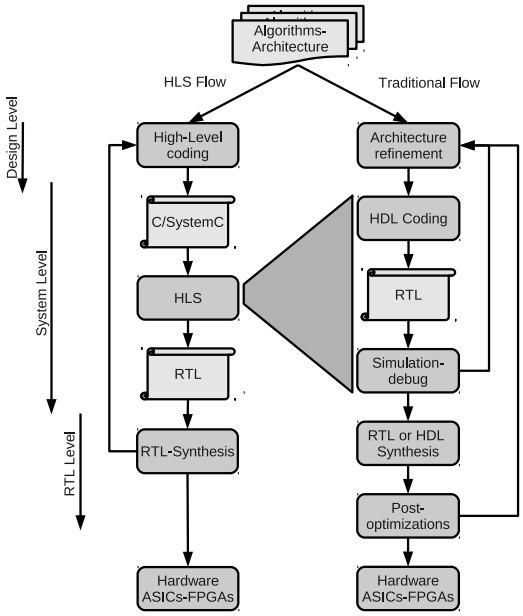


Figure 2.21 – HLS and traditional design process [184]

and Stateflow charts. The generated HDL code can be used for FPGA programming or ASIC prototyping and design. HDL Coder provides a workflow advisor that automates the programming of Xilinx and Altera FPGAs. The HDL architecture and implementation can be controlled, by highlighting critical paths, and generating hardware resources utilization estimates.

Synflow [25] proposes to use the RVC-CAL [213] language for the application description in order to produces VHDL or verilogs code. RVC-CAL is a language specially dedicated to the description of data-flow application.

The FPGA vendors have also proposed their own HLS tools. Xilinx with Vivado HLS [89] propose from a C-based design description, directives and constraints to generate design files in Verilog, VHDL and SystemC. In addition verification and implementation scripts, used to automate the RTL verification and RTL synthesis steps are also created. This HDL representation can then be synthesized into a form that can be implemented in a Xilinx FPGA.

Altera with Altera OpenCL proposes on the other hand to start with an OpenCL application description. Altera OpenCL [183] from this application description generates dedicated VHDL code for the hardware architecture. The goal of the OpenCL approach is to ease the extraction of the application parallelism. Designing in this way allows the designer to easily migrate to new FPGAs that offer better performance and higher capacities because the OpenCL compiler will transform the same high-level description into pipelines that take advantage of the new FPGAs.

Academics have also proposed to generate HDL from a high level description. Some of them take as inputs data flow applications described in C in order to be able to generate a HDL description of the applications [7, 168]. Other approaches propose to target several application domains [11, 14, 160], but their approach is not yet mature which lead to inefficient architectures, or put a big effort on re-writing the C code to a solution that can be easily handled by the tool.

In conclusion the HLS tools provide an efficient way from a high level application description to generate a hardware architecture. However despite the benefit on the design time all these approaches are not able to reach the level of performance of a hand coded IP block. Moreover since these tools generate for each task a hardware IP block, these approaches can lead to big architectures which are not always suitable for the embedded system domain. In addition the abstraction level does not allow the exploration of a large set of solutions (Figure 2.15).

2.3.5 Summary

The question of an efficient design space exploration methodology has been extensively studied in the past years. This result in a set of tools used to ease the design space exploration at different level of abstraction as shown on This result in a set of tools used to ease the design space exploration at different level of abstraction as shown on Figure 2.22.

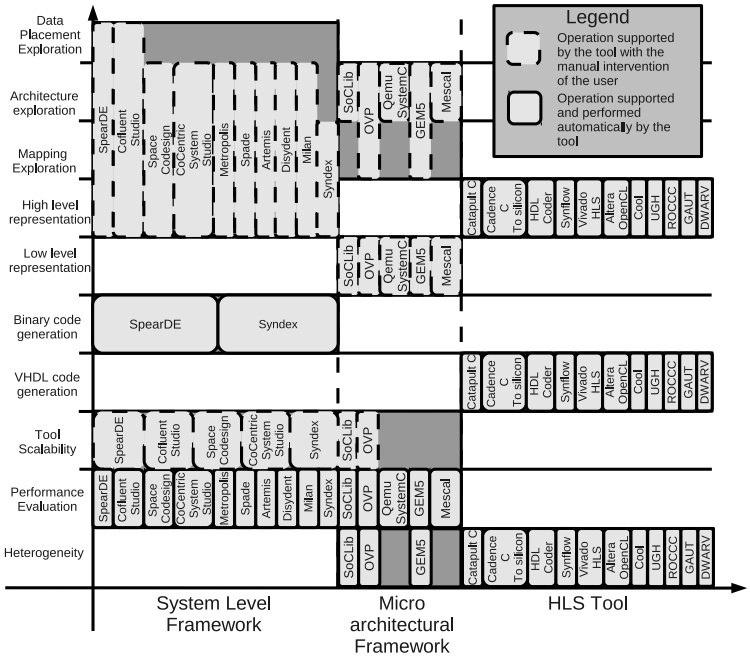


Figure 2.22 – Representation of the needs fullfiled by each tool presented during the state of the art

On this figure the tools are compared regarding several characteristics parameters depicted into section 1.1.

On one hand, the system level framework allows an easy modeling of both the application and the architecture, in order to reduce the time to market. However the models approaches do not allow to take into account all the application and architectural constraints.

This is why tools that explore the micro-architecture have been defined. These tools allow to clearly details all the architectural constraints, but in most of the case the application constraints are not taken into account or the tool does not allow to fully explore the overall architecture. Moreover for the system level and the micro-architectural frameworks the exploration is done by hand which is error prone and

time consuming.

To reduce the time to market HLS tools that produce a VHDL code from an application description were proposed. But the level of performance provided by these tools does not reach the level of performance of a hand-coded IP block.

2.4 Summary

In this chapter we saw that the MPSoC architectures appear as a promising solution to face the new application needs. These architectures have high computing capabilities all in a restricted power budget.

However these architectures are difficult to design, to program and the management of the heterogeneity is another issue. To ease the conception and the programming of these platforms a set of DSE tools have been proposed.

Despite the services bring by these initiatives none of these approaches are able to meet the needs of a complete design space exploration processes as depicted on Figure 2.23. One of the main drawbacks of existing tool flows is that no tool is able to take into account at the same time the application and architecture characteristics while these parameters mutually influence each other. This last statement becomes particularly true for current and future MPSoC architectures.

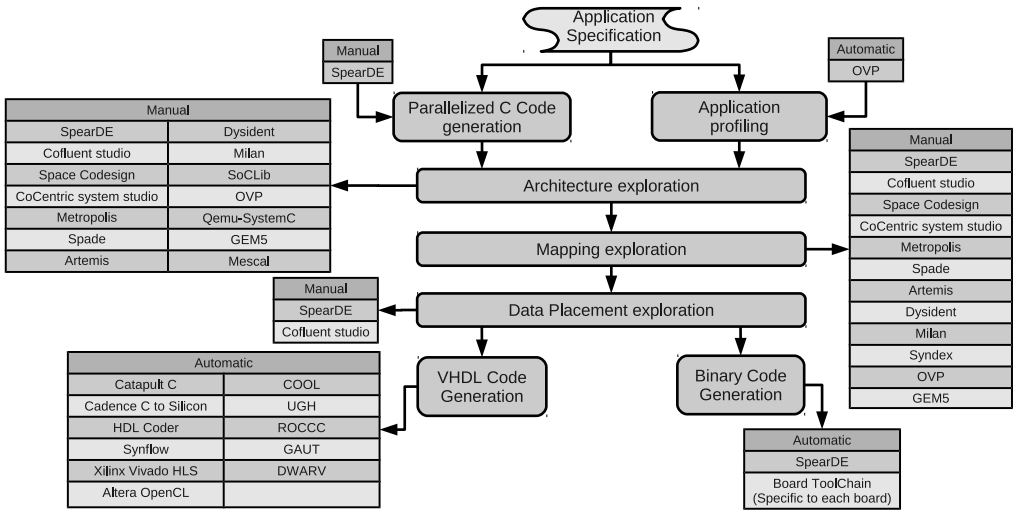


Figure 2.23 – Representation of an ideal flow where each state of the art tools is positioned

This why in the context of this thesis we propose a DSE methodology to ease the design and programming of MPSoC architectures. Moreover we also propose to define at the hardware level abstraction interfaces to ease the MPSoC architectures programming.

3

Parsimonious architecture solution space exploration

Abstract: The optimized configurations of MPSoCs architectures requires complex researches and spotlight the need of an automatic design processes to reduce the time to market and allow a fast and efficient design space exploration. To that end this chapter introduces a design space exploration methodology which explores automatically and at the same time the application and the architecture. To allow the automatic exploration a tool called PARSE is defined. PARSE is based on recursive mechanisms and on a parsimonious representation of the solution space to allow an efficient exploration in a reasonable amount of time. This methodology has been tested on both benchmarks and real life applications and has shown promising results.

Contents

3.1 Introduction

We show in the state of the art (see section 2) that none of the existing tools are able to explore at the same time the application and the architecture of a MPSoC.

Currently either the application or the architectural model is set and the exploration of the solution space aims to improve the definition of the input models. Furthermore the most exhaustive tasks of current DSE approach are under the responsibility of the user which has been proven to be time consuming and error prone. One possible solution to ease the user task is to use HLS tools, but this can lead to oversized architecture and sub-optimal solution.

In order to ease the development of MPSoC architectures, this chapter presents a design space exploration methodology which proposes to explore and define at the same time the architecture, the application, the mapping and the data placement of the application onto the architecture.

To that end we have defined a tool called PARSE for parsimonious architecture solution space exploration. PARSE is based on a set of evolutionary algorithms and on a parsimonious representation of the solution space to efficiently explore this latter. Moreover a set of metrics have been added to the algorithms to reduce the exploration runtime by avoiding the accurate exploration of uninteresting design points.

The rest of this chapter gives in section 3.2 the definitions used in this chapter. The proposed DSE methodology is depicted in section 3.3. The mapping and data placement heuristics are explained in section 3.4 and 3.5 along with the results obtained in a standalone mode. Finally the results obtained when the mapping and data placement heuristics are used jointly are depicted in section 3.6.

3.2 Definitions

3.2.1 Application model

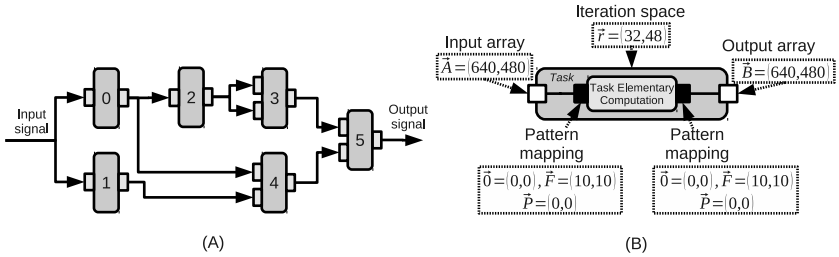


Figure 3.1 – (A) Application represented as an oriented graph, (B) Task specification represented with the Array-OL formalism.

The application model needed for the solution space exploration (Figure 3.1.A) is an oriented graph $G(T, E)$ composed by a number of tasks (nbt).

With this representation the tasks are elementary computation (EC) (Figure 3.1.B) iterated into nested

loop allowing to consume the multidimensional input array represented as the edges in the graph (Figure 3.1.A).

Data handling is based on the Array-OL formalism [100]. The Array-OL formalism specifies the size of the consumed and produced array ($\overrightarrow{A}$ and $\overrightarrow{B}$). The information about the input/output array consumption/production is given by the origin vector $\overrightarrow{O}$, the fitting matrix $\overrightarrow{F}$ and the paving matrix $\overrightarrow{P}$. The origin vector specifies the coordinate of the first data to process. The fitting matrix says how to parse the input array to fire the EC, the paving matrix explains how the patterns are positioned into the array. The iteration space of the loop ($\overrightarrow{r}$), specify how many times the EC has to be repeated on each array dimensions in order to consume all the input data.

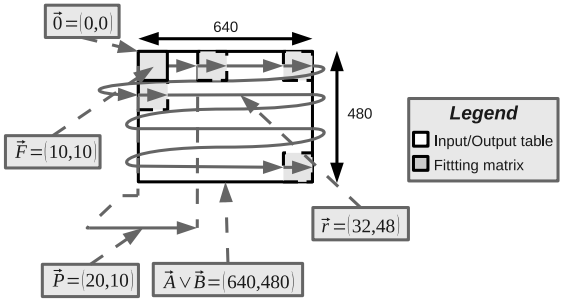


Figure 3.2 – Example based on the task representation given Figure 3.1.B of how the input/output array are read/write.

Based on the task representation given Figure 3.1.B, the Figure 3.2 represent how the fitting matrix $\overrightarrow{F}$ parse the input/output array $\overrightarrow{A}$ and $\overrightarrow{B}$ based on the iteration space $\overrightarrow{r}$.

The data exchange between the processing cores being done based on array exchange. These arrays are located either in the external or into the local memory of the architecture.

It is important to note that this application model is not provided by the user as entry point. Indeed the application model is automatically obtained by the proposed DSE methodology and used as an internal representation to enable the exploration of the solution space.

Moreover, each application is associated with an execution time constraints set by the user (et_cst). This constraint is set to guide the exploration of the solution space.

3.2.2 Task model

In the context of this thesis a task has several timing related constraints. These enable us to consider heterogeneous architectures. These constraints are then dependent of the current mapping and of the architecture and need to be evaluated at runtime. The considered timing are represented on Figure 3.3, for one task (t_0): (1) the fire time of the task ($t_i ft$), (2) the time needed by the task to process the data ($t_i et$ ie its execution time on a specific core of the architecture), (3) the time needed by the task to access the data ($t_i ct$), (4) the overlap of the communication time over the computation time ($t_i over$).

Based on the above information the total task runtime ($t_i run$) can be computed according to Eq 3.1.

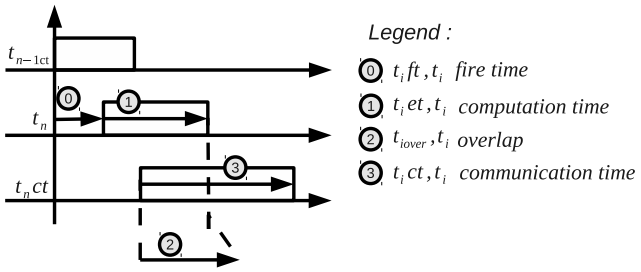


Figure 3.3 – Example of task execution time evaluation.

$$t_i run = t_i ft + t_i et + t_i ct - t_i ct * t_i over \tag{3.1}$$

As can be seen, optimizing the overlap of tasks (i.e. performing communications during computations of others tasks, enables to optimize the overall application execution.

3.2.3 Architecture model

In order to be accurate enough and propose architectures close as possible from the optimal design point, an architecture is divided in two levels: (1) the platform level and (2) the node level. At the platform level (Figure 3.4) the architecture is seen like a set of interconnect and nodes. This allow PARSE to measure and evaluate the benefit of an interconnect, to see if it is beneficial to add or remove nodes. Existing architecture can be modeled also with this representation.

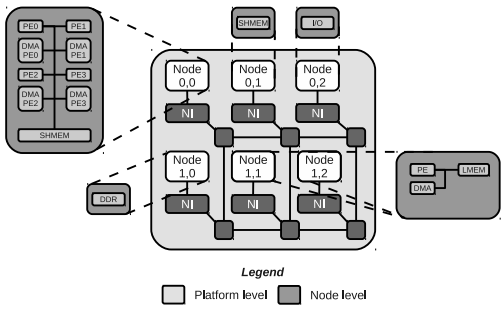


Figure 3.4 – Architecture model generated by PARSE

Two types of physical nodes are considered: (1) the active nodes and (2) the passive ones. The active nodes contain one or several cores and are used for processing purpose. The passive node exposes their address space to the active nodes. These nodes are not able to initiate anything on the interconnect and they can be read or written by the other nodes. These nodes are typically memories plugged onto the interconnect or I/O peripherals.

An architecture is described thanks to a hierarchical topology graph. A hierarchical topology graph is a graph $P(M, F)$ made of two levels: (1) the platform level and (2) the node level. At the platform level the architecture is constructed around a number of nodes (nbn). These nodes are divided into a

set of processing nodes (nbc) and a set of passive nodes (nbp) (Eq.3.2). Each $node_i \in M$ representing a node in the topology and the directed edge $f_{i,j} \in F$ representing a communication link between the $node_i$ and $node_j$. The weight of the edge $f_{i,j}$ (not shown on Fig 3.4 for sake of clarity), denoted as the $bw_{i,j}$, represents the bandwidth available on the link $f_{i,j}$.

$$nbn = nbc + nbp \tag{3.2}$$

At the node level the node are either active or passive and can be organized around processors, memories, dedicated processing IPs. For each architecture its related cost ($arch_{cst}$) is computed. The architectural cost compares the amount of used resources with the amount of available resources ($arch_{resources}$) which allow to valid or invalid the current architectural solution ($arch_{sol}$) Eq.3.3.

$$arch_{cst} \leq arch_{resources} \qquad \text{then } arch_{sol} = valid \text{ else } arch_{sol} = invalid; \tag{3.3}$$

3.2.4 Nomenclature

To ease the reading of this chapter, the table 3.1 resumes the variable names used in this work.

Table 3.1 – Variable names nomenclature

Variable Name	Representation
t_i	Task t_i
$t_i ft$	Fire time of the task t_i
$t_i et$	Time needed by task t_i to process the data
$t_i over$	Task t_i overlap
$t_i ct$	Task t_i communication time
$t_i run$	Task t_i runtime
$\overrightarrow{O}$	Origin vector
$\overrightarrow{\overrightarrow{F}}$	Fitting matrix
$\overrightarrow{r}$	Iteration space
$\overrightarrow{\overrightarrow{P}}$	Paving matrix
$node_i$	Node i
nbt	Number of tasks
nbn	Number of nodes
nbc	Number of processing nodes
nbp	Number of passive nodes
$f_{i,j}$	Communication link between $node_i$ and $node_j$
$bw_{i,j}$	Bandwidth of the link $f_{i,j}$
et_cst	Execution time constraints set by the user
$arch\_sol$	Current architectural solution
$arch\_cst$	Architectural cost
$arch\_resources$	Available architectural resources

3.3 DSE methodology

3.3.1 Presentation

The proposed DSE methodology (Figure 3.5) is based on three inputs: (1) the application sequential C code, (2) the architecture library, (3) the constraints file. In order to explore the solution space and generate both a hardware architecture and an executable binary code we defines six services: (1) a code profiler, (2) a code parallelization tool, (3) a design space exploration tool (PARSE), (4) a SystemC simulator, (5) a binary code generator and (6) a HDL code generator.

The first stage of the proposed methodology extracts from the sequential C code the application parallelism and its task graph. Based on the tasks graph the application is profiled. The goal is to profile each task onto each core of the library. This operation is done to identify the hot points of the application. Based on these informations (profiling and task graph) the user can:

- Choose to adapt the application sequential C code if the demands in terms of resources are too important (Memory load/bandwidth, computing power ...)
- Define new IP into the library, if specific tasks of the application have to be speeded-up.
- Leave the application and architecture library as is and define for each task a set of cores that can run it. This operation is done in order to remove from the solution space the unsuitable cores.

The reduction of the solution space is a key step in the proposed DSE methodology. Indeed the simultaneous definition of the architecture, the mapping and the data placement generate the exploration of a huge solution space. The reduction of the number of solution based on the user experience allows to define the parsimonious representation taken as input by PARSE to focus on the most promising regions of the solution space and to reduce the exploration runtime.

Based on the IPs choices made by the user PARSE is triggered. The exploration of the solution space is done in a recursive manner where three evolutionary algorithms evolve at the same time their population of solutions. The first algorithm explores the architectural solution space, the second algorithm explores the mapping solution space and the third one go through the data placement solution space.

In order to evaluate each potential solution (Architecture, task mapping and data placement), a SystemC simulator is generated. The performance evaluations provided by the SystemC simulator are used by the three evolutionary algorithms to explore and evolve the potential solutions and converge to the most promising region of the solution space.

Once PARSE concludes its exploration, the best compromise is chosen as the final solution. The final operations of the DSE methodology consists on one hand to generate the HDL code of the identified architectural solution and the executable binary code able to run on top of the generated architecture.

In the context of this thesis the functionality ensured by each function of the DSE methodology is detailed along with the expected inputs and outputs. However due to the inherent complexity of the proposed approach we choose to focus on PARSE specification.

The code parallelization, profiling HDL and binary code generation steps have already been described by the state of the art.

This chapter then describes the architecture, mapping and data placement heuristics behaviors and

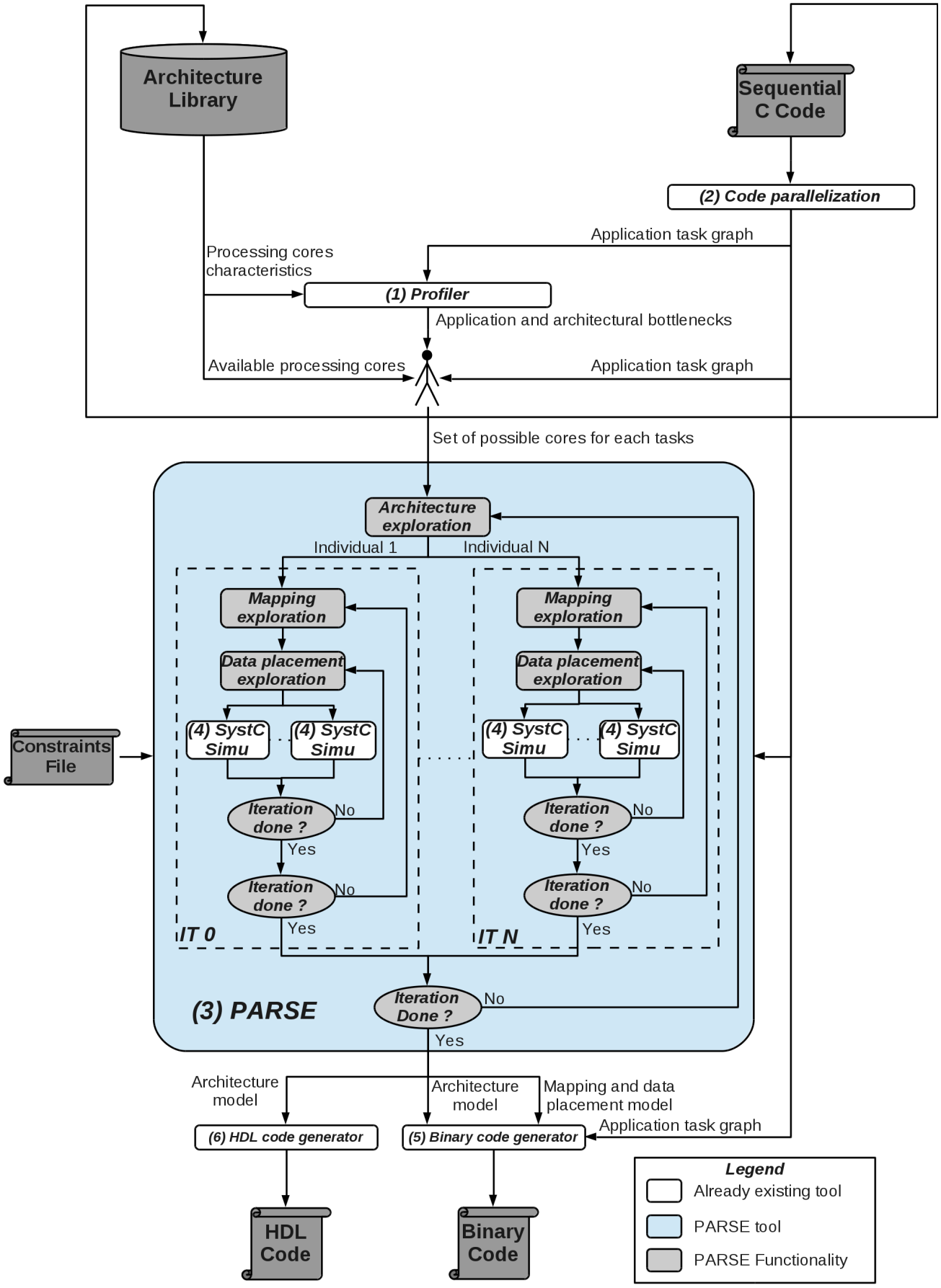


Figure 3.5 – Design space exploration methodology.

operations. However due to the inherent complexity of PARSE only the mapping and data placement heuristics are fully described, implemented and tested. We concentrate on the mapping and data placement heuristics since they are needed to allow the architecture exploration. Moreover it already exist an important set of MPSoC architectures to efficiently train the mapping and data placement heuristics.

A functional implementation proposal of the DSE methodology with already existing tools is however given in annex A.1.

3.3.2 Methodology description

3.3.2.1 DSE methodology inputs

Architecture library:

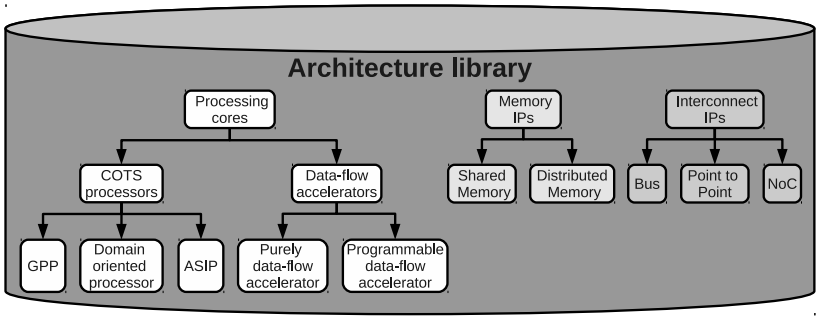


Figure 3.6 – Architecture library representation.

The architecture library is defined by the user and has to be build once. The architecture library used by the DSE methodology is composed of all the IPs necessary to define and build a MPSoC platform. All the IPs are captured thanks to the IP-XACT [35] standard formalism to ease the architecture and HDL code generation. The library (Figure 3.6) is composed of three categories of IPs: (1) The processing cores used to execute the application tasks, (2) The interconnect IPs used to transfer the data between the cores and between the memory, (3) The memory IPs used to store the data shared among the platform. The processing cores are distributed into two categories as proposed in [203]:

- The commercial on the shelve (COTS) processors:
  - The GPP are designed to support general purpose computation and target several application domains. The GPP execute a standard ISA.
  - The domain oriented processors (DSP, GPU...) are designed for a specific domain of application. They execute a domain specific ISA.
  - The application specific instruction set processors (ASIP) are processors designed for the benefit of specific applications or application domain. They execute a custom domain specific ISA.
- The data flow accelerators:
  - The purely data flow accelerators provide a high level of performance on a dedicated task. To that end the communication is based on a data streaming approach, and the accelerators just read and write into FIFOs. No ISA is involved since these accelerators are dedicated to a

specific function (FFT, Convolution, Filtering...).

— The programmable data flow accelerators have the ability to exhibit different behaviors based on the value written into the accelerators registers. Typically these accelerators are filters where the filter coefficient values can be modified at runtime. The programmable data flow accelerators provide the same level of performance as the purely data-flow accelerators, but their hardware cost is higher.

The memory IPs are composed of IPs that allows to implement and test different memory hierarchy within the architecture (distributed and shared). The interconnect are composed of bus, NoC and point to point connection into order to be able to identify the most adapted interconnect.

In the context of this thesis the FPGA dynamic reconfiguration capabilities are not used. The IPs mapped onto the FPGA are considered mapped during all the application execution.

Application inputs:

- **Sequential C code**: The code given as input of the DSE methodology is a fully standard sequential C code. This code is the representation of the applications behavior from which the application parallelism is extracted.
- **Constraints file**: The constraints file is define by the user to set the specification constraints. The user define the parameters that the architecture needs to respect in terms of power consumption, area, energy dissipation, memory size, memory technology and operating frequency. The user also defines within this file the application execution time and the environmental constraints.

3.3.2.2 Code parallelization

The code parallelization extract from the application sequential C code, the task graph and its inherent parallelism (Figure 3.7). This step gives to PARSE the possibilities to exploit the application parallelism and adapt its solution (architecture, mapping and data placement) based on the user constraints and on the obtained performance.

On the example (Figure 3.7) the code parallelization extracts from the application sequential C code a task graph composed by three tasks.

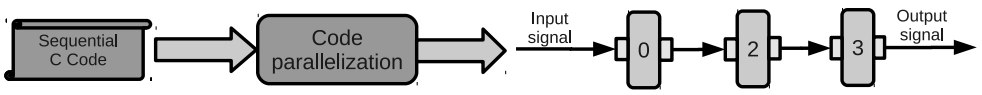


Figure 3.7 – Example of the operation done by the code parallelization tool, which from a sequential C code produce a task graph.

This NP difficult problem is under the work of this thesis and will rely on existing tools and related work.

3.3.2.3 Profiler

The application profiling is done to ease the selection of a set of cores model by the user task for each task of the application. The information provided give for each task the CPU time needed to run onto each COTS processors of the library.

These profiling information are provided automatically and only for the COTS processors. Indeed it is difficult from the application task graph to identify in an automated manner a code section that can be run onto data-flow accelerators. This is why the user when defining the parsimonious representation has the possibility to include data-flows accelerators based on its own experience (see section 3.3.2.4).

As an example and based on the application task graph depicted Figure 3.7 the profiler produces the table 3.2 which defines for each core if it is possible to run the task and the time needed to run it.

Table 3.2 – Exploration runtime of the three test applications.

	ip_0	ip_1	ip_2	ip_3	ip_4	ip_5
t_0	0.1s	1.8s	0.16s	1s	1.3s	Unsuitable
t_1	1.2s	0.2s	2s	1.7s	1.3s	3s
t_2	0.08s	1.15s	0.16s	2.2s	1.8s	1.6s

Based on this table and on the execution time constraints, the user to defines the dictionary of the parsimonious representation (see section 3.3.2.4). It is important to note that the profiler suppose zero cycle latencies to access the data.

3.3.2.4 Parsimonious representation

The profiling informations along with the tasks graph are used to point out to the user the bottlenecks at the architecture and application levels. Based on these informations the user can either modify its application, enrich the architecture library with new IPs or define a set of IPs for each tasks.

The definition by the user of a set of cores for each task is needed since the solution space to explore is huge. To reduce the size of the solution space and the associated exploration runtime the user then provides to PARSE a parsimonious representation of the solution space [144]. This point relies on his invaluable experience.

A parsimonious representation is given by the equation 3.4, where y is the solution matrix, x the parsimonious decomposition of y and D is the dictionary used for the decomposition. The corresponding matricial notation is given by equation 3.5.

$$y = D \times x \tag{3.4}$$

$$\begin{bmatrix} y_{0,0} & \cdots & y_{0,n} \\ \vdots & \dots & \dots \\ y_{m,0} & \cdots & y_{m,n} \end{bmatrix} = \begin{bmatrix} ip_0,t_0 & \cdots & ip_0,t_n \\ \vdots & \dots & \dots \\ ip_m,t_0 & \cdots & ip_m,t_n \end{bmatrix} \times \begin{bmatrix} x_{0,0} & \cdots & x_{0,m} \\ \vdots & \dots & \dots \\ x_{n,0} & \cdots & x_{n,m} \end{bmatrix} \tag{3.5}$$

In the context of PARSE y is the matrix solution of size N by M (where N is the number of tasks of the application and M the number of possible cores for the architecture) which represent the best compromise for the application and the architecture. D is the dictionary defined by the user which described the set of available cores for each task. x is the matrix automatically determined by PARSE to get the y matrix solution.

If we take as an example an execution time constraint of 1 second based on the application task graph of Figure 3.7 and the profiling informations of Table 3.2 the result obtained following the PARSE exploration is the one of equation 3.6.

The dictionary D defines that the task t_0 can be mapped onto the ip_0 and the ip_2, the task t_1 onto the ip_1 and the task t_2 onto the ip_0 and the ip_2. Based on the x matrix determined by PARSE we see that the find architecture is composed by three IPs, and the task t_0 is mapped on the ip_0, the task t_1 is mapped on the ip_1 and the task t_2 is mapped on the ip_2.

$$\begin{bmatrix} ip_0,t_0 & 0 & 0 \\ 0 & ip_1,t_1 & 0 \\ 0 & 0 & ip_2,t_2 \end{bmatrix} = \begin{bmatrix} ip_0,t_0 & 0 & ip_2,t_0 \\ 0 & ip_1,t_1 & 0 \\ ip_0,t_2 & 0 & ip_2,t_2 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.6}$$

3.3.2.5 PARSE

The exploration of the solution space within PARSE is based on three evolutionary algorithms. For the exploration of the architecture solution space a genetic algorithm (GA) [102] is used. The GA creates and evolves a heterogeneous population of architectural solutions. The generated solutions are composed of different cores, memory and architecture hierarchy and interconnect. Based on this architecture population the second step defines for each architectural solution a mapping of the application onto that architecture (Figure 3.5). To that end a tabu search algorithm is used to explore the solution space. For each potential architecture and mapping solution, a third evolutionary algorithm is triggered to explore the data placement solution space. The exploration is based on a GA to apply loop based transformation in order to define the most efficient data placement solution. This process is summarized as IT_x on Figure 3.5.

As stated above in the context of this thesis only the algorithms used to explore the mapping and data placement solution space are fully described and tested respectively into section 3.4, 3.5 and 3.6. The rest of this section then depicts the architecture exploration heuristic. To that end the function and operations realized by the architecture exploration heuristics is described along with the representation of the solution space in a form of a chromosome.

Architecture exploration:

The number of parameters of MPSoC architectures is huge and leads to an exponential increase of the number of potential solutions. The exploration of the solution space in an exhaustive manner even with a parsimonious approach is impossible in a reasonable amount of time. The introduction of heuristics to efficiently go through the solution space is then needed. In the context of the architecture solution space exploration, the uses of neighborhood heuristics allow the optimization of one criterion at a time which is penalizing. Indeed the architecture solution space exploration imposes the optimization of several design points at the same time (memory and architecture hierarchy, number and type of processing IPs, interconnect topology) and then requires the use of a multi-objective heuristics [62, 181]. The GA

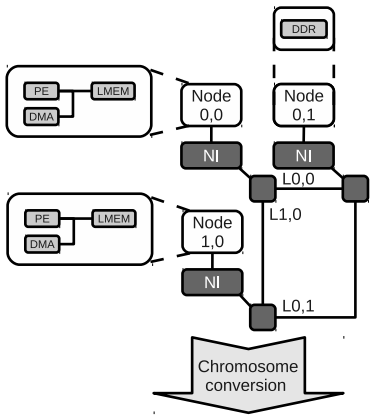
are then preferably chosen since they are more efficient than the other heuristics for multi-objective exploration [63, 64, 155].

The genetic algorithms are search heuristics that mimics the process of natural selection. The genetic algorithms are used to evolve a population toward the most promising regions of the solution space. Each individual or chromosome represents a potential solution to the problem.

Genetic algorithm transformations:

In order to define hardware efficient architecture the parameters that can be changed automatically by the GA at the platform and at the node level are the following: (1) The number and kind of IPs, (2) The memory size, (3) The memory hierarchy, (4) The architecture hierarchy, (5) The interconnect topology.

Chromosome coding:



Nb Node	3
Interconnect Type	NoC
Name of the interconnect	HERMES
Node 0,0	Active
Cluster based	No
Shmem based	No
Size of the shared memory	0 Kb
Nb PE	1
PE type	Microblaze
LMEM based	Yes

LMEM size	512 Kb
Nb interconnect link	2
Interconnect link 0	L0,0
Interconnect link 1	L1,0
Node 0,1	Passive
Node type	DDR
Nb interconnect link	2
Interconnect link 0	L0,0
Interconnect link 0	L0,1

Node 1,0	Active
Cluster based	No
Shmem based	No
Size of the shared memory	0 Kb
Nb PE	1
PE type	LEON
LMEM based	Yes
LMEM size	512 Kb
Nb interconnect link	2
Interconnect link 0	L0,0
Interconnect link 1	L0,1

Figure 3.8 – Conversion of an architecture to a chromosome. Within this architecture the three nodes are connected to a NoC. The NoC is the HERMES NoC [154]. Two nodes are active and constructed around the microblaze and LEON processors, while the third node gives access to a DDR memory.

The proposed chromosome is based on a variable chromosome size. Known chromosome size indeed reduces the possibilities to change the architecture organization and characteristics, which is unacceptable when doing architecture design space exploration [102]. With the proposed encoding, the size of the chromosome is variable in order to ease the addition or the removal of nodes onto the interconnect. As shown on Figure 3.8 each chromosome describe: (1) The number of node implemented onto the interconnect, (2) the type of the node, (3) the interconnect topology and how the nodes are connected with each other. Depending on the node the chromosome describe: (1) the organization of the node, (2) the size and type of memory implemented within the node and (3) the number and type of PE.

Cost function:

The cost function used by the GA is the same as the one used by the mapping and data placement heuristics and is detailed in section 3.4.

3.3.2.6 SystemC simulation

In the context of MPSoC architectures, it is quite difficult to precisely model the behavior of the architecture and of the application using an analytic approach. This lack of accuracy is a main drawback since the evolutionary algorithms can lead to sub-optimal solution. The use of a SystemC simulator is then needed to have a more accurate representation of the solution space.

The problem is then the time required to run the SystemC simulations. This is why the solution space is represented in a parsimonious manner to focus on the most promising regions of the solution space. Moreover to avoid the accurate exploration of uninteresting design points we propose a mixed evaluation method (See section 3.4) based on metrics to reduce the number of simulations.

3.3.2.7 Binary and HDL code generation

The binary code generator is used to provide to the user a parallelized binary code to run on top of the proposed architecture. The binary code generator takes as inputs the application task graph, the architecture model, the mapping and data placement representations.

The HDL code generation is done to provide the user with a hardware description of the architecture. Since PARSE in the context of the exploration has to test and combine different type of IPs together, the architecture is what we call a ”plug and play” architecture. The principle is to be able to use all the IPs library without putting any effort on the user side to interface the IPs. To that end the connection to the interconnect is abstracted by the means of a generic interface (Figure 3.9) called accelerator interface (AI) detailed in the chapter 4. All the accelerators of the library have to be compliant with this interface in order to easily construct the architecture and generate the HDL code.

Moreover in order to ease the task mapping and data placement resolution the AI gives to the connected IPs an autonomous behavior regarding the master processor once programmed. Indeed the AI thanks to its architecture and programming protocol release the master processor from the management of the task scheduling and data transfers.

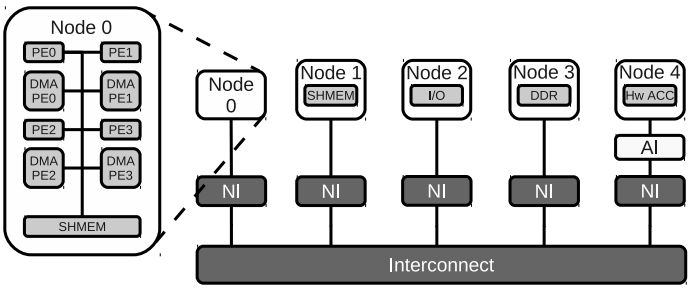


Figure 3.9 – Representation of how the AI connect the IPs onto the interconnect.

3.4 Mapping exploration

3.4.1 Introduction

The application mapping in a MPSoC is a key point to reach high performance. Since the application mapping is dependent of the interconnect, the memory size and bandwidth and of the processor efficiency, the impact of the application mapping has to be evaluated during the architecture definition in order to find the best compromise for the architecture and the application. An ideal task mapping being a mapping where the communication cost is minimal and where the time needed to run the task is greater or at least equal to the task communication time.

Depending on the time the tasks are assigned to the cores, the mapping techniques can be classified to be either static or dynamic [175].

The dynamic mappings perform the assignment and ordering of tasks at runtime. Dynamic mapping always tries to detect the performance bottleneck and distribute the workload among the processors. In case of static mapping, the mapping of task is performed off-line, before the application is running. For a given application and a target level of performance the static mapping always tries to define the best mapping at design time. In the context of this thesis the static mapping is considered. Indeed our aims with the proposed framework is to get performances evaluations and go through the solution space in order to identify the best compromise for both the architecture and the application.

The static mapping approach can be broadly classified to be either exact mapping or search based mapping [175].

The exact mappings produce optimal solution and have been extensively studied in the literature. In [158] an approach for mapping cores onto NoC based topology was proposed with the aims of ensuring an optimization tradeoff between the execution time, the resources occupation and the communication cost. In [99] a mixed integer linear programming (ILP) is proposed to define energy efficient application mapping. This work was further extended in [118] to find a tradeoff between computation and communication times. With the same philosophy [197] proposes an ILP formulation for energy consumption reduction tested on different benchmarks. In [69] the factors that produce network contention and delays have been analyzed and taken into account. However the main drawback of exact mapping approaches is the high CPU time needed to determine the solution. To overcome these limitations [187, 198] proposes to cluster the application graph. Despite the gain in CPU time this approach reduces the obtained performances.

Search based mappings on the other hand do not always produce an optimal solution, but aims to produce a solution close to the optimal by exploring a reduced number of design points. Depending on the search type and results there are two types of search based mapping algorithm: (1) deterministic and (2) heuristic based.

Branch and bound (BB) [136] belongs to the deterministic search algorithm. In [116] based on BB algorithm, an energy and performance aware mapping algorithm for NoC based architecture is proposed. Still based on the same approach [141] proposes a traffic balanced algorithm. However the deterministic search requires long exploration time as for the exact mappings.

The heuristic search algorithms on the other hand do not always find the same final solution, but are

constructed in a way to ensure the convergence to the most promising regions of the solution space. The heuristic search techniques can be classified into three categories: (1) the transformative heuristic, (2) the constructive heuristic without iterative improvement and (3) the constructive heuristic with iterative improvement. The Genetic Algorithm (GA) and the Particle Swarm Optimization (PSO) are the mainly used transformative heuristics. In [36] these heuristics are implemented for performance aware mapping, while in [46, 211] they are used for energy efficient mapping and for communication reduction in [70, 177].

The constructive heuristics without iterative improvement evolve one solution at a time. No optimization techniques are applied upon the initial solution to evolve toward a better solution. These solutions were used to define communication aware [163, 216] and energy aware [68, 199] mapping algorithm. While [179] have been proposed to define hardware cost effective NoC.

In contrast, the constructive heuristics with iterative improvement still evolve one solution at a time but iterative improvements are done upon the initial solution to always go toward a better solution. These algorithms have also been used to define communication [123, 148, 218] and energy [84, 85, 210] efficient mapping algorithm, as well as for hardware cost reduction of NoC [109].

The heuristic approach has proved to be scalable and particularly well suited to scale with the increase of the number of cores. The most efficient approach for the application mapping is the constructive heuristic [175] and more particularly the tabu search (TS) heuristic [101]. Indeed this heuristic has proven to converge in a more efficient manner compared to the other approach [76, 178, 191, 196]. The TS heuristics are then preferably chosen in the context of PARSE.

However, despite the quality of these techniques they are all based on analytic formulas to evaluate the application mapping. This is no more possible in the framework of modern MPSoC architectures. Indeed it becomes highly complex to accurately model the communication topology and architecture behavior of a MPSoC architecture with an analytic formula. Moreover with these approaches the data placement is determined after the task mapping which creates a separation of concerns between task and data parallelism and leads to sub-optimal solution.

In order to avoid a separation of concerns and have an accurate representation of the solution space the mapping and data placement have to be explored jointly. SystemC simulations are then required for accuracy purposes but come with high runtime overhead. However due to the size of the solution space each design points cannot be evaluated accurately due to the exploration runtime overhead.

To that end this work introduces a mixed evaluation heuristics based on metrics to determine the quality of a design point. Based on the metrics results the data placement solution space is explored for the most promising design points while the others designs points are evaluated thanks to an analytic formula to not affect the exploration process. This approach allows the reduction of the exploration runtime while maintaining an accurate representation of the solution space.

3.4.2 Tabu search

3.4.2.1 Presentation

The proposed tabu search reduce space (TSRS) framework for the mapping exploration is depicted in Figure 3.10. The inputs are a parallelized application task graph, an architecture model, the profiling informations and the constraints file. The first step of the proposed framework is to compute the metrics used to evaluate the quality of the design points.

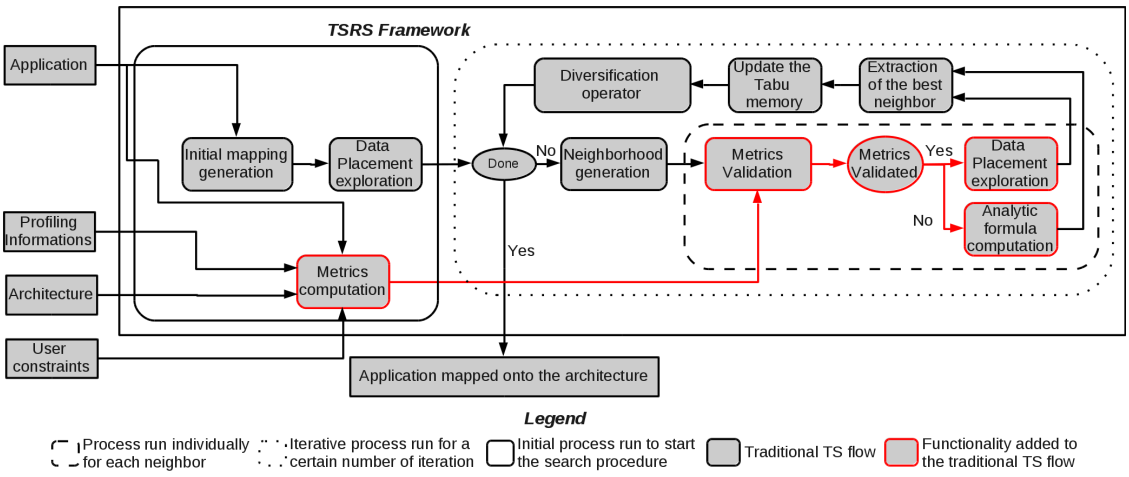


Figure 3.10 – TSRS framework used to explore the application mapping solution space.

To avoid the accurate exploration of uninteresting design points a set of metrics is automatically computed prior to the mapping exploration. These metrics are computed based on the application task graph, the architecture model, the profiling informations and the user constraints. Based on these metrics the TS algorithm is able to avoid the accurate exploration of uninteresting design points (see section 3.4.2.4 and 3.4.2.5).

From these inputs, the TS algorithm is triggered to automatically explore the mapping solution space and an initial solution is randomly created.

TS is a heuristic that evolves only one solution at a time. The change from one configuration s to another one t is done in three steps: (1) All neighbors of s are constructed from elementary movement (see section 3.4.2.2), (2) Each design point quality is evaluated thanks to the metrics to avoid the accurate exploration of uninteresting design points (see section 3.4.2.4), (3) Based on the metrics results the data placement is evaluated for the most promising design points while the other design points are evaluated with a analytic formula to not affect the exploration process (see section 3.4.2.5 and 3.4.2.6). The neighbor that succeeds to s (i.e t) is the neighbor producing the best task mapping and data placement. The data placement is explored in parallel of the task mapping.

To avoid a cycle phenomenon and return to a previously selected solution, a memory of Tabu solution is updated at each iteration. The Diversification operator is also implemented. This operator avoid that a large region of the solution space remains completely unexplored [101] by restarting the TS process if the best solution stays constant during a certain number of iteration.

3.4.2.2 Neighborhood generation

The neighborhood $N(s)$ is generally defined as the set of neighboring solutions of the current configuration s. The set $N(s)$ is the set of valid solutions obtained by applying to s a movement m belonging to the set M of possible movements.

A movement in the context of the TSRS framework is the remapping of a task originally located to a $node_i$ to $node_j$. An example is given Figure 3.11 where the task 1 is moved from the $node_3$ to the $node_0$. In the context of PARSE only one task is moved at a time in order to avoid the generation of a huge solution space.

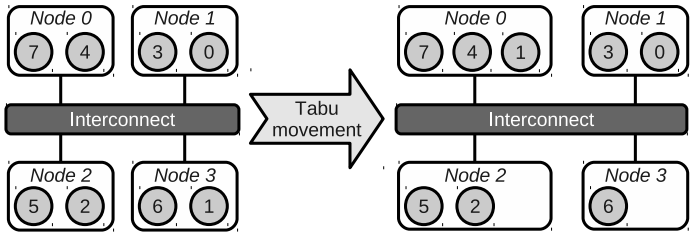


Figure 3.11 – Example of Tabu movement used to generate the neighborhood. One task is moved at a time. Here the task 1 is moved from the $node_3$ to the $node_0$.

3.4.2.3 Diversification operator

The diversification operator encourages the search to explore unvisited regions. This operator is triggered if the best solution stays constant for many iterations. The diversification operator implementation follows the one proposed by [101] and randomly redefines the current solution and reset the tabu memory if the best solution stays constant for a number of iteration equal to the number of iteration defined by the diversification operator level.This diversification operator level is set depending of the application complexity.

3.4.2.4 Metrics

Distance metric

This metric is intended to identify quickly a problem in communication requirements for the current mapping. Based on the application task graph and architecture topology graph, the first step is to compute the raw throughput (thg) of the application, which depends on the tasks data production and on the execution time constraint set by the user (Eq. 3.7).

$$thg = \sum_{i=0}^{nb_t-1} \sum_{\substack{j=0 \\ j\neq i}}^{nb_t-1} \frac{t\_prod_{i,j}}{et\_cst}; \tag{3.7}$$

In parallel the mean path ($mean\_path$) between all the nodes of the architecture is computed (Eq

.3.8). Where the distance between $node_i$ and $node_j$ ($dist_{i,j}$) is computed based on Dijkstra algorithm [67].

$$mean\_path = \frac{\sum_{i=0}^{nbn-1} \sum_{\substack{j=0 \\ j \neq i}}^{nbn-1} dist_{i,j}}{nbn * nbn}; \tag{3.8}$$

Based on the throughput and on the mean path values, the mean communication ($mean\_com$) for the application is obtained (Eq. 3.9).

$$mean\_com = \frac{thg \times mean\_path}{nbt * nbt}; \tag{3.9}$$

In parallel of this computation, the communication cost ($t\_com_{i,j}$) to transfer the data from task t_i to task t_j is computed (Eq. 3.10). Where $dist_{mi,mj}$ represent the distance between the task t_i mapped onto the core mi and the task t_j mapped onto the core mj. The minimum communication cost to transfer the data from task t_i to task t_j ($t\_min\_com_{i,j}$) is also computed in order to handle the case where two tasks which highly communicate together are responsible of most of the communications over the platform (Eq. 3.10 and 3.11). $t\_min\_com_{i,j}$ represent a mapping on which task t_i and task t_j are mapped on the same node.

$$t\_com_{i,j} = \frac{t\_prod_{i,j}}{et\_cst} \times dist_{mi,mj}; \tag{3.10}$$

$$t\_min\_com_{i,j} = \frac{t\_prod_{i,j}}{et\_cst}; \tag{3.11}$$

Algorithm 1 Distance metric computation

```
for i ← 0 to nbt − 1 do
  for j ← 0 to nbt − 1 do
    if i==j then

    else
      if t_com_{i,j} < mean_com or t_min_com_{i,j} > mean_com then
        d_met = 1
      else
        d_met = 0
        return
      end if
    end if
  end for
end for
```

The distance metric is validated (Algorithm 1), if the communication cost to transfer the data from task t_i to task t_j is smaller than the average communication cost ($mean\_com$). Or if the minimal communication cost to transfer the data from t_i to t_j ($t\_min\_com_{i,j}$) is greater than the average communication cost.

Otherwise the mapping is not validated, the solution penalized and the design point considered as uninteresting. This metrics represent a problem in communication for the mapping.

Node load metric

The node load metric (l_{met}) compute the load of each active node based on the profiling and mapping informations. The first step compute the load of each node ($node_{load\_i}$) (Eq.3.12):

$$node_{load\_i} = \sum_{i=0}^{nbt-1} t_i et, \forall\ t_i \text{ if } mi; \tag{3.12}$$

A mapping is then valid if the load of each active node ($node_{load\_i}$) is smaller than the global execution time constraints (et_cst)set by the user for the application (Algorithm 2). Which means that no core needs all the time to compute its assigned tasks. These solutions, even if not interesting, are kept in order to keep the possibility of explore the entire solutions space.

Algorithm 2 Node load metric computation

```
for i ← 0 to nbc − 1 do
  if node_load_i > et_cst then
    l_met = 0
    return
  else
    l_met = 1
  end if
end for
```

3.4.2.5 Analytic performance

In order to avoid the accurate exploration of uninteresting design point, each point of the solution space is evaluated regarding the two above metrics. If at least one of the two metrics is not valid the analytic performance is then computed.

In order to compute the analytic performance ($perf_{\text{ana}}$) we first compute for each mapping the maximum path between two nodes of the architecture (max_path) (Eq. 3.13).

$$max\_path = max_{i=0,j=0,j\neq i}^{nbn-1,nbn-1} dist_{i,j}; \tag{3.13}$$

Based on the maximum path value, the maximum communication cost (max_com_{cst}) for the proposed application and architecture models is computed (Eq. 3.14).

$$max\_com_{\text{cst}} = \sum_{i=0}^{nbt-1} \sum_{\substack{j=0 \\ j\neq i}}^{nbt-1} \frac{t\_prod_{i,j}}{et\_cst} \times max\_path; \tag{3.14}$$

The communication cost (com_{cst}) of the proposed mapping is then computed and normalized between 0 and 1 according to the maximum cost value (Eq. 3.15).

$$com_{cst} = \frac{\sum_{i=0}^{nbt-1} \sum_{\substack{j=0 \\ j \neq i}}^{nbt-1} \frac{t\_prod_{i,j}}{et\_cst} \times dist_{mi,mj}}{max\_com_{\text{cst}}}; \tag{3.15}$$

In parallel of the communication cost computation, the load of each node of the architecture is determined and the maximum execution time (max_et) of the application is computed following the Eq. 3.16.

$$max\_et = \sum_{i=0}^{nbt-1} t_i et; \tag{3.16}$$

From these results the execution time cost ($exec_{cst}$) is computed (Eq.3.17). The Eq.3.17 extracts the runtime needed by the most loaded node and normalizes this value according to the maximum execution time of the application.

$$exec_{cst} = \frac{\max_{i=0}^{nbn-1}(node_{\text{load\_i}})}{max\_et}; \tag{3.17}$$

Based on the execution time and on the communication cost the analytic performance is computed following the Eq. 3.18. If either the distance metric or the node load metric is not met a penalty of 0.5 is added to the analytic performance. If both metrics are not met the analytic performance is equal to a penalty of 1 (Eq. 3.18.3). The Eq. 3.18.1 represents a solution where the node load metric is not met and the solution penalized. The Eq. 3.18.2 represents a solution where the distance metric is not met and the solution penalized.

$$perf_{\text{ana}} = \begin{cases} \frac{\alpha_2}{2} \times com_{\text{cst}} + 0.5; & \text{if } d_{\text{met}}\text{=1 and } l_{\text{met}}\text{=0 (1)} \\ \frac{\alpha_2}{2} \times exec_{cst} + 0.5; & \text{if } d_{\text{met}}\text{=0 and } l_{\text{met}}\text{=1 (2)} \\ \alpha_2 \times 1; & \text{if } d_{\text{met}}\text{=1 and } l_{\text{met}}\text{=1 (3)} \end{cases} \tag{3.18}$$

The penalty is included between 0.5 and 1 to ensure that a solution that does not respect the metrics is not preferred regarding a solution that respects the metrics. Moreover we have two values for the penalty (0.5 and 1) to ensure that a solution that respects at least one metric is preferred instead of a solution that does not respect any metrics. Finally α_2 is a parameter set by the user to guide the exploration (See section 3.4.2.6).

3.4.2.6 Cost function

The cost function used to explore the solution space is common to the architecture, the mapping and the data placement heuristics. This provides to the three algorithms a common and uniform represen-

tation of the solution space which eases the convergence to the most promising regions of the solution space.

Eq. 3.19 presents the proposed cost function (f). The Eq. 3.19.1 is only computed when the memory usage rate is bellow 1 and the two metrics (l_{met} and d_{met}) are met. The memory usage rate (mem_{rate}) is automatically computed for each memory of the architecture and represents the memory occupancy rate. The memory usage rate is computed after the data placement (see section 3.5) in order to check if the mapping and data placement respect the memory size constraint and to see if the memory space is sufficient to store all the data needed by the application. If a solution has a memory usage greater than 1 (i.e there is no sufficient place in memory) the cost function is not computed and the solution removed from the solution space.

$$f = \begin{cases} \alpha_1 \times arch_{\text{cst}} + \alpha_2 \times perf_{\text{cst}}; \text{ if } d_{\text{met}}=1 \text{ and } l_{\text{met}}=1 \text{ and } mem_{\text{rate}} \leq 1 \text{ (1)} \\ \alpha_1 \times arch_{\text{cst}} + perf_{\text{ana}}; \text{ if } d_{\text{met}}=0 \text{ or } l_{\text{met}}=0 \text{ (2)} \end{cases} \tag{3.19}$$

The Eq. 3.19.1 evaluate for each solution its architectural cost ($arch_{\text{cst}}$) and its performance cost ($perf_{\text{cst}}$).

The architectural cost is computed in order to choose for two equivalent solutions the most efficient one.

The performance cost is computed based on a SystemC simulation. The SystemC simulation takes into account (1) the task mapping, (2) the scheduling, (3) the data transfers, (4) the memory bandwidth, (5) the interconnect bandwidth, (6) the IPs computing power, (7) the application parallelism.

From the SystemC simulation the performance cost is to computed following the Eq 3.1 and is equal to the maximum runtime value for a task (Eq 3.20).

$$perf_{\text{cst}} = \max_{i=0}^{nbt-1}(t_i run); \tag{3.20}$$

If one of the two metrics is not met the cost function is based on the analytic performance (Eq 3.19.2).

Finally the α_1 and α_2 parameters (Eq. 3.21) are set by the user to guide the exploration to solutions which are more hardware economic (α_1) or high performance (α_2).

$$\alpha_1 + \alpha_2 \leq 1; \tag{3.21}$$

3.4.3 Experimental results

To demonstrate the validity of the mapping exploration heuristic, the proposed TSRS framework is tested in a standalone mode (without the call to the data placement heuristic) and the results are compared with the one presented into [175].

To that end the benchmarks and architecture models implemented are exactly the same. Moreover the same rules are used for the quality evaluation of the obtained solution. The benchmarks implemented

are: The PIP, the MPEG-4, the VOPD, the DVOPD, the MWD, the 263enc mp3dec, the mp3enc mp3dec and the 263dec mp3dec benchmarks which can be found in annex A.3 [175].

The number of nodes of the architecture topology graph varies from 8 to 64. All the nodes are able to run all the tasks. The architectures are based on a mesh NoC for the interconnect topology. The benchmarks architecture models can be found in annex A.4 and in [175].

Moreover the results obtained by the TSRS framework are compared with a classical TS (CTS) framework which is only based on SystemC simulation for performance evaluation and does not use the metrics to evaluate the design points.

The experiments were performed on an Intel i5 quad core processor running a 2.5GHz with 8 GB of RAM.

3.4.3.1 Experimentations parameters

Application benchmark evaluation

In order to compare the performance of our approach with the state of the art the same evaluation method as the one defined in [159, 175] is used. To that end we introduce the following hypothesis:

Hypothesis 1: A mapping of the task graph $G(T, E)$ onto the architecture graph $P(M, F)$ is defined by the function map: $T, E \rightarrow U$ where U represent the mapping solution, such that, $\forall t_i \in T, \exists node_j \in M \quad and \quad map(t_i) = node_j$ and $\forall e_{i,j} \in E, \exists f_{i,j} \in F$ and $bw_{i,j} \geq t\_prod_{i,j}$. The function associates task t_i to $node_j$ and ensure that 1) all tasks are mapped to a core 2) all communications are ensured. Due to the fact that the load of a task is equal to the processing capabilities of a core, a mapping is defined only when the number of cores is greater or equal to the number of tasks ($nbt \leq nbc$). This limitation is not due to TSRS but used for fair evaluation purpose.

Hypothesis 2: The quality of such a mapping is defined in terms of the total communication cost [159], and is computed following the Eq. 3.22.

$$bench\_com_{cst} = \sum_{i=0}^{nbt} \sum_{j=0}^{nbt} t\_prod_{i,j} * dist_{i,j} \tag{3.22}$$

This evaluation of the mapping quality taken by the state of the art is quite simple and does not accurately model the complexity of the underlying architecture. This is why we explore the solution space with SystemC simulation. And at the end of the exploration the chosen solution is evaluated by the use of the Eq.3.22 to compare our results with the one of the state of the art.

Tabu search algorithm settings

The parameters used to configure the two TS algorithms (CTS and TSRS) are summarized into Table 3.3.

These parameters settings have been chosen from a set of trials demonstrating that the accuracy of the exploration was not improved by a further increase of these values as advised by [101].

Table 3.3 – Tabu search parameters settings

		TS parameters		
	Benchmark	Mem size	Nb of it	Divers ope rate
TSRS	PIP	6	10	6
	VOPD	250	1000	150
	MPEG-4	250	1000	150
	DVOPD	1000	5000	700
	MWD	250	1000	150
	mp3enc mp3dec	250	1000	150
	263enc mp3dec	250	1000	150
	263dec mp3dec	250	1000	150
CTS	PIP	8	10	4
	VOPD	500	1000	200
	MPEG-4	250	1000	150
	DVOPD	1750	6000	800
	MWD	250	1000	150
	mp3enc mp3dec	250	1000	150
	263enc mp3dec	250	1000	150
	263dec mp3dec	250	1000	150

It can be seen that the number of iterations (Nb of it) is limited in order to keep the exploration in a reasonable amount of time. Moreover the values of these parameters are adapted depending on the application complexity.

The memory size (Mem size) defines the number of iteration a solution stays a Tabu solution. The diversification operator rate (divers ope rate) defines after how many iteration with the same best solution, the TS search process is restarted. This study is given in annnex A.2.

Finally in the context of these experiments the α parameters have no influence on the final solution. This is due to the benchmark hypothesis. Indeed since one task has to be mapped to one node the architectural cost is equal for all the solutions and no optimization on the architectural side can be done. Due to these reason α_1 is equal to 0 and α_2 is equal to 1.

3.4.4 Results

The results given in this section suppose that the architecture topology graphs given in inputs of the framework are optimal (Ideal number of nodes, interconnect and memory size). Setting these architecture topology graphs is beyond the scope of this section.

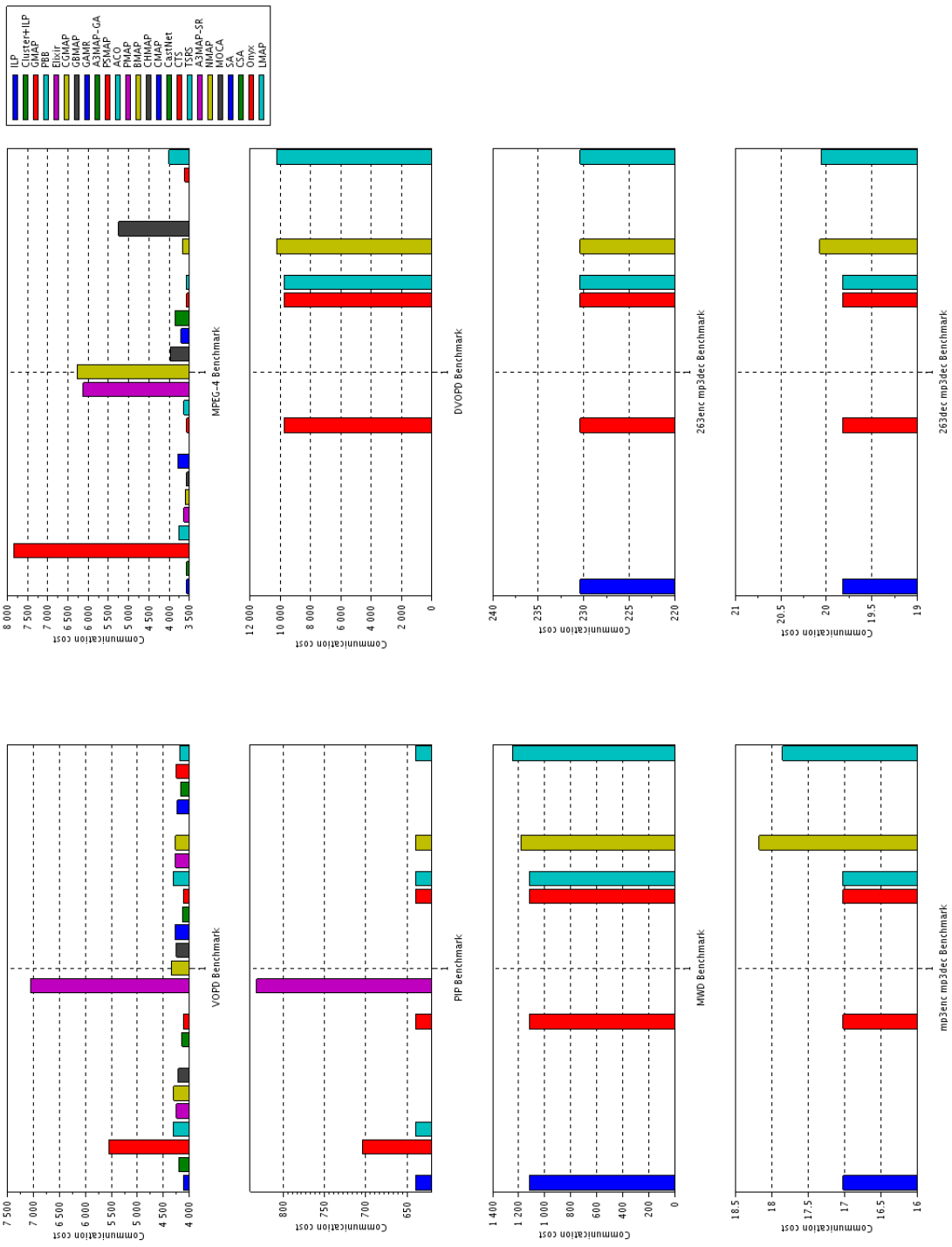


Figure 3.12 – Communication cost for the ILP [197], Cluster + ILP [198], GMAP [114, 115, 116], PBB [175], Elixir [174], CGMAP [175], GBMAP [193], GAMR [96], A3MAP-GA [122], PSMAP [175], ACO [209], PMAP [131], BMAP [179],CHMAP [192], CMAP [68], CastNet [199],A3MAP-SR [122], NMAP [175], MOCA [186], SA [145], CSA [145], Onyx [123], LMAP [176], CTS and TSRS mapping algorithms on the tested benchmarks

3.4.4.1 Communication cost

The results characterizing the communication cost are given Figure 3.12. These results are extracted from the study done into [175]. In this table the TSRS and CTS framework are compared against the best solutions of the state of the art. All the mapping techniques are evaluated regarding their communication cost, computed following the evaluation method proposed in [175].

The results from this figure show that the mapping provided for all the benchmarks (except the VODP and DVOPD) by the CTS and TSRS are always the same as the one get by the ILP approach which is an exhaustive approach and thus propose the optimal solutions. These benchmarks show the efficiency of our frameworks for different levels of application and architecture complexity. Indeed our framework is able to efficiently adapt its mapping based on the architecture and application model to map the highly communicating tasks close to each other, this in order to reduce the long communication distance and the bottlenecks onto the network.

Regarding the results obtained for the VOPD benchmark, the CTS is again optimal while the solution obtained by the TSRS is a bit less efficient. This is due to the fact that when the number of task is equal to the number of nodes the chosen hypothesis (one task per node) and the metrics limit the search process of the TSRS. This is especially true when the application complexity is important.

Indeed as depicted Figure 3.13.A the first mapping that will be produced is a mapping where each task is mapped onto one node. Since this is the only solution where any node is overloaded and any solution penalized. Due to the neighborhood generation process, which move one task at each iteration the next current solution is going to be penalized (Figure 3.13.B), since two tasks are mapped onto the same node. Because of the tabu memory and of the metrics limitations which favor the solution where any node is overloaded, the next current solution will map the task originally located onto the overloaded node to the free node (Figure 3.13.C).

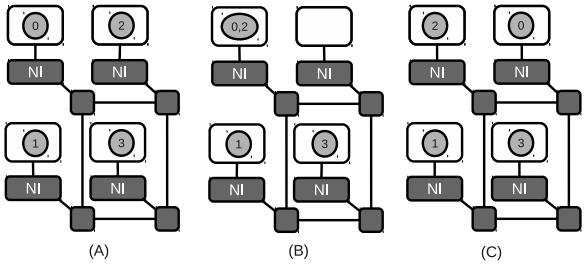


Figure 3.13 – Limit of the TSRS on the VOPD benchmark. From the initial solution (A), the task 2 is moved on the same node as the task 0 because all the solution are penalized and this is the most efficient penalized solution (B), the next solution (C) is then to move the task 0 to the free node since this is the only solution which is not penalized or Tabu.

As shown on this example due to the metrics constraints, the exploration process is limited and not as efficient as possible when the number of tasks is equal to the number of nodes. However the hypothesis that imposes to have one task mapped onto one node is specific and the quality of the obtained results are still kept high.

Regarding the results obtained for the DVOPD benchmark it is no more possible to run an exhaustive approach due to the complexity. However the CTS and TSRS framework perform as well as the best

algorithm found into the state of the art. These results prove that the proposed framework scale well with the increase of the number of nodes since the DVODP is mapped onto an architecture composed of 64 nodes. Finally we can also see that we obtain the same level of performance as the PSMAP [175] algorithm. However as stated the PSMAP is based on a simple analytic formula which does not allow to accurately model the MPSoC architecture and scale with the technology improvement.

3.4.4.2 Exploration runtime

Regarding the runtime of the exploration the results are resumed in Table 3.4. The CTS and TSRS mapping algorithm are only compared to the NMAP [175], LMAP [176], PSMAP and ILP [197] because the exploration runtime for the other algorithms are not available.

From this table we can see that the difference in terms of computing time between the two TS frameworks is important and evolves exponentially with the number of tasks. This is due to the fact that the CTS framework is only based on SystemC simulation for performance evaluation. Since most of the time is spent for the validation of the mapping and the generation and execution of the SystemC simulator the TSRS framework take benefit of the metrics evaluation to save time. This proves the interest of using the proposed metrics to only generate for the most interesting design points a SystemC simulation. Moreover the exploration runtime needed by the TSRS framework on all the benchmarks are close to the performance obtained by the state of the art algorithms except for the MPEG-4 benchmark. This proves the efficiency of the proposed metrics to avoid the unnecessary SystemC generation and save time. Indeed the NMAP, LMAP, PSMAP and ILP are based on a simple analytic formula for performance evaluation while our evaluation requires the generation of a SystemC simulator which is much more time consuming. The SystemC simulation is however needed in order to be able in the future to have an accurate representation of the MPSoC architecture and to scale with the increase of the number of nodes. Since we have almost the same exploration runtime this proves that the TSRS is able to focus on the most promising regions of the solution space to scale with the technology improvement. For the MPEG-4 benchmarks the TSRS is less efficient than the state of the art mapping algorithms. This is due to the fact that the metrics do not identify as aggressively as for the other benchmarks the uninteresting design points. However the exploration is kept in a reasonable amount of time, the result is optimal and the gain compared to a CTS is quite important.

Finally when the benchmark complexity increase as for the DVOPD we see that the TSRS framework scale well with the increase of the number of nodes. Indeed we still propose the same level of performance as the state of the art algorithm while offering a more accurate representation of the platform behavior.

Table 3.4 – Exploration runtime of TS on the seven tested benchmarks

Mapping algorithm	NMAP		PSMAP		ILP		CTS		TSRS	
	CPU time in s	Overhead norm to NMAP	CPU time in s	Overhead norm to NMAP	CPU time in s	Overhead norm to NMAP	CPU time in s	Overhead norm to NMAP	CPU time in s	Overhead norm to NMAP
VOPD	0.0240	1.0	0.260	10.8	4474.730	186447.0	147.7	6154.0	0.311	13.0
MPEG-4	0.0160	1.0	0.040	2.5	21.53	1345.6	147.6	9225.0	43.09	2693.0
PIP	0.010	1.0	0.010	1.0	1.280	128.0	0.250	25.0	0.0140	1.40
DVOPD	0.380	1.0	14.287	37.6	NA	NA	7020	18473	14.8	38.94
MWD	0.016	1.0	0.020	1.25	200.510	12531.88	0.328	20.5	0.0274	1.71
263enc mp3dec	0.012	1.0	0.268	22.3	191.910	15992.5	1.728	144	0.292	24.3
mp3enc mp3dec	0.016	1.0	0.320	20	1432.430	89526.875	2.914	182.125	0.332	20.75
263dec mp3dec	0.016	1.0	0.260	16.25	4895.250	305953.125	2.722	170.125	0.310	19.375

3.5 Data placement exploration

3.5.1 Introduction

In the embedded system domain, the data placement is a key point to reach both high performances and to define hardware efficient architectures but the number of potential solution evolves exponentially likes a Stirling number of the second kind [171].

Since the data placement is dependent of the interconnect, of the memory size and bandwidth and of the processor efficiency, the impact of the data placement has to be evaluated as soon as possible in the development process of the architecture. In order to avoid the definition of incorrectly sized architecture. An ideal data placement being a placement where the data transfers are minimal and where the time needed to process the data is greater or at least equal to the time needed to transfer the data.

However, in current approaches the data placement is decided during the application code generation step. This vision creates a separation between data and task parallelism which leads to sub-optimal solutions. Loops based analysis and synthesis methods have been largely studied at the processor level to explore the memory reuse [40, 117, 219], the instruction level parallelism [34, 185] and the redundant memory traffic [219]. Nevertheless, most of these works fail to capture the interaction between nested loops [51]. This is a main drawback in the context of deeply nested loops since a wrong data placement will be repeated several times and the penalty occurred will lead to a big overhead on the application runtime.

Approaches based on SDFG were proposed to explore the interconnection network and the memory hierarchy [217, 97, 137]. However the SDFG are not well suited to efficiently exploit the data parallelism of an application [72]. Consequently they are not well suited to measure the impact of loop transformations that are usually used in the context of data placement.

Approaches based on the Array-OL formalism were also proposed. However the approaches based on this formalism propose simplified communication models [72, 71, 41], which do not reflect the complexity of MPSoC architectures and limits their use to specific applications and architectures. Innovative approaches based on reconfigurable architectures were also proposed in order to explore the data placement [126, 127, 128]. Despite the good results provided by these works, their uses are limited to reconfigurable architectures and to data flow oriented applications.

Works constructed around evolutionary algorithms were also defined in order to efficiently go through the solution space. However, most of these approaches construct their architecture model around a shared bus for the interconnect network [146, 43, 72], which limit the use of these methods in context of MPSoC architectures [206], where the interconnect is more and more often a NoC. Furthermore some approaches do not take into account the architectural aspects [40] and only focus on improving the application implementation. Finally works [190, 87] exploring the data placement into grid systems were also proposed, but their hypothesis and constraints are not possible for the embedded systems.

Nevertheless these works have shown than the evolutionary algorithm and more particularly the GA are well suited to go through the solution space and converge efficiently to the promising regions. This is why in the context of PARSE the GA heuristic is preferably chosen.

However this work in contrast with the state of the art does not limit the architecture interconnect

to the bus or to the point to point communication model but also include the NoC paradigm. Moreover the considered application can be either data stream or based on more complex communication model between the tasks. Finally a new specific operator is introduced to help the genetic algorithm to converge to the most promising regions of the solution space.

3.5.2 Genetic algorithm with fusion operator

3.5.2.1 Presentation

The proposed genetic algorithm with fusion operator (GAFO) framework for the data placement exploration is depicted in Figure 3.14. The inputs are the application tasks graph along with the architecture model and the mapping model provided by the TSRS framework (see section 3.4).

Based on these inputs an initial population is randomly created and the solution space explored.

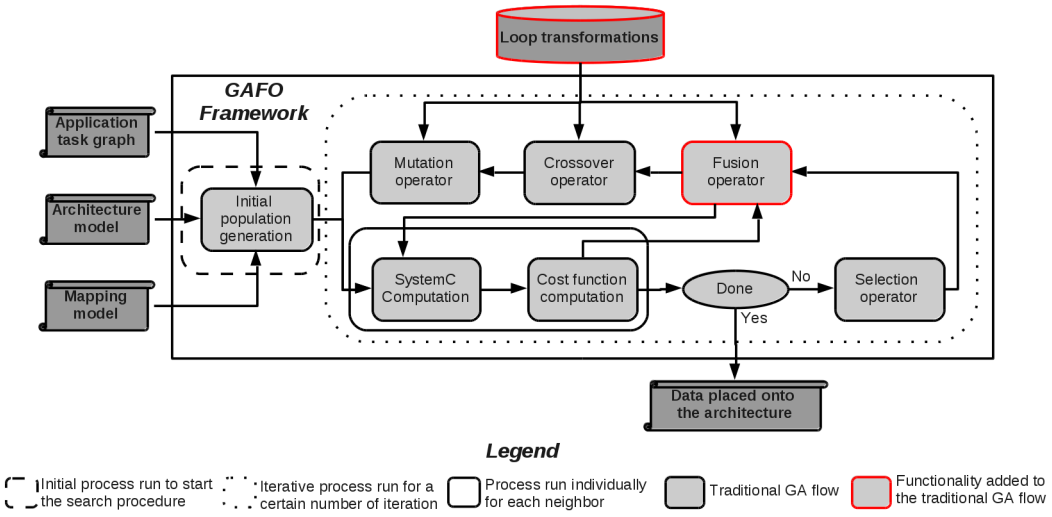


Figure 3.14 – GAFO framework used to explore the application data placement solution space.

The selection operator is used to evolve and improve the population across the generation. The selection is needed to choose the individuals that guarantee the improvement of the solution quality. The crossover is used to bring diversity into the population by manipulating the chromosome structure. The crossover creates new chromosome sequences for the childrens inherited from the parents. This operation implies based on probability set by the user. The mutation is an occasional random change of the value of one or more gene of a chromosome. The mutation is used to keep the diversity into the population and avoid to being stucked in local optimum. These operations are implemented following the classical approach described in [102].

The cost function used by the GA is the same as the one presented in section 3.4. Moreover in the context of this thesis a fusion operator (FO) is defined and added to ease the convergence of the GA to the most promising regions of the solution space by exploring the neighborhood of the current solution.

The loop transformations that can be applied to the application model are:

1. Task fusion [73] which merges iterative tasks within the same iteration space (Figure 3.15.A). The iteration space describes the number of iterations needed to consume the input/output array, how the input/output arrays are produced/consumed, and the pattern mappings specific to each input/output.
2. Tiling which add a level of depth to a loop nest (Figure 3.15.B). This operation automatically done allow the double buffering mechanism [215] within an iteration space. The double buffer mechanism is used to enable an overlap of the computation time with the communication time, in order to improve the performance.
3. The creation of a communication for: (1) data transfer between processing nodes and (2) data-reorganization if the data have to be read on another axis than the write axis. These operations are mapped onto the direct memory access (DMA) of the architecture (Figure 3.15.C).

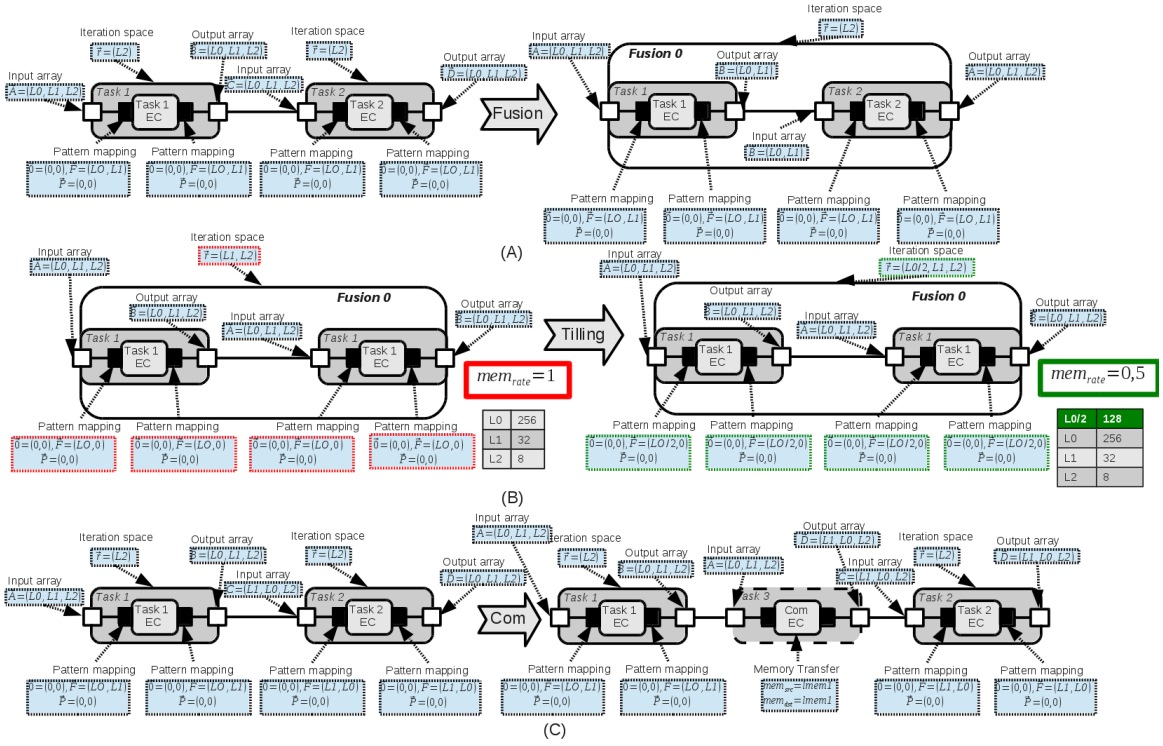
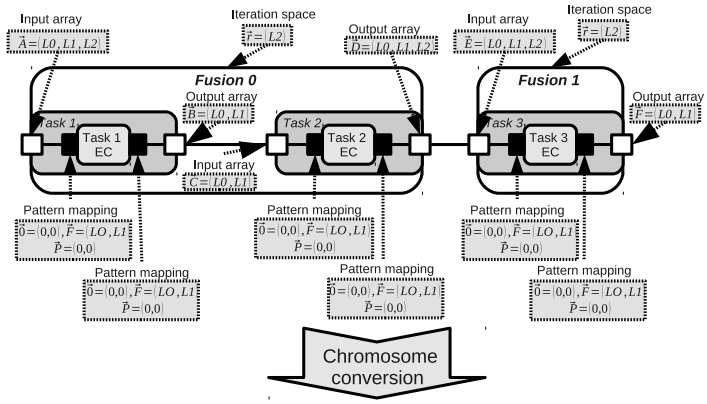


Figure 3.15 – Loop transformations that can be applied to the application model. (A) Task fusion to merge tasks into the same fusion, (B) Tilling change add a level of depth to a loop nest, (C) The creation of a communication for data reorganization or data transfer.

3.5.2.2 Chromosome encoding

The first step when building the chromosomes is to randomly allocate all the tasks to a fusion. The validity of each fusion is then checked and invalid fusions are corrected by removing conflicts. A fusion is said to be valid if the data dependency within the fusion are respected.

Once all the fusion validated, the loop transformations are applied to each fusion based on the estimated memory occupancy rate. Our approach when applying the loop transformations is to maintain the data locality as much as possible. One constraint is to keep the occupancy rate of each memory below 50%. The memory occupancy rate has to be under 50% in order to implement the double buffering mechanism. Once all the tasks are allocated to a fusion the tiling and communication creation operations are applied to ensure the data placement validity. The chromosome are then evaluated (i.e. computation time obtained).



Nb Fusion	2
Fusion Id	0
Fusion input array	$\vec{A}=[L0,L1,L2]$
Fusion output array	$\vec{D}=[L0,L1,L2]$
Fusion iteration space	$\vec{r}=[L2]$
Nb task fuse	2
Task id	1
Task 1 nb input	1
Task 1 input 1 data location	LMEM 0
Task 1 input 1 array	$\vec{A}=[L0,L1,L2]$
Task 1 input 1 pattern mapping	$\vec{0}=[0,0],\vec{F}=[L0,L1],\vec{P}=[0,0]$
Task 1 nb output	1
Task 1 output 1 data location	LMEM 0
Task 1 output 1 array	$\vec{B}=[L0,L1]$
Task 1 output 1 pattern mapping	$\vec{0}=[0,0],\vec{F}=[L0,L1],\vec{P}=[0,0]$

Task id	2
Task nb input	1
Task 2 input 1 data location	LMEM 0
Task 2 input 1 array	$\vec{C}=[L0,L1]$
Task 2 input 1 pattern mapping	$\vec{0}=[0,0],\vec{F}=[L0,L1],\vec{P}=[0,0]$
Task 2 nb output	1
Task 2 output 1 data location	LMEM 0
Task 2 output 1 array	$\vec{0}=[0,0],\vec{F}=[L0,L1],\vec{P}=[0,0]$
Task 2 output 1 pattern mapping	$\vec{D}=[L0,L1,L2]$
Fusion Id	1
Fusion input array	$\vec{E}=[L0,L1,L2]$
Fusion output array	$\vec{F}=[L0,L1,L2]$
Fusion iteration space	$\vec{r}=[L2]$
Nb task fuse	2
Task id	1

Task 3 nb input	1
Task 3 input 1 data location	LMEM 0
Task 3 input 1 array	$\vec{E}=[L0,L1,L2]$
Task 3 input 1 pattern mapping	$\vec{0}=[0,0],\vec{F}=[L0,L1],\vec{P}=[0,0]$
Task 3 nb output	1
Task 3 output 1 data location	LMEM 0
Task 3 output 1 array	$\vec{F}=[L0,L1]$
Task 3 output 1 pattern mapping	$\vec{0}=[0,0],\vec{F}=[L0,L1],\vec{P}=[0,0]$

Figure 3.16 – Conversion of a data placement to a chromosome, within this chromosome two tasks are merged within the same iteration space and one task is allocated into its own iteration space.

A chromosome (Figure 3.16) is composed of several iteration spaces. Each iteration space is at least made of one EC. The chromosome also details where the data necessary to fire each EC are located. This representation captures all the information necessary to reconstruct the input files needed by the SystemC simulator.

3.5.2.3 Fusion operator

The FO is used to increase the data locality by exploring the neighborhood of each chromosome. For each chromosome the FO evaluate within the chromosome if it is beneficial to apply the task fusion operation within the chromosome Figure 3.17.

The FO operating principle is the following: (1) The neighborhood of the current chromosome is generated, (2) The validity of the generated chromosomes is checked, (3) The valid chromosomes are

evaluated, the invalid chromosomes are removed from the solution space (4) If several iteration spaces can be merged with benefit the best fusion is applied, if no benefit is found the chromosome is left unchanged. The FO is not applied to each chromosome but depending on a probability set by the user in order to keep the exploration in a reasonable amount of time.

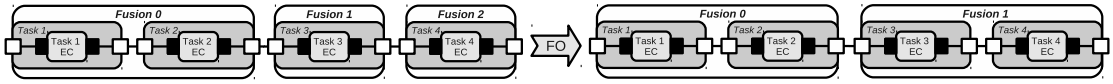


Figure 3.17 – Operating principle of the fusion operator.

3.5.3 Experimentation

To demonstrate the validity of the data exploration framework this latter is tested in a standalone mode without the call to the TSRS framework. This means that the task mapping provided in input is supposed to be optimal.

Three applications have been implemented. The chirp (Figure 3.18) [170], is a simple signal processing application with a low complexity, the jpeg encoder (Figure 3.19) [71] the well known image processing application and the stap (Figure 3.20) [100] which is a signal processing application used in the military domain. These applications were chosen because they target several application domains (signal and image processing) and they present different levels of complexity. A fully detailed representation of the applications tasks graph with the loop details are given in Annex A.6.

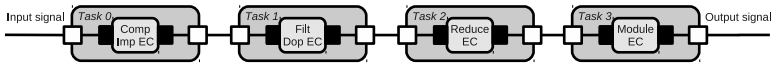


Figure 3.18 – Chirp application task graph.

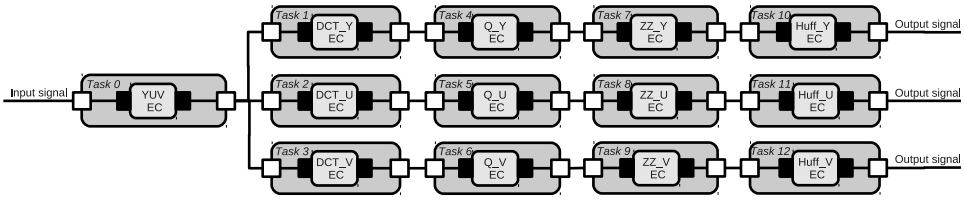


Figure 3.19 – Jpeg application task graph.

The architecture model used for the chirp application (Figure 3.21.A) is constructed around a processing node (Node 0,0) and a passive node (Node 0,1) connected by a bus. The processing node is based on a PowerPC processor [21] along with a local memory (LMEM) and a DMA. The results obtained for the chirp application are compared with an exhaustive algorithm (ESA).

The architecture used for the jpeg application (Figure 3.21.B) is the same as the one used in [72] and is constructed around 13 processing nodes (Node 1,0 - Node 5,1) and one passive node (Node 0,1) connected through a NoC. This architecture is heterogeneous. The processing nodes are based on

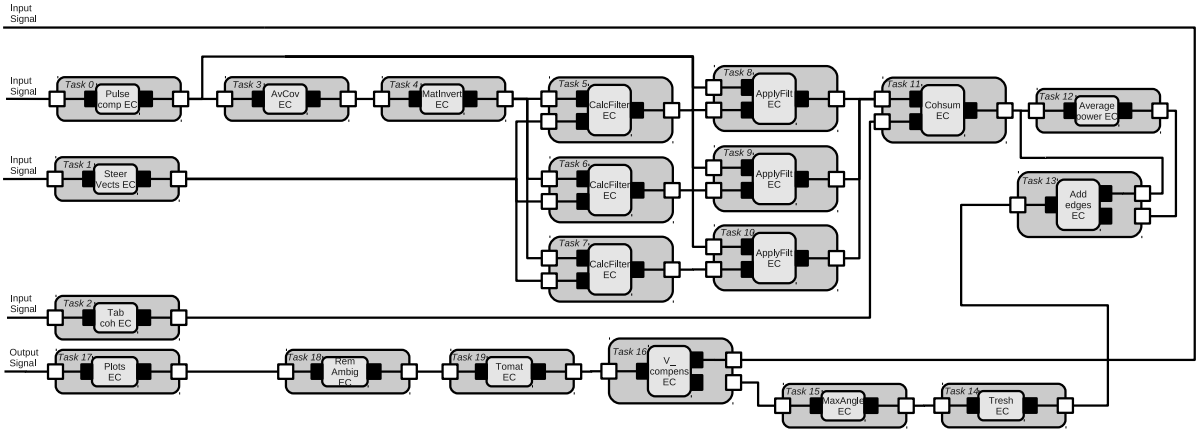


Figure 3.20 – Stap application task graph.

dedicated PE specific to each application task. The results obtained for the jpeg application are compared with an ESA and with the results in [72].

For the stap application the MPPA architecture presented in section 2.2 is used. The results obtained for this application are compared with the one obtained by the MPPA tool chain. Finally all the results obtained by the GAFO framework are compared with a classical GA (CGA) to evaluate the FO cost and benefits.

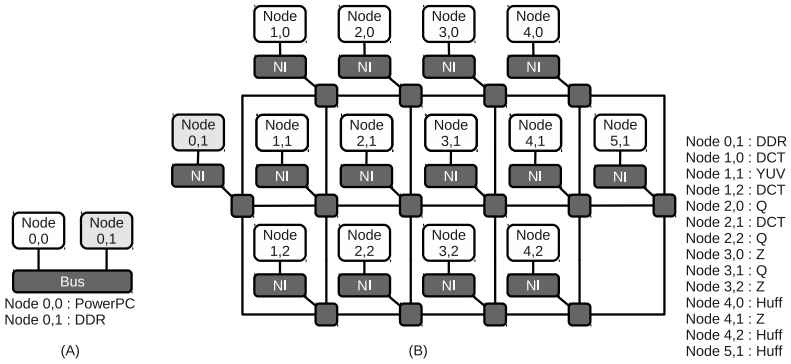


Figure 3.21 – Architecture models used for the chirp and jpeg application.

3.5.3.1 Experimentations parameters

Genetic algorithm settings:

The parameters used to configure the two GA are resumed into Table 3.5. It can be seen that the number of generations and the population size is limited to 2000 in order to keep the exploration in a reasonable amount of time. The parameter settings have been chosen from a set of trials demonstrating that the accuracy of the exploration was not improved by a further increase of parameter values as advised by [102]. This study is given in annex A.5.

Table 3.5 – Genetic algorithm parameters settings

	set values
Population size	2000
Generation number	2000
Number of parents selected for reproduction	1000
Number of children elected for next generation	1000
Crossover rate	0.8
Fusion operator rate	0.4

In the context of these experiments where the data placement heuristics is in a standalone mode the α_1 parameters have no influence on the final solution. This is due to the fact that the architectural cost is already known by the mapping and architecture exploration heuristics and no architecture optimization are done by the data placement heuristic. For these reasons α_1 is set to 0 and α_2 is set to 1.

3.5.4 Results

The task mapping and architecture models given in inputs of the data placement heuristics are supposed to be optimal which supposed that the task load is equally distributed between all the cores of the architecture. The experiments were performed on an Intel i5 quad core processor running at 2.5GHz with 8 GB of RAM.

3.5.4.1 Memory and communication cost

For the the chirp and jpeg applications several memory configurations were tested. A configuration is a modification of the memory size. For the stap application only one configuration was tested since the MPPA256 is a COTS architecture.

For the chirp application (Table 3.6) the solutions provided by the GAFO are always optimal (i.e equal to the ESA results) in terms of memory usage and data transfers. Despite the application simplicity, these results demonstrate our framework effectiveness. Indeed the framework efficiently converges to the most promising regions of the solution space by exploring just a few numbers of potential solutions. For the 256 kB case the proposed solutions are nearly ideal since the communication time is almost equal to the computation time as depicted Table 3.6. Our framework efficiently exploits the increase of the memory size by defining data placement where more data are located close to processors thus reducing the communication distance and the data transfers.

Regarding the results obtained for the jpeg application (Table 3.6), we see that the solution obtained by the framework is always the same as the one found by the exhaustive algorithm. However for the 0.125 kB case the obtained results are not the optimal one. Indeed it is possible to reach manually a performance of 3.1s. In this case the data are written in a specific order into the memory. The data organization into the memory is then quite specific and the necessary data movements to reach this organization are complex and cannot be handled by an automatic exploration. Compared to the solution proposed by Corvino et al [72], our solution processes 4 frames less per second. But their solution was

Table 3.6 – Application execution time for different memory configuration, along with the percentage of the application execution time used for the computation and for the communication.

		chirp				jpeg				stap
	Memory Size kB	0.25	1	32	256	0.125	10	300	720	32000
ESA	Runtime (s)	0,15	0,12	0,11	0,10	3.5	2.3	1.9	1.6	
	Comp. time %	29	41	46	48	24	41	44	49	
	Com. time %	71	59	54	52	76	59	56	51	
GAFO	Runtime (s)	0.15	0.12	0.11	0.10	3.5	2.3	1.9	1.6	2.7
	Comp. time %	29	41	46	48	24	41	44	49	42
	Com. time %	71	59	54	52	76	59	56	51	58

based on a point to point connection between communicating IPs which then remove the interconnect impact but limits their architecture to a single application. Moreover the memory size of their solution is about 1.8 MB while the solution proposed by our framework reduces by 2.5 this number (720 kB).

Finally for the stap application (Table 3.6) the obtained result is close to the optimal solution obtained by hand-using the MPPA tool-chain (2.556 second) [75]. The solution proposed by the framework has a 4% overhead in terms of execution time. This communication cost increase is due to the fact that some task are not regrouped into the good iteration space, which then causes misplaced data, inefficient use of the memory space and communication onto the network. The data placements which give the best results for each application are given in annex A.7.

3.5.4.2 Fusion operator cost

Table 3.7 – Performance benefit bring by the FO.

		chirp				jpeg				stap
	Memory Size kB	0.25	1	32	256	0.125	10	300	720	32000
CGA	Runtime (s)	0.15	0.12	0.11	0.10	3.5	2.4	2.2	1.9	3.2
	Comp. time %	29	41	46	48	24	39	42	44.5	34
	Com. time %	71	59	54	52	76	61	58	55.5	66
FO benefit %		0	0	0	0	0	4.4	10.1	20.6	18.4

In order to measure the benefit brouht on the solution quality by the FO, a CGA have been implemented. The results obtained with the CGA are shown Table 3.7. For the chirp application the FO is useless due to the low level of complexity of the application. However when the application complexity increase the benefit bring by the FO is more important. Indeed expect for the jpeg with the 0.125 kB case which is a specific use case the improvement ranges from 4.4% to 20.6%. These results shows that the movements performed by the FO and the exploration of the neighborhood of each chromosome allow to more efficiently use the available memory space by reducing the data transfer and by maintaining the data locality.

However if the application complexity is low as for the chirp application the use of the fusion operator is not needed to converge to the most promising regions of the solution space.

3.5.4.3 Exploration runtime

Regarding the execution time of the exploration the results are resumed in Table 3.8. We can see that the difference in terms of computing time between the two GA evolves linearly with the application complexity. This is due to the fact that the fusion operator needs SystemC simulation to evaluate the neighborhood of the current solution and move towards a better one.

Since most of the time is spent for the validation of the data placement and the generation and execution of the SystemC simulator the execution time evolve linearly. However the gains of the GA over the exhaustive algorithm are important and prove that our framework has the ability to rapidly converge to near optimal solution in a short amount of time.

Table 3.8 – Exploration runtime of the three test applications.

	chirp		jpeg		stap	
	Runtime	Overhead	Runtime	Overhead	Runtime	Overhead
ESA	10 min	934%	68 min	2472%		
GAFO	65s	12%	197s	16%	437s	19%
CGA	58s	0	165s	0	353s	0

Furthermore our approach can handle high complexity applications in a short evaluation time (less than 8 minutes for the stap application).

3.6 TSRS-GAFO

The results given in this section show the solutions obtained when the data placement and mapping heuristics are used jointly. The combination of the two heuristics is called TSRS-GAFO in the rest of this document. To demonstrate our approach our experiments are based on the chirp, the jpeg and the stap applications which have been presented into section 3.5.4.

For the chirp application the results are compared with an exhaustive algorithm and with a combination of a CTS for mapping exploration and a CGA for data placement exploration. This combination is called CTS-CGA in the rest of this document. The results obtained for the jpeg application are compared with the results of [72] and with the results obtained by the CTS-CGA. The stap results are compared with the one of the MPPA tool chain [75] and the one of the CTS-CGA. For the jpeg and stap applications the number of potential solutions for the mapping and the data placement prohibit the use of an exhaustive algorithm.

3.6.1 Experimentation

3.6.1.1 Architecture models

Several architecture models were used to see the evolution of the proposed solution with the number of nodes. These models are all organized around a mesh networks and composed of different number of nodes with different characteristics.

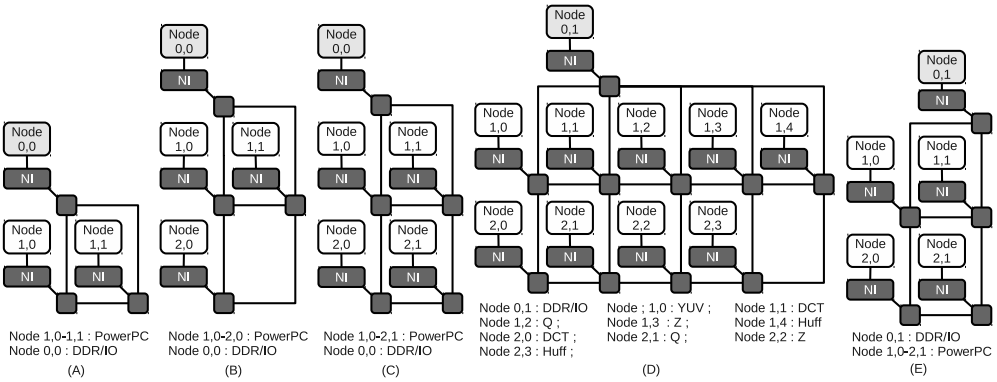


Figure 3.22 – Architecture models used for the TSRS-GAFO experimentation.

For the chirp application the architecture models (Figure 3.22.A, B and C) are based on a processing node composed of a PowerPC [21] along with a LMEM and a DMA. The number of active nodes varies from 2 to 4. All the architectures have access to a DDR memory and I/O peripherals (Node 0,0). The result of the profiling of the chirp application onto the PowerPC is given Table 3.9.

For the jpeg application two architecture models are based on heterogeneous data-flow cores (Figures 3.21.B, 3.22.D). The architecture model Figure 3.21.B is the same as the one used into [71] to allow a fair comparison of the obtained results. While the architecture model described (Figure 3.22.E) is a fully homogeneous architecture model, where all the cores are based on a PowerPC and are able to run all the

Table 3.9 – Profiling of the chirp application onto the PowerPC. Task numbering refer to figure 3.18.

Task Number	0	1	2	3
Computation time	49.8 ms	6.3ms	0.2ms	0.6ms

application tasks. All the architecture models have access to a DDR memory and I/O peripherals. The profiling of the jpeg application on the PowerPC core is given Table 3.10.

Table 3.10 – Profiling of the jpeg application onto the PowerPC. Task numbering refer to figure 3.19.

Task Number	0	1	2	3	4	5	6
Computation time	0.01 s	0.92 s	0.92 s	0.92 s	0.01 s	0.01 s	0.01 s
Task Number	7	8	9	10	11	12	
Computation time	0.01 s	0.01 s	0.01 s	0.01 s	0.01 s	0.01 s	

For the stap application the MPPA architecture presented in section 2.2 is used. The profiling of the stap application on the MPPA architecture is given Table 3.11. These results come from the MPPA tool-chain and give the number of processing cores needed to run each task in one second.

Table 3.11 – Profiling of the stap application onto the MPPA. Task numbering refer to figure 3.20.

Task Number	0	1	2	3	4	5	6
Number of cores	0.99	0.79	0.22	2.67	8.82	13.93	13.93
Task Number	7	8	9	10	11	12	13
Number of cores	13.93	15.13	15.13	15.13	4.22	0.05	0.11
Task Number	14	15	16	17	18	19	
Number of cores	1.185	1.185	0.13	0.66	0.01	0.01	

The summary of the architecture used for these tests are given Table 3.12. The architectures models are supposed to be optimal which means that they are optimized in regard of the application needs.

Table 3.12 – Summary of the architectures used for these tests.

	Number of processing nodes	Cluster based	Test application	Comments
archi_1	2	No	chirp	Figure 3.22.A
archi_2	3	No	chirp	Figure 3.22.B
archi_3	4	No	chirp	Figure 3.22.C
archi_4	9	No	jpeg	Figure 3.22.D
archi_5	13	No	jpeg	Figure 3.21.B
archi_6	4	No	jpeg	Figure 3.22.E
archi_7	256	Yes	stap	Figure 2.9

Execution time constraints:

Since in the context of these experiments the mapping heuristics has the possibility to map several tasks onto the same node, it is then able to adapt its mapping depending on the execution time constraints

set by the user to have an efficient use of the hardware resources.

To that end the table 3.13 resumes the execution time constraints set for each application and architecture model. The execution time constraints were chosen to see the ability of the proposed framework to define near ideal solution since the execution time is almost equal to the communication time based on the profiling information.

Table 3.13 – Execution time constraints set for all the application and architecture models.

	chirp			jpeg			stap
Archi	1	2	3	4	5	6	7
Execution time constraints (et_{cst}) (s)	0.080	0.080	0.080	3.5	1.7	2.5	3

3.6.1.2 Algorithm Parameters settings

The chosen parameters for the TSRS and the CTS frameworks are resumed into Table 3.14. The parameters used for the GAFO and the CGA are the same as the one given Table 3.5.

Table 3.14 – Parameters settings for the TSRS and CTS.

		Parameters settings				
	Application	Memory size	Nb of it	Diversification.	α_1	α_2
TSRS	chirp	2	2	2	0.2	0.8
	jpeg	10	20	8	0.2	0.8
	stap	8	10	6	0.2	0.8
CTS	chirp	2	3	2	0.2	0.8
	jpeg	20	40	15	0.2	0.8
	stap	6	10	6	0.2	0.8

3.6.2 Results

3.6.2.1 Memory and communication cost

As can be seen on Table 3.15 chirp application, the solutions provided by the TSRS-GAFO and the CTS-CGA are always optimal in terms of task mapping, memory usage and data transfers since they are the same as the ESA results. The framework efficiently converges to the most promising regions of the solution space by exploring just a few numbers of potential solutions. Moreover the proposed solutions are ideal since the computation time is greater than the communication time as depicted Table 3.15. Our framework then efficiently exploits the memory space and architecture capabilities by increasing the data locality. Indeed the framework proposes data placement where the data locality is increased thus reducing the communication distance and the data transfers. Moreover the two TS algorithms efficiently adapt their mapping based on the data placement and efficiently exploit the architecture resources since all the solutions are based on two cores. Indeed based on the user timing constraints and on the profiling informations it can be seen that it is not beneficial to use more than two cores.

Table 3.15 – Application execution time for different architecture model, along with the percentage of the application execution time used for the computation and for the communication.

		chirp				jpeg				stap
	Mem size (Kb)	256				720				32000
	Archi	1	2	3		4	5	6		7
ESA	Exec. time (s)	0.075	0.075	0.075						
	Comp. time %	54	54	54						
	Com. time %	46	46	46						
	Nb core used	2	2	2						
TSRS-GAFO	Exec. time (s)	0.075	0.075	0.075		3.9	1.6	2.01		2.754
	Comp. time %	54	54	54		48	49	48		40
	Com. time %	46	46	46		52	51	52		60
	Nb core used	2	2	2		9	13	4		144
CTS-GA	Exec. time (s)	0.075	0.075	0.075		4.04	1.93	2.14		2.81
	Comp. time %	54	54	54		54	44.5	45		38.7
	Com. time %	46	46	46		46	55.5	55		61.3
	Nb core used	2	2	2		9	13	4		192

For the jpeg application, the solutions provided by the TSRS-GAFO always show an improvement that ranges from 3.5% to 20.6% against the CTS-CGA (Table 3.15). This improvement is due to the fact that the TSRS-GAFO proposes solution with a more efficient use of the memory space and of the memory resources. Moreover with the mappings proposed by the TSRS-GAFO the task load is equally distributed on all the core of the architecture and the communication cost reduced. Compared to the results of [72], the proposed solutions for the archi_4, archi_5, archi_6 process 44, 4 and 17 frames less per second respectively. However as stated in section 3.5.4 their solution is based on a point to point communication model and our framework reduces by 2.5 the memory size. Finally the architectural cost for the archi_4 is reduced compared to their solution (9 cores instead of 13), while the arch_6 can be reused for any other kind of processing.

Finally for the stap application (Table 3.15) the obtained solution is close to the solution obtained by hand using the MPPA tool-chain (2.556 second). Indeed the solution proposed by the TSRS-GAFO has an 8% overhead in terms of execution time, while the CTS-CGA has a 10% overhead. As depicted in Table 3.15, this overhead is due to an increase of the communication and of the data transfers which then slows down the application execution time. This communication cost increase being due to the fact that some tasks are not mapped to the good core or regrouped into the good iteration space, which then produces misplaced data, inefficient use of the memory space and communication onto the network. The data placements and task mappings which give the best results for each application are given in annex A.8.

3.6.2.2 Exploration runtime

The results characterizing the exploration runtime are summarized in Table 3.16. This table presents the number of solutions explored accurately (Sol explo) with the call to a SystemC simulator along with the exploration runtime overhead occurred by each framework. This overhead is normalized by the

TSRS-GAFO execution time

From this table we can see that the difference in terms of runtime between the TSRS-GAFO and CTS-CGA is dependent of the application and architecture complexity. However the gain over the ESA and the CTS-CGA for the TSRS-GAFO are important and prove that our framework has the ability to rapidly converge to near optimal solution in a short amount of time.

Moreover as can been seen the metrics have importantly reduce the number of design points explored accurately.

Table 3.16 – Exploration run-time.

chirp	Archi_1			Archi_2			Archi_3		
	Sol explo	Runtime	Overhead	Sol explo	Runtime	Overhead	Sol explo	Runtime	Overhead
ESA	16	1040 s	300 %	81	3782 s	197 %	256	12052 s	508%
TSRS-GAFO	4	260 s	0 %	19	1273 s	0 %	30	1980 s	0 %
CTS-CGA	14	910 s	250 %	24	1558s	22 %	38	2508s	26 %
jpeg	Archi_4			Archi_5			Archi_6		
	Sol explo	Runtime	Overhead	Sol explo	Runtime	Overhead	Sol explo	Runtime	Overhead
ESA									
TSRS-GAFO	5	985 s	0 %	6	1182 s	0 %	600	118200 s	0 %
CTS-CGA	87	17139 s	1640 %	197	38809 s	3189 %	1002	197394 s	167 %
stap	Archi_7								
	Sol explo			Runtime			Overhead		
ESA									
TSRS-GAFO	105			42800s			0 %		
CTS-CGA	2042			816800s			1808 %		

3.6.2.3 Influence of the α parameters

In order to measure the influence of the α parameters described into section 3.4.2.6, we propose to vary the α parameters from 0 to 1 by step of 0.1, α_2 being equal to $1 - \alpha_1$. The test case applications and architectures are: (1) the chirp to map onto the archi_3 with a memory size of 256 Kb, (2) the jpeg to map onto the archi_5 with a memory size of 720 Kb and (3) the stap to map onto the MPPA with a memory size of 32 MB. The application execution time constraints are set to 80 ms, 1.7 s and 3 s respectively.

The results obtained from these experiments for the TSRS-GAFO and the CTS-CGA are shown Figure 3.23. As it can be seen for the chirp application the α parameters have no impact on the final solution for the TSRS GAFO due to the metrics constraints and the application simplicity. For the CTS-CGA and due to the application complexity the α_1 parameters only influence the exploration when they are close or equal to 1. Otherwise the α_2 parameter is dominant due to the execution time constraint which does not allow to reduction of the number in use.

For the jpeg application the TSRS-GAFO always proposes solution organized around 13 cores due to

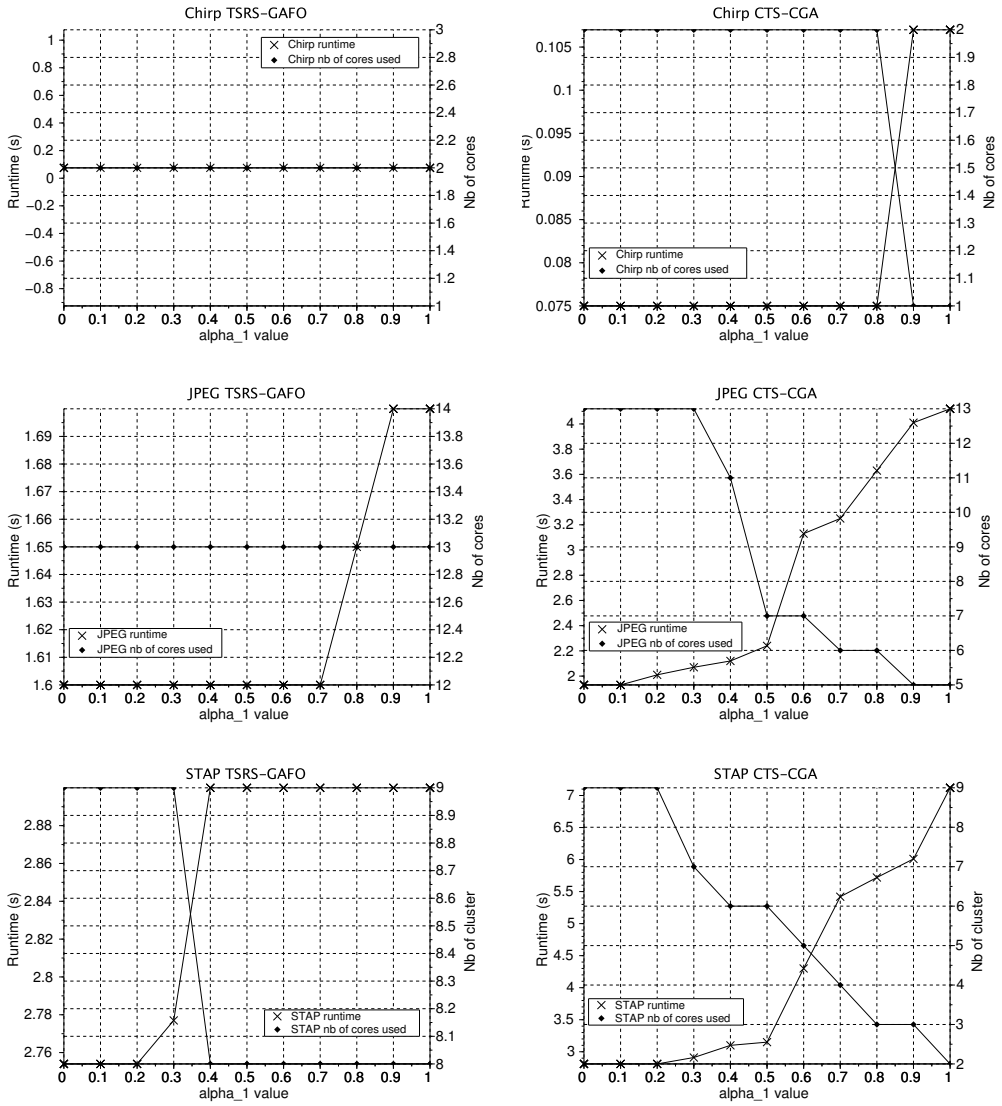


Figure 3.23 – Evolution of the proposed solution for the chirp, jpeg and stap application with the α parameters.

the metrics constraints. Since the number of cores is constant the graph shows that it is more beneficial to guides the search for high performance solution. Otherwise as shown Figure 3.23 the TSRS-GAFO leads to solution which propose an inefficient use of the hardware resources. For the CTS-CGA the α parameters allow to guide the exploration to more hardware economic or high performance solution. Moreover depending of the value set for the α parameters we see that the CTS-CGA define solution offering a compromise between hardware economic solutions and high performance solutions.

Finally for the stap application the same trend are observed for the TSRS-GAFO and the CTS-CGA.

These results show that the impact of the α parameters are limited by the metrics for the TSRS-GAFO. In order to give more freedom to the TSRS-GAFO the definition of the execution time constraints is then a key point.

3.7 Summary

In this chapter the proposed DSE methodology and PARSE tool have been presented. The needs of each methodology components have been detailed. A description of each heuristic algorithms used within PARSE is given, along with the internal representation of each design point and the cost function used by the heuristics to go through the solution space.

This description shows that the proposed methodology is able to explore at the same time the architecture, the mapping and the data placement solution space which allows to identify the best compromise for the application and the architecture. Moreover this avoids a separation of concern found into the state of the art between the application and the architecture.

Furthermore the TSRS-GAFO framework shows its ability to handle the heterogeneity of a MPSoC architecture and to map several tasks onto the same node.

The results obtained by the mapping and data placement heuristics have been given for a set of benchmarks and applications. These results are close or equal to the optimal one obtained by using an exhaustive approach.

The impact of the fusion operator and of the metrics onto the quality of the final solution and onto the exploration runtime was also measured. These set of experiments have proved the ability of PARSE and of the proposed DSE methodology to define in a reasonable amount of time an architecture, an application mapping and a data placement from an application and an architecture library.

Moreover we offer a more accurate representation of the solution space thanks to the use of a SystemC simulator. This SystemC representation is needed in order to be able to scale in the future with the technology improvement.

4

Globally homogeneous architecture but locally heterogeneous cores

Abstract: Most of the MPSoC are now becoming heterogeneous because they present better performances and power efficiency [133]. This heterogeneity brings the possibility to have for each domain the most adapted hardware architecture but also requires the support of different programming model on the same platform. To ease the design and programming of MPSoC architectures this chapter introduce two hardware modules. The first hardware module is the accelerator interface. The role of the accelerator interface is to provide a common interface for all the processing IPs connected onto the NoC. The second hardware module is a hardware memory management unit used to hide the underlying memory hierarchy for cluster based multi-processors system on chip architectures. Within the architecture each cluster is composed of processing cores, along with a memory. To maintain the consistence of the memory a hardware memory management unit is added in the cluster to increase the performance and control the memory access.

Contents

4.1 Introduction

The increasing amount of complex applications has forced the designers to define architectures with high performance constraints. The embedded paradigm introduces new constraints and these types of architectures have to provide high performance throughput, without using a lot of hardware resources and within a very limited power budget.

To face these hard constraints MPSoC architectures have appeared has a major solution. Nevertheless, industry is reluctant to take the plunge into the MPSoC world. Among the reasons, the impossibility of reusing most legacy code (which is mainly sequential C code), coupled with the risk of using an unsustainable solution and the increase in development cost are strong factors that make industry being conservative. Additionally, there is also a problem of accessibility for small product volumes where it is not profitable to design custom heterogeneous MPSoC architectures.

In order to fulfill the requirements of future applications it is then necessary to ease the access to future MPSoC architectures. To that end this chapter presents the FP7 FlexTiles project [9]. The goal of this project is to define and develop an energy efficient and programmable heterogeneous MPSoC architecture with self adaptive capabilities.

Our contribution to this project is the definition of two hardware modules constructed with the aims to ease the programming and the management of the architecture. The first hardware module is the accelerator interface which is used to abstract and ease the integration of heterogeneous IP within MPSoC architectures. The second module is a hardware memory management unit used within cluster based MPSoC architectures to abstract the underlying memory hierarchy, optimizes the application execution time and limits the data transfer among the platform.

The rest of this chapter depicts in more details the FlexTiles project in section 4.2. The accelerator interface and the hardware memory management unit are respectively detailed in section 4.3 and 4.4 along with the results.

4.2 FlexTiles

The FlexTiles project [9] aims at defining a heterogeneous MPSoC platform along with a complete tool flow. The FlexTiles architecture (Figure 4.1) is composed of a set of resources including both programmable GPP and DSP cores and reconfigurable dedicated hardware accelerators. The hardware accelerators are mapped on a FPGA fabric to provide a high-level of flexibility. The FPGA fabric is on a dedicated layer in a 3D-stacked chip with the MPSoC layer [112]. All the resources are connected through a NoC.

In the proposed execution model, GPP nodes are used as masters that can benefit from accelerators to delegate processing tasks. The accelerators, working as slaves, are either DSP cores or hardware accelerators mapped on the FPGA fabric. To homogenize the control of accelerators by the masters, a common accelerator interface ”AI” is proposed. All the resources are connected to the NoC by a NI. The NoC itself is decomposed into different channels offering different services that match the quality of service requirements.

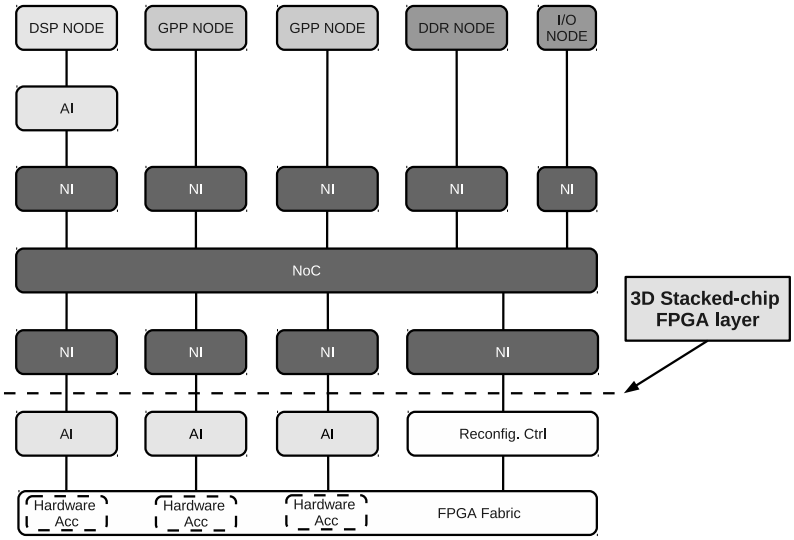


Figure 4.1 – FlexTiles architecture. The architecture is composed of two layer the MPSoC layer and the FPGA layer. GPP and DSP are mapped onto the MPSoC layer while the hardware accelerator are mapped onto the FPGA layer. All the resources are connected through a NoC, the accelerators (DSP and hardware accelerators) connection onto the NoC being abstracted by the means of the AI.

The MPSoC layer shares all the resources of the FPGA fabric. Even if there are distinct individual AIs to access to the FPGA fabric, it hence offers a high-degree of flexibility to map the hardware accelerators. The dynamic reconfiguration of the FPGA fabric is enabled via a dedicated controller also visible from the NoC. Finally, additional peripheral resources are connected to the platform acting as I/O. In particular, external memory controllers can be added (such as DDR).

4.2.1 FlexTiles Platform

The whole solution developed in the FlexTiles project targets high-performance and dynamic algorithms in embedded products. The programmer's view is a set of concurrent threads. Different priorities express the parallelism that allows dynamic resource allocation at run-time. In addition, each thread can address domain-specific accelerators to meet real-time or high performance constraints of the application.

The description of the application is captured thanks to a tool chain that helps programmers implementing and deploying their applications on the targeted architecture. With this tool chain, designers are able to express the parallelism by describing application threads as a series of dataflow graphs and combining them with priority orders and synchronization mechanisms.

The main application is programmed in C code. The accelerated parts of the applications are written in the language of the targeted accelerator: for DSP cores they are programmed in C code (possibly using assembly-level optimized libraries), for the FPGA fabric they are written in HDL code or in C code converted to HDL through HLS tools. An operating library embeds a system monitoring to be able to allocate the resources and load balance the works over the processors according to several sensors inside the chip.

4.2.2 MPSoC layer

The MPSoC layer includes all the programmable components of the FlexTiles platform. The goal of the FlexTiles platform is to provide a scalable architecture offering a large flexibility of programming and high-computational performance. The concept is to mix on the same platform general purpose processors (for flexibility and homogeneity of programming) and accelerators (for efficiency of data processing).

The nodes are the building blocks of the FlexTiles platform. Three types of actives nodes are considered: (1) The GPP nodes which are constructed around a GPP, a local memory, an instruction cache memory and peripherals for synchronization with the rest of the platform. (2) The DSP nodes which are constructed around a DSP along with an AI. (3) The hardware accelerators which are constructed around an AI and a hardware IP mapped on the FPGA fabric.

In addition the FlexTiles platform is composed of two passive nodes: (1) The I/O node used to make the connection with the outside environment. (2) The DDR node used to store the data shared among the platform.

From the functional point of view, four types of node behaviors are proposed: (1) Supervisor, a single node of the NoC that controls the master nodes attached to the same NoC. It manages the boot, reset, health monitoring, built-in self-tests and debug tasks. (2) Masters execute the user application. They may control slave nodes to manage I/Os or deported processing. These nodes are typically a GPP. (3) Slaves are nodes that receive deported work to do and communication. These nodes are typically dedicated accelerators that can process specific functions or I/O facility. (4) Passive, nodes that exposes their address-space to the other nodes. These nodes are not capable of initiating anything on the network; the other nodes can read or write into it. These nodes are typically a shared memory plugged onto the NoC.

4.2.3 Reconfigurable layer

The reconfigurable layer of the platform contains all the resources for the integration of the FPGA fabric in the FlexTiles architecture. It contains the FPGA fabric where hardware IPs (accelerators) can be mapped, the reconfiguration manager (including its configuration memory) and the resources for interfacing with the MPSoC layer (NI and AI). It is defined in opposition with the MPSoC layers that include all the programmable components (GPP and DSP).

The FPGA fabric has several common features with a “classical” FPGA architecture. It is a heterogeneous fabric of computing, memory and interconnection resources. Computing resources are logic blocks including mainly look-up tables and flip-flops, memory resources are static memory embedded for data storage during computations, and interconnection resources (wires, switch boxes) aim at connecting the different computing and memory resources to each other. The FPGA fabric also includes some arithmetic computing resources (arithmetic configurable datapaths) to improve the overall processing performance, if necessary. Because the reconfiguration layer is stacked, this layer does not have direct access to the I/O pads of the chip, the I/O cells of a classical FPGA are replaced by the AI that give access to the MPSoC layer. Finally the FPGA layer provides capabilities for online reconfiguration and task migration. This last feature allows the FlexTiles platform to have more flexibility in regards of dynamic application workload.

4.2.4 NoC

On the FlexTiles platform, a large number of resources have to exchange information such as control, configuration and data. The proposed approach is to adopt a communication interconnect based on NoC technologies for scalability reasons [49].

The topology is based on a mesh approach since it provides the best performance-complexity compromise and the simplest implementation [26, 28, 45, 113]. Indeed, local communications are well supported and favored by this topology. Finally a mesh topology offers a better scalability than its counterparts [49].

4.2.5 Programming model

An application is a set of static clusters (Figure 4.2). A cluster is a representation of a particular function of the application and regroup is described using synchronous data flow (SDF) or cyclo static data flow (CSDF) models of computation [162].

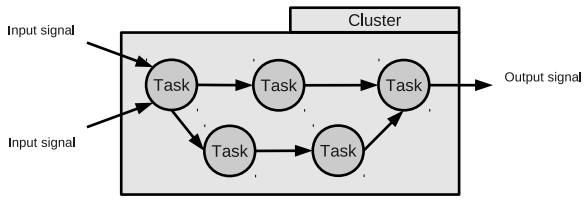


Figure 4.2 – Static cluster.

Within an application, each consumer/producer of tokens is called a task. A task is featured by nested loops implementing the operator and the rules of token consumption/production. Two tasks communicate through FIFOs of tokens.

Each cluster can be started or stopped depending on events acting globally like a discrete event (DE) representation. This dynamicity is represented by groups of clusters. In each group, several clusters having the same data flow inputs and outputs are sensible to different events or events combination. At a given time, only one cluster of the group is active.

Three possibilities are represented on Figure 4.3:

- Cluster group 1 and 2: The computing element is started when a dedicated event appears (cluster 1) else nothing is done (cluster 0).
- Cluster group 3: When there is no dynamicity, there is only one cluster.
- Cluster group 4: This computing element has 3 possible behaviors.
- Cluster group 5: This cluster is able to dynamically duplicate. Depending on how the chip is loaded and how many nodes this cluster group is allowed to use, it can duplicate itself to parallelize its processing. It is called data parallelization because the data are spread among the duplicated clusters. The designer decides the rules on how to parallelize and split the data depending on the number of instances of the cluster as well as the allowed possible numbers of instances.

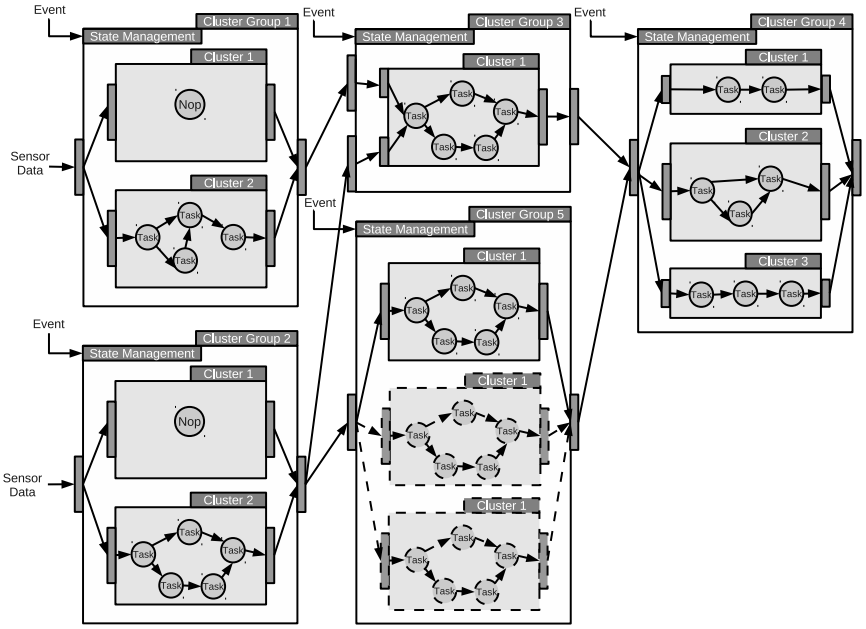


Figure 4.3 – Management in FlexTiles: Cluster groups can behave differently depending on their dynamicity level

4.2.6 Software tools

As described above, the application is described as a set of static cluster. Each cluster can be independently optimized by using the proposed tools SpearDE [139] and Cosy [88].

SpearDE allows to describe the application with the SDF models, to express the application parallelism, and to map the different tasks on the different heterogeneous node represented in the architecture model of FlexTiles. Finally, SpearDE generates the mapped code used by Cosy.

In the Cosy framework, a streaming compiler is used to convert annotated C code into CSDF programs. The CSDF programs are mapped on the MPSOC platform, i.e. tasks are allocated to processors, instructions and data are mapped to local memories, and communication channels are mapped to NoC connections, and memories. All embedded driver C code to configure processors, DMAs, NoC, and memory arbiters are generated.

For the task executed by the accelerators, each computation is generated by using the corresponding tool chain depending on the targeted (DSP or FPGA) fabric.

4.3 Accelerator interface

4.3.1 Introduction

There exist many initiatives which have provided standard interfaces in order to abstract and ease the programming of hardware accelerators. The most common approach is to abstract the communication with the accelerators by the means of FIFO. For example in [130] hardware accelerators are abstracted as UNIX process that access operating system (OS) communication services and communicate using FIFO buffers. Despite the benefit provided by these approaches, the only targeted accelerators are the data-flow ones which limit the use of these solutions.

Some works propose to extend the thread programming model to accelerators for hardware abstraction. This is the case of the hybrid threads project [33], which focuses on implementing the synchronizations primitives provided by the POSIX multithreaded programming model as dedicated hardware cores. However their communication model was based on a monolithic shared memory paradigm which is not scalable. With the same philosophy [94] introduces a configurable hardware interface for hardware tasks. The interface uses memory mapped I/O to communicate with its accelerator. However with this approach the memory hierarchy is shared by the GPPs with the slave accelerators which therefore limit the number of potential accelerators within the platform. In [207, 208] the integration of the hardware accelerators by the use of a virtual memory space is proposed. This approach supports the creation and control of both hardware and software threads through Linux and allows hardware threads to access Linux virtual memory address space. Despite the benefit provided, the hardware thread complexity is greatly increased to maintain the virtual address translation tables and invokes the memory manager running on the CPU.

In contrast with these works, [121] propose an operating system abstraction where the communication model between the software and the hardware is not defined. With this approach the support of any kind of accelerators is enabled. However with their approach the hardware interface making the abstraction and the connection of the accelerators onto the interconnect is not generic. This approach then implies to redefine the hardware interface each time the accelerator is changed.

IP-XACT [35, 173], SHIM [82] and OCP [17] are other major contributions in order to ease the integration and the abstraction of hardware IP. IP-XACT allows the description of the electronics components through compliance rules specified using the extensible markup language (XML). However, there are some semantics limitations in this approach, since IP-XACT has a limited number of semantics rules to model an architecture [83]. Unlike the IP-XACT standard, SHIM proposes to define an architecture description standard from the software perspective. Despite the benefit enabled by the software viewpoint the difficulty to conceive and build heterogeneous MPSoC architecture is still present. OCP on the other hand defines an efficient, bus-independent configurable and highly scalable interface for on-chip subsystems communications. However the bus based protocol approach does not comply with the scalability needs of MPSoC architectures.

This lack of genericity and scalability then makes the programming of heterogeneous MPSoC platform difficult. This is why in current heterogeneous MPSoC architectures, hardware accelerators are preferably used with a data-flow programming model where the accelerator execution is only determined based on the availability of input arguments. However, these platforms by limiting accelerators to a data-flow programming model reduce the attractiveness of these solutions. Indeed only one programming model is supported which limits the flexibility of the platform. Moreover by using this approach we do

not take a full advantage of the computing power provided by the accelerators since the processor has to perform a lot of control on the accelerator and is then unable to realize any other operations in parallel of the accelerator execution.

In order to fully benefit from heterogeneous architectures and ease their definition and programming the AI is introduced. The AI provides a flexible abstraction interface which allows to ignore the addressed accelerators. Moreover in order to fully exploit the computing power of the MPSoC platform the AI is autonomous once programmed. This property enables to offload the processor from fine-grained control of the accelerators. The accelerators types abstracted by the AI are the ones of the classification proposed in [203] and introduced in chapter 3.

4.3.2 Architecture

The AI architecture (Figure 4.4) is constructed around one arbitrator and four channels: (1) the control/status channel, (2) the programming channel, (3-4) the data input and output channel.

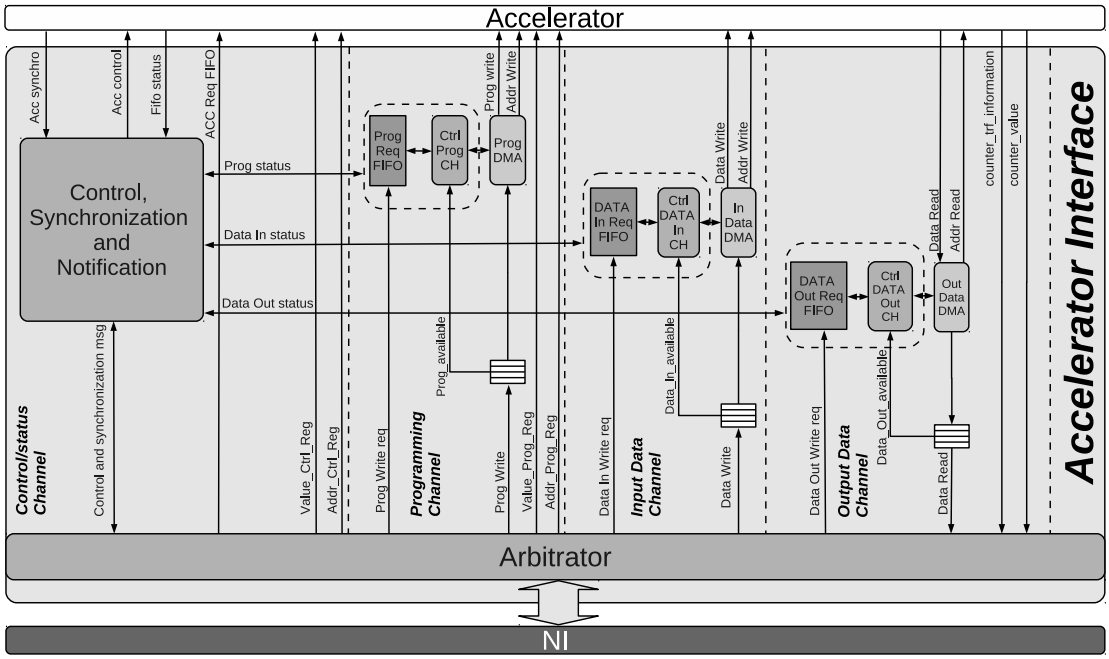


Figure 4.4 – Accelerator Interface global view.

4.3.2.1 Arbitrator

The arbitrator is used to reduce the number of input and output ports going in and out of the AI, in order to reduce the NI complexity and simplify its architecture.

Within the arbitrator, two modules are implemented one to manage the inputs and one to manage the

outputs. On the input side the component drives the incoming signals and data to the correct channels based on the information provided by the NI. On the output side a round robin scheduling policies between the channels granting the access to the network is performed.

To drive the signals to the correct channel the arbitrator decode the ai_component_id_t (Listings 4.1). The ai_component_id_t identifies a component within the AI. A component can be a DMA, a FIFO or a proxy used to access specific registers of the accelerators. This structure has three fields:

- **type**: Controls the semantics of the fields.
- **channel_id**: Identifies the channel to be accessed.
- **reg_proxy**: Identifies the register to be accessed.

```
typedef struct {

  enum {
    ai_component_is_dma,
    ai_component_is_fifo,
    ai_component_is_proxy
  }type;

  uint channel_id;
  uint reg_proxy;

}ai_component_id_t;
```

Listing 4.1 – Structure used for the ai_component_id_t

4.3.2.2 Control/Status channel

This channel is used by the GPP to offload processing tasks on the accelerator. Thanks to this channel, the GPP can write control orders and receive status information. In addition it allows the accelerator to provide direct notifications to the GPP. In the AI, this channel provides an access to a Work Request FIFO (Located inside the accelerator) where a GPP writes requests to start processing in the accelerator (Only used by the domain oriented processors or ASIP).

4.3.2.3 Programming channel

This channel is used by the GPP to program the accelerator. It is used to set registers and load binary instruction code into the accelerator. In the AI, the programming channel is implemented with an interface to read and write all memories and registers that are address-mapped in the accelerator and visible from the master nodes on the platform.

A DMA is used to manage the loading of large set of configuration parameters or instructions. The DMA is associated to a request FIFO and a state machine (Ctrl Prog CH in Figure 4.4). The DMA sees a unique address space. The ”Prog request FIFO” is used to store a sequence of load operations. The state machine control and synchronize the channel with the rest of the platform by interpreting the sequence

of load operations. Note that the FPGA reconfiguration uses dedicated resources whose access is not managed by the AI.

4.3.2.4 Data Input and Output channel

The input and output channels are constructed based on the same architecture. The input/output data processed by the accelerator are sent through these channels. These channels implement a DMA to manage the transfers of data which as for the programming channel sees a unique address space.

The DMA is associated with a request FIFO and a state machine (Controller Data IN/OUT channel in Figure 4.4). The "Data In/Out Req FIFO" can store a sequence of load operations. The state machine controls and synchronizes the channel with the rest of the platform by interpreting the sequence of load operations.

4.3.3 Command interface

As depicted on Figure 4.4 hardware mechanisms are introduced to reduce the GPP control overhead by distributing simple control commands over the AI.

The control distribution is performed by request FIFO used both for processing and communication purpose. The synchronizations are performed thanks to specific commands pushed by the master nodes into the request FIFO's that allow the AI to be synchronized with the other components of the platform. In order to give an autonomous behavior to the AI and increase the platform computing power the GPP can also anticipate and pre-load a sequence of commands.

Moreover to increase the autonomy of the AI regarding sequencing control commands, the channel controller is also able to support extended execution patterns such as loop that allows repeating a sequence of commands. The list of these specific commands is given in Table 4.1.

Table 4.1 – List of specific commands used to program the AI.

Name	Function
exec	Address of the task to be executed into the instruction memory of the accelerator.
sd_data	Send data to another accelerator or to a GPP.
sd_data_ext	Ask for a write access into the DDR memory.
wait_evt	Wait for a synchronization event from another channel or from another component of the platform.
sd_evt	Send a synchronization event to another channel or to another component of the platform.
rcv_data	Receive data from the NI and send it to the accelerator.
rcv_data_ext	Ask for a read access into the DDR memory or onto an I/O
loop	Allow to execute a loop on the requests contained into the FIFO.

4.3.4 Synchronization scheme and execution patterns

To describe and explain the synchronization scheme and execution patterns of the AI, the application tasks graph and architectural model given Figure 4.5 are proposed.

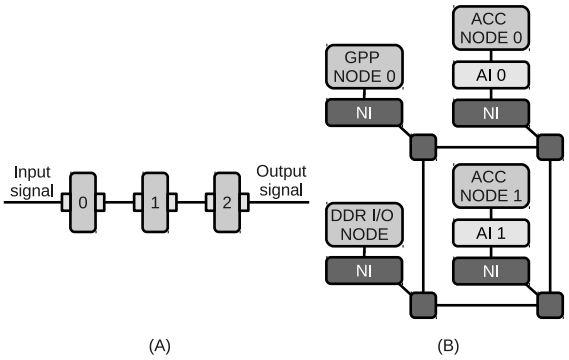


Figure 4.5 – (A) Application model, (B) Architecture model of the proof of concept example.

In this example the task 0 is run on a data-flow accelerator mapped onto the accelerator node 0, while the task 1 is run on a micro-programmed accelerator mapped on the accelerator node 1. Finally the GPP node 0 concludes the application execution and runs the task 2. Based on these models and mapping, the synchronization scheme and execution patterns are described Figure 4.6.

The first step needed to run the application is to program the AI of the platform. To that end the GPP node 0 is used to program the AI 0 and 1.

Once the AI programmed, the data coming from the I/O are fetched by the AI 0. Since the accelerator 0 is a data-flow one, the task 0 is fired based on the input data. As soon as the first results become available the AI 0 automatically transfers the results and synchronizes based on the data with the AI 1.

Since the accelerator 1 is a micro-programmed accelerator the AI 1 sends a specific synchronization message to inform the accelerator that the data are present into its local memory. The synchronization event implies the firing of task 1.

Once done the accelerator 1 sends a synchronization event to the AI 1, to inform that the task execution is done. Based on this synchronization event the AI 1 sends the data and a specific synchronization event to the GPP node 0. The GPP 0 then stops its current processing and the task 2 is fired to finish the application execution.

As can be seen from this example, the main advantage of the proposed approach is that the access to the accelerators is homogeneous. Indeed the access to a data-flow or to a micro-programmed accelerator is exactly the same for the master GPP.

Moreover as shown on the sequence diagram the accelerator is autonomous once programmed which allow the GPP to be offload from the control task and to do other processing in parallel.

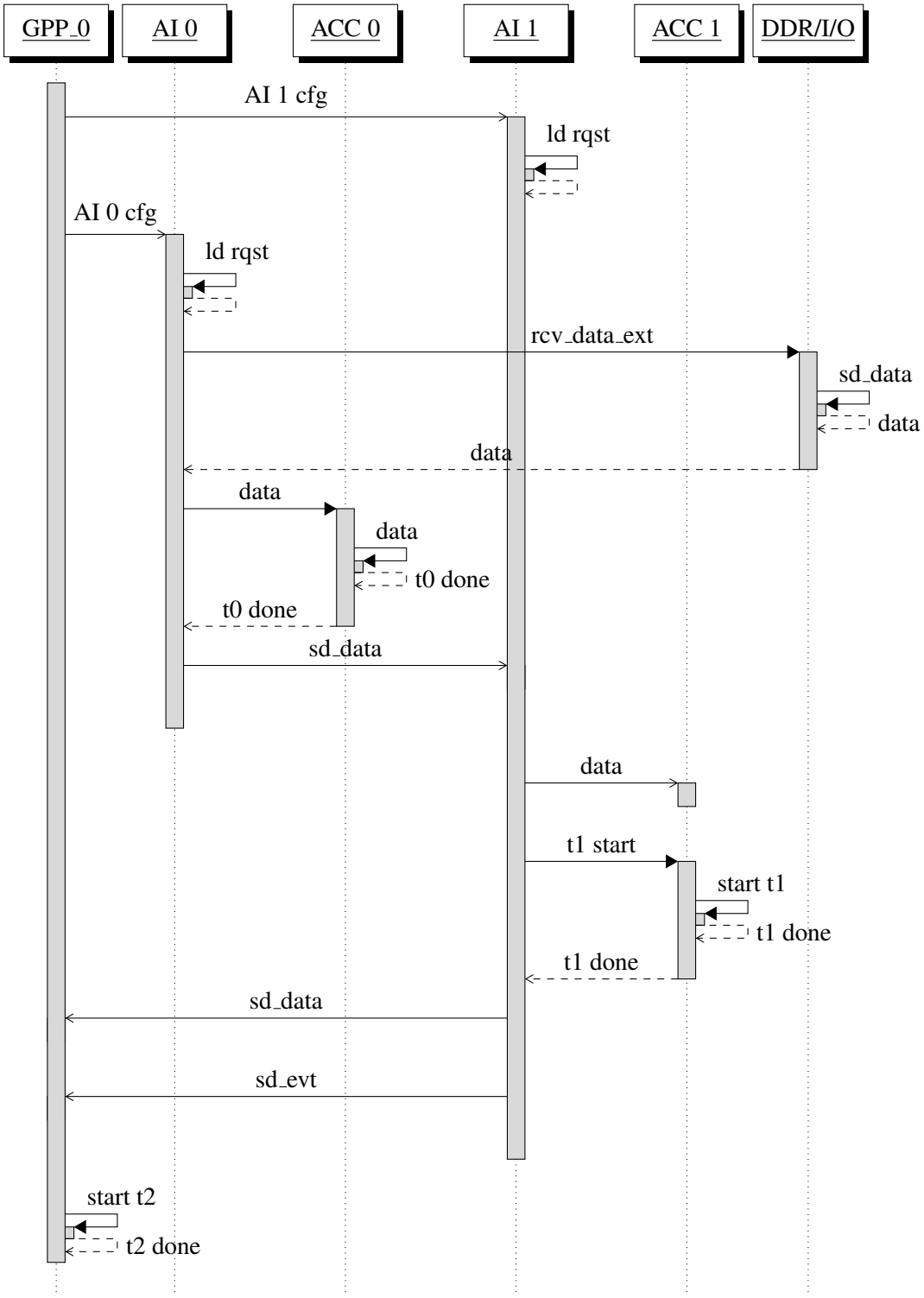


Figure 4.6 – Synchronization scheme and execution patterns.

4.3.5 Experimentation and results

4.3.5.1 FlexTiles Board

The FlexTiles board [8] (Build by Sundance for the project) (Figure 4.7) is used as an experimentation platform. The board is constructed around two Xilinx Virtex 6 SX475T FPGA [29] and several I/O peripherals.

In the context of the FlexTiles project one FPGA is used to implement the MPSoC layer, and the second one is used to emulate the FPGA fabric. These two FGPAs are connected by the means of an AURORA bridge.

Thanks to these features the FlexTiles board is able to host a MPSoC of approximately 100 GPP nodes on the first FPGA. The number of accelerators implemented on the second FPGA is dependent of the running application and of the needed accelerators.

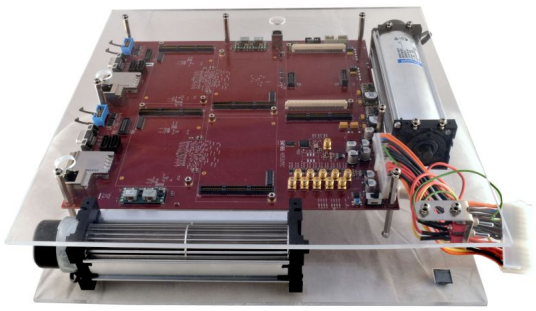


Figure 4.7 – FlexTiles board.

4.3.5.2 Application

The vehicle registration plate detection algorithm [132] was chosen as test application. The purpose of this application (Figure 4.8) is to locate and extract a registration plate from a larger image. Despite the low complexity level of the application, this one contains several steps that run in parallel and requires the use of both data-flow and micro-programmed accelerators. This application is then particularly well suited to show the AI ability to abstract any kind of accelerators.

The application takes as input a single image (420*340 pixels), 8-bit grayscale raster format. It performs the mathematical morphology operations erode and dilatation independently on two copies of this image (Tasks 0,1,2,3) combines and filters the results (Tasks 4,5,6,7) and then applies the output mask to the input image (Task 8). The output is a copy of the input image with all the registration plate extracted and the rest of the image masked out in black (Figure 4.9).

In the context of this application the execution time constraints is set to 400 ms/image.

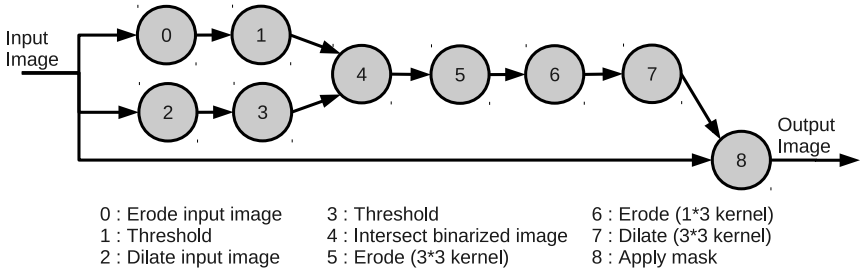


Figure 4.8 – Vehicle registration plate, application tasks graph

Figure 4.9 – Vehicle registration plate input and output images

4.3.5.3 FlexTiles platform Implementation

The implemented architecture, shown in Figure 4.10.

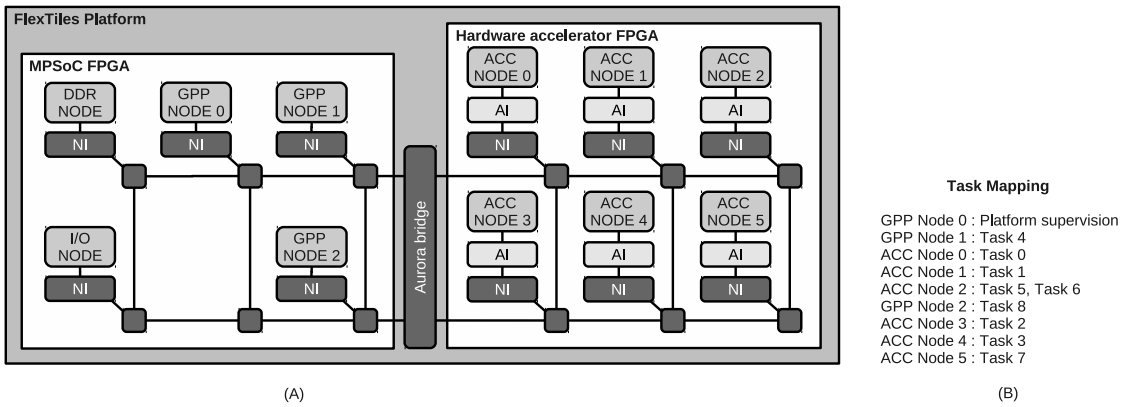


Figure 4.10 – Architecture implemented onto the FlexTiles platform.

The GPP node is build around a Xilinx Microblaze, a local memory, a DMA, peripherals for synchronization with the rest of the platform and a NI used to send and receive data transmitted over the network. The memory size is limited to 64 KB, and is used for both executable code and scratchpad storage by the processor.

All the nodes share an identical set of common components and memory layout. In addition the supervisor node runs the global OS and supervises a finite set of nodes within the NoC. Therefore, the

supervisor (GPP node 0) is in charge of the task mapping, the boot and the I/O management. To that end the supervisor receive a dedicated application to manage the overall platform.

The NoC implemented is based on HERMES [154] and provides a raw bandwidth of 3.2 GB/S for each router input and output. Each node of the architecture is connected to one router of the NoC. The mesh connection allows us to route messages through several paths across the architecture. The on-chip network is reliable, in the sense that a packet sent from one of the nodes is guaranteed to be delivered to its destination node. Any packet loss would therefore be due to a faulty component.

The shared memory node provides a high-bandwidth data connection between several end points from the NoC and a passive DDR3 memory able to store data at a maximum rate up to 4.2 GB/s.

The selected test application requires a high level of parallelism. Therefore, algorithm profiling suggests accelerating morphological operations via hardware accelerators in order to meet real time constraints. Six accelerators are implemented, four data flow oriented and two micro-programmed.

The data flow accelerators (ACC NODE 0, 2, 3, 5) are used to erode and dilate the images, while the micro-programmed accelerators (ACC NODE 1, 4) are used to threshold the images. The GPP NODE 1 and 2 are used to program the AI and execute task 4 and 8. Finally the GPP node 0 is used to supervise the platform.

The mapping of the application onto the architecture is given Figure 4.10.B . Moreover the synchronization scheme and execution patterns used to run the application onto the architecture are given in Annex B.1.1.

4.3.5.4 Impact of the accelerator interface

The results obtained for the AI hardware cost are summarized into Table 4.2. This table show the number of Flip Flop, LUT, BRAM and slice used by one AI along with its area occupancy in regards of the FPGA capabilities. The request FIFO size is limited to 32 requests per FIFO, which turns to be sufficient in the context of deeply nested loops. Based on these results we can see that the AI operates at a high frequency while occupying a small amount of hardware resources in regards of the FPGA capabilities.

Table 4.2 – Area occupancy of one Accelerator Interface

	AI	Area Occupancy
Flip Flop	3510	0.6% (3510/595200)
LUT	3046	1.0% (3046/297600)
BRAM	18	1.7% (18/1064)
Slice	724	2.6% (1999/74400)
Frequency	400 Mhz	N/A

Moreover in regards of the other component of the platform (Table 4.3) the AI area consumption is less than the size of a processor and of a router and more than 2 times the size of a NI.

In the context of the architecture used for the vehicle registration plate extraction the resources used by the 6 implemented AI represent less than 1% of the amount of the slices used by the overall architec-

ture (4400 for the 6 AI, 51705 for the overall architecture). This shows the AI reduced hardware cost in context of a MPSoC architecture.

Table 4.3 – AI area consumption (Occupied slices) as relative ratio between components

AI/Component	Consumption ratio
AI/Router	0.93
AI/Processor node	0.72
AI/NI	2.2

The latencies induced by the AI (Table 4.4) to transfer the data from the network to the accelerator are just applied onto the first data transfer since the AI presents a pipeline behavior. As the accelerators work on application with deeply nested loops the latencies imposed by the AI on the first data transfer do not impact the performance.

From a user point of view the AI hides the inherent heterogeneity of the accelerators, which helps programming the platform and reduce the time to market. Moreover the AI gives to the accelerators an autonomous behavior once programmed which increase the platform parallelism.

Table 4.4 – Latency of each elementary operation supported by the AI.

	Latency in clock cycle
Transfer of data from the NoC to the Accelerator	7
Transfer of data from the accelerator to the NoC	8
Control channel registers access	5
Programming channel registers access	6
Request transmission	6
Request Execution	5

4.3.5.5 Mapping test

The supervisor of the architecture is in charge of the tasks mapping onto the architecture. To that end the supervisor based on the information provided by the platform actuators has to decide if a cluster has to be dynamically duplicated, stopped or started.

The time taken by the initialization phase is not critical for the application as it is only done once at system start up and each time a reconfiguration of the system is needed. This implies at a low frequency in comparison with the application data flow. The application work is real-time between each task allocation. At each task allocation, a reconfiguration and initialization time is allowed.

At initialization stage, the mapping must fulfill a set of rules, such as communication latency and throughput, as well as task execution time. Thanks to application constraints (sensor input throughput, communication paths between tasks, data sizes, tasks execution time, resources needed by each task) the supervisor find a mapping of the tasks on the architecture, and a route for each data path thanks to the NoC.

The mapping realized by the supervisor gives for each part of the program a processing unit and

each logical data exchange between those parts a physical routing on the network that works for the application. This means that the routing algorithm finds physical communication paths that allow no contention or deadlocks on the network.

Because the map and route algorithm will use heuristics to find a solution, these solutions will not be equivalent but will always be valid, i.e. the application will work functionally (the application gives the expected result) and temporally (execution time, throughputs and latency are met). This point is addressed in another part of the project.

To show that different mapping will not jeopardize the application behavior we exhaustively tried all the application mappings and we have evaluated the impact of that mappings on the communication latency. This is the only impact a mapping can have on the communication, the throughput being guaranteed by the network.

Table 4.5 – Mapping experimental Results

Mapping solution	Application elapsed time (ms)	Application penalty elapsed time when compared to best mapping (ms).
Best Mapping	377.33	0
Worst Mapping	384.44	7.11

In our test the supervisor is in charge of the input and output data for the application. Therefore it can measure the time difference between the time the input image is finished to be sent to the first processing task and the time the output result is beginning to be received from the last processing task.

We only keep the worst and best mappings to measure the jitter on the map and route task. Table 4.5 shows the latency difference between the best mapping and the worst one. The difference is about 7.1ms which compared to the 377ms time needed by the application to run, is about 2% which is caused by a difference of hops count in the network.

4.4 Cluster based architecture a memory abstraction

4.4.1 Introduction

Designing the memory hierarchy is one of the greatest challenges of the designers of MPSoC architectures, even more with the emerging heterogeneous MPSoC architectures. This heterogeneity brings the possibility to support different types of application on the same architecture and so to support various computation models. To take full advantage of this heterogeneity, it is therefore necessary to support different programming models specific to the targeted application domains. The choice of a memory hierarchy is then a key issue in the development process of the platform.

In current MPSoC architecture, especially for medium and large scale systems, memory are preferably distributed, in order to offer a good scalability, fair contention and low delay of memory accesses, since the centralized memory has become the bottleneck of performance, power and cost [124]. However programming a distributed memory system is difficult, and can be not efficient for all the targeted domains [65].

In order to overcome the so called memory wall [124] innovative approaches based on shared memory have been proposed like in the STHORM and the MPPA platforms. In these works, the processing elements are grouped in cluster connected to a shared memory through the use of a cache memory. However the cache memories are not always used in the embedded systems domain. Indeed the access time to the memory is not predictable which is unacceptable in the case of real time applications.

With the same philosophy, the Ter@ops project [26] and the codelet abstract machine model associated with the *codelet* program execution model [188] propose to connect through a shared memory the processors inside of the cluster. However the management of the shared memory consistency is only done at the software level which is less efficient than a hardware implementation [90, 201].

Some approaches proposed to have a memory space physically distributed and logically shared, in order to ensure data access time and to limit the potential bottleneck at the shared memory port level as the *scalable chip multi-processor* [205]. However the size of the crossbar needed to switch the path of the PE to the memory is of significant size and limits the scalability of the solution.

Garibotti et al [95] proposes multithreading capability onto a distributed private memory NoC-based MPSoC. This proposal is realized through the implementation of a hardware module that enables virtual symmetric multiprocessing clusters definition at run-time, along with a software-based memory consistency approach. This approach shows effective results and is scalable but is not as efficient as a physically shared memory due to the communications overhead. Moreover only the owner of the shared piece of memory can write the memory which therefore limits its use. Finally the consistency of the shared piece of memory is not fully managed at the hardware level which increases the programmer task.

Intel with its Single Cloud Computer (SCC) [13, 113] proposes the only MPSoC platform implementing an on chip message passing (MP) protocol. Within the SCC all the cores are connected to their own local memory and communicate thanks to an on-chip MP protocol.

In order to reduce the programmer task and improve the performance, some academic works propose to manage the shared memory at the hardware level and have shown promising results [90, 147, 153, 167, 169]. However, most of these studies mainly focus on quantifying the impact on the network on

chip (NoC) bandwidth of such solutions. Whereas in order to validate such an approach it is necessary to quantify the impact on the application performance which is more sensitive to the size of the memory, its location and to the data access delay.

The main contributions in order to implement an on-chip MP protocol were proposed by Tota et al. [65, 201, 200]. They indeed propose a MPSoC architecture implementing the MP paradigm, where for each running application a processor is responsible to ensure the data consistency and synchronization between processors, while the memory banks are controlled by a specific processor. Their approach has shown efficient results, but they used a single monolithic memory which creates bottlenecks onto the network when the number of processors granting an access increases.

All these works have shown promising results and have tried to optimize the data access time in the context of MPSoC architectures. The main limitations of these studies are a lack of scalability and the consideration of fully homogeneous architectures. While nowadays more and more architectures propose to target domain specific accelerators to meet real-time or high performance constraints of the application and to reduce the energy consumption.

In order to take full advantages of the MPSoC architectures it is necessary to define memory management models that allow the reduction of the communication overhead and ease the programming of the overall machine.

To that end this section introduces based on the principle used in the FlexTiles project for the programming models and the software tools, a cluster based MPSoC architecture where the processors are grouped into cluster connected to a local and a shared memory. This approach allows us to reduce the communication overhead and to have the on chip memory locally shared and globally distributed.

The main difference of the proposed architecture with the state of the art is that the memory consistency and management is done at the hardware level. This approach allows the performance improvement and eases the programming of MPSoC architectures by partially hiding the underlying memory hierarchy to the programmer. This is achieved by a hardware MMU which allows the programmer to get rid of the memory management and allocation issue.

4.4.2 System description

4.4.2.1 Architecture presentation

The proposed MPSoC architecture (Fig. 4.11) is constructed around clusters connected through a NoC as done in current MPSoC architectures [26, 45, 75]. Indeed local data transfers are favored since the communications latencies are reduced inside of the cluster. Moreover the cluster approach reduces at the same time, the NoC area, the propagation delay and the energy consumption [45].

The NoC for its part ensure the communications of the various elements of the architecture. For the topology a mesh approach was chosen because it provides the best performance-complexity compromise and proposes the simplest implementation as stated into section 4.2.

Within the architecture the memory space is physically distributed into all the clusters of the architecture and locally shared within the cluster. In order to partially hide the underlying memory hierarchy and ease the programmer task a specific hardware MMU is used to define a generic communication model.

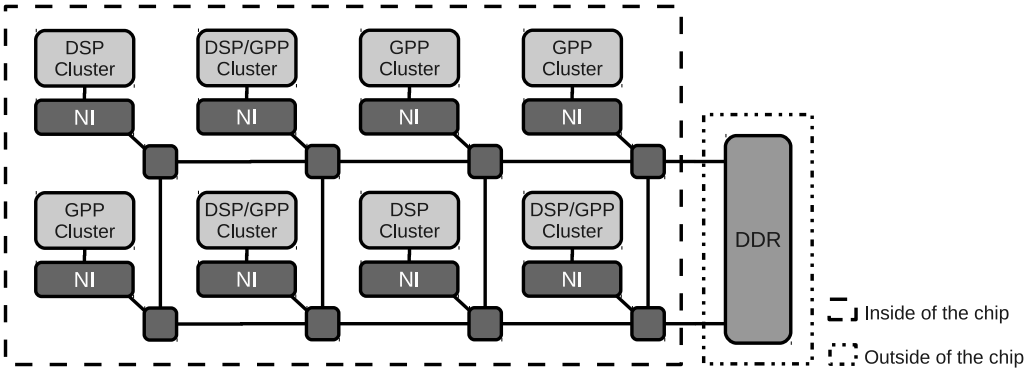


Figure 4.11 – Proposed architecture where clusters are implemented to favor local data transfer thereby increasing the overall system performance.

Indeed with the proposed hardware MMU the processor access transparently and with the same protocol all the memories of the platform. To that end the hardware MMUs present into each cluster are able to forward the messages that do not target the local MMU. This approach leads to a non uniform memory access model but this does not affect too much the flexibility of our architecture. Nevertheless for cluster to cluster communication this approach causes a communication overhead. Indeed the NoC has to arbitrate several requests, which can lead to a higher latency than a fully flat architecture model. Finally all the processors can access to an external DDR memory. In the context of the proposed approach this memory is used for data storage purpose when the memory space available on-chip is not sufficient.

4.4.2.2 Cluster Architecture

The proposed cluster (Fig. 4.12) is organized around an interconnect connecting: processor nodes which execute tasks from a process, and a memory node which store the data shared among the clusters. The processor nodes of the whole architecture can concurrently access the memory nodes present in all the different clusters of the architecture. However it is better to promote local accesses, due to the overhead (power, latency) induced by remote accesses.

The architecture of the processor node (Fig. 4.12) is heterogeneous to be able to target specific application domains. To that end the processor nodes are based on a GPP or a DSP along with a set of peripherals. The peripherals are used to connect and synchronize the node with the rest of the platform and to give an autonomous behavior to the node.

The processor node is also associated with one instruction cache memory that contains the instruction code to be executed. Finally a local memory that contains the data currently computed by the processor is implemented. This memory is only accessible by the attached processor.

Thus to take a full advantage of the memory node the data transfers between remote memory node resources have to be limited. Obviously the local data accesses are dependent of the tasks mapping and data placement.

In the rest of this section the tasks mapping and the data placement are supposed to be ideal. This hypothesis assumes that the task mapping and data placement take a full advantage of the architecture

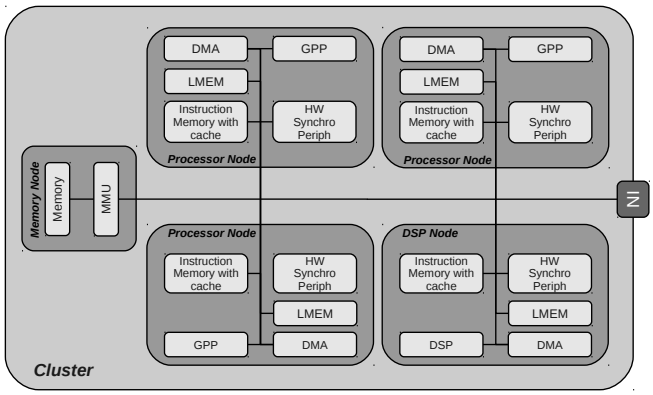


Figure 4.12 – Cluster architecture, each cluster is composed of a set of processor nodes along with a memory node. A processor node being composed of all the elements necessary to communicate and synchronize with the rest of the platform.

capabilities to maximize the processors load. However the proposed approach allow to ease the task mapping and data placement since the user does not have to take care about the memory hierarchy and data transfers because they are managed at the hardware level.

4.4.3 Memory node

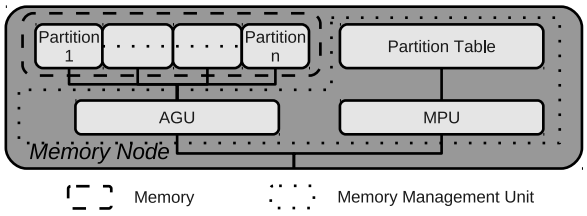


Figure 4.13 – Memory node composed of a partition table, an address generation unit (AGU) and a memory protection unit (MPU)

To be able to protect and maintain the memory consistent the MMU is constructed around three elements (Fig. 4.13): (1) a partition table used to record the information about each partition of the memory, (2) an address generation unit (AGU) used to make the virtual to physical address translation, (3) a memory protection unit (MPU) used to ensure data consistency. Finally the memory is cut into a set of blocks called partition. The memory is divided into a set partitions at design time depending on the targeted application domain.

4.4.3.1 Partition Table

The role of the partition table is to record the information related to each partition, in order to provide the MPU with enough information to maintain the memory consistency. The informations contained into

the partition table are the following (Fig. 4.14): (1) Partition ID: Represent the ID of the partition, (2) Partition Properties: Information used to tell if the partition is read only, read/write, write only, (3) Remaining read operation: Number of read operations to perform on the partition before writing new data, (4) Write: Indicates that a write operation is in progress and (5) Free: Indicates that the partition is not allocated to a processor and can be re-allocated.

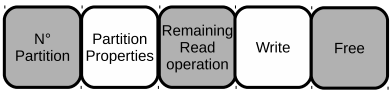


Figure 4.14 – Structure of a line into the partition table.

4.4.3.2 Address Generation Uuit

The AGU (Figure 4.15) is here to make the virtual to physical address translation. Its architecture is composed of a control unit which checks if the access targets the local or a remote memory node and a adder which adds to the virtual address the required offset to enable the access to the value in the memory.

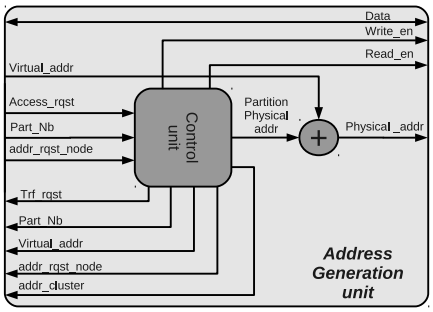


Figure 4.15 – Architecture of the address generation unit.

The behavior of the AGU is the following, the AGU first check if the access targets the local memory node or a remote memory node. If the access is loca,l the virtual address is converted to a physical address to access the value into the memory node. If it is a remote access the incoming request is forwarded by the AGU to the correct cluster and memory node. This forward procedure is fully transparent to the processor and to the programmers who do not have to take care about the memory node location and position.

This procedures includes additional latency and should be limited as much as possible (see section 4.4.4)

4.4.3.3 Memory Protection Unit

The MPU realizes the allocation and the de-allocation of the memory partitions, and ensures that a read and a write operations are not performed at the same time on the same memory partition. The

MPU works as follows, each processor upon request will be assigned a memory partition, and then this area will serve as a mean of communication for the different tasks of the program. Once the execution achieved the partition is released by the owner. To that end the MPU (Figure. 4.16) is constructed around 4 channels (Write, Read, Alloc and De-Alloc channels) and two arbitrators. Each channel is composed of one FIFO and a FSM which is used to control and limit the access to the memory depending on the partition status. The arbitrators are used to limit the number of I/O going in and out of the MPU.

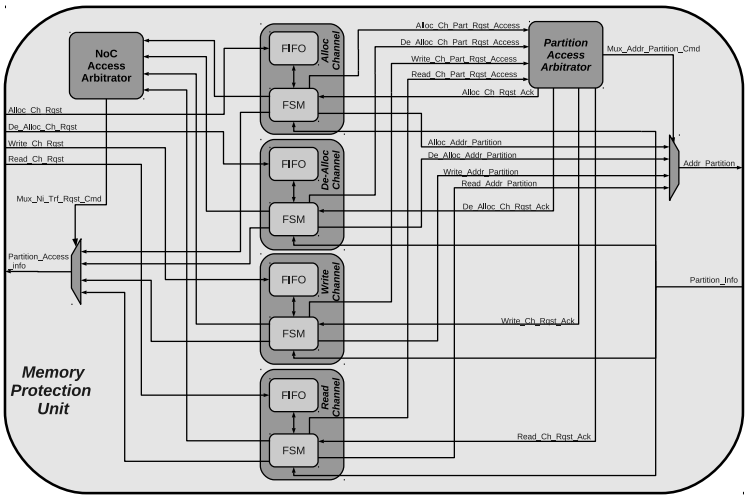


Figure 4.16 – Architecture of the memory protection unit.

4.4.4 Operating Principle

4.4.4.1 Intra cluster communication

The sequence used by the processors to enable a communication channel within the same cluster is depicted in Figure 4.17. In this example the GPP_0 works on data that need to be transmitted to the GPP_1. To ensure this the GPP_0 ask to the MMU a memory partition allocation.

Once the partition allocated, the GPP_0: 1) transmits the number of the allocated partition to the GPP_1, 2) ask for a write access in the memory node (MMU_C0 and Memory_C0), and 3) waits for the write acknowledge. Once the data written in the memory node, the GPP_0 informs the MMU that the write operation is done and tells the MMU the number of processors that need to read the data written into the partition. From this point the partition is only readable until all readers read the data. The number of reader for a partition is deducted from the application task graph at compile time.

The GPP_1 then asks for a read access onto the partition. Since data have been written the GPP_1 read request is acknowledged. At the same time the GPP_0 informs the MMU that it can free the partition. However since all the read operations have not been done the free request is queued. As soon as the read operation acknowledged the GPP_1 transfers the data contained in the memory node and informs the MMU that the data has been read which enables the release of the partition.

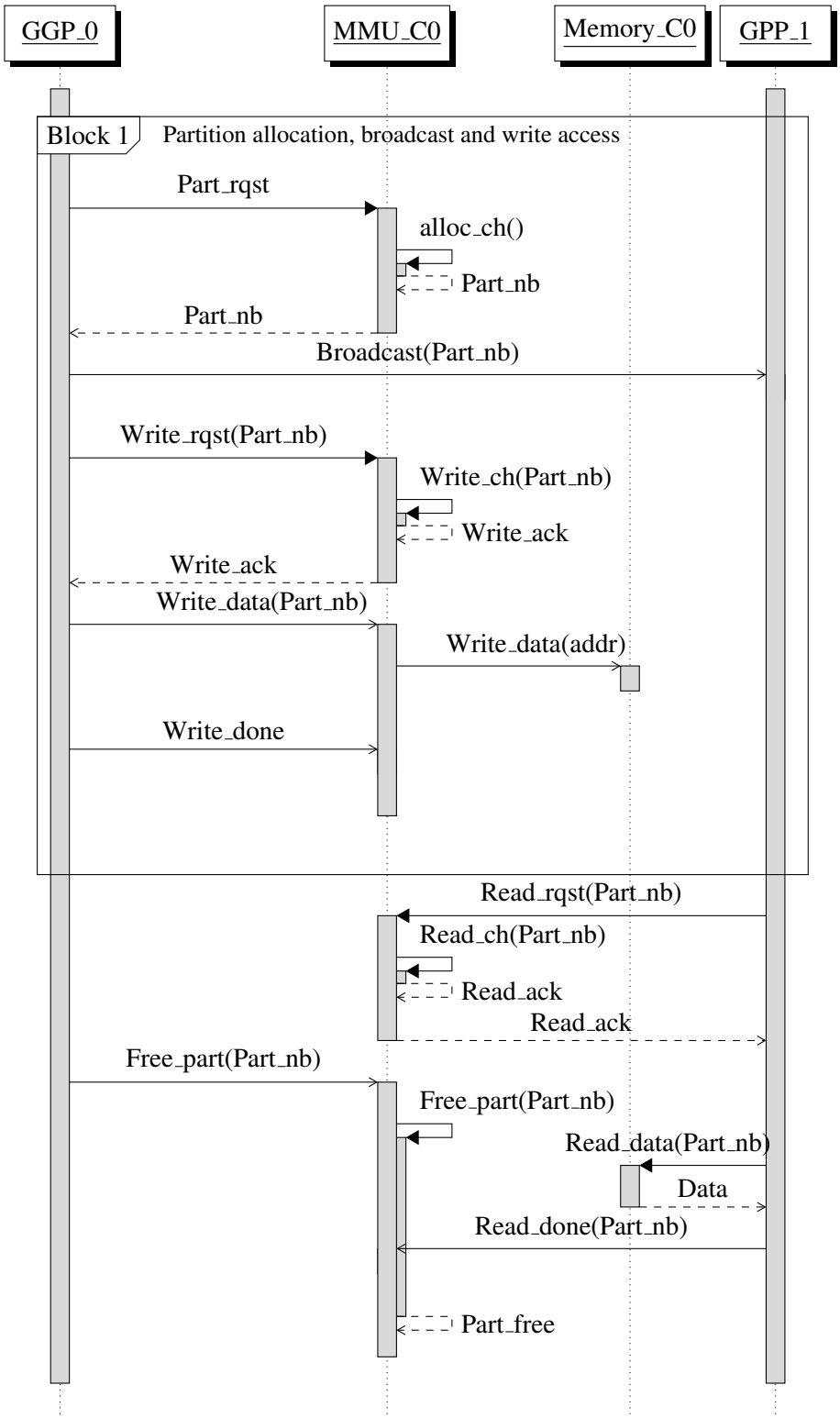


Figure 4.17 – Example where two GPP within the same cluster are communicating through a partition of the memory node.

4.4.5 Inter cluster communication

The sequence used by the processor to enable a communication with a remote cluster is depicted in Figure 4.18. In this example the GPP_0 works on data that need to be transmitted to the DSP_0. The GPP_0 is located into the cluster 0 while the DSP_0 is located into the cluster 1.

As for the intra cluster communication and depicted within the block 1 of the Figure 4.17 the first steps are to request a partition, broadcast the partition number, ask for write access and write the data into the partition. Since this sequence is similar to the one of the Figure 4.17 the block 1 is not represented into the Figure 4.18 for simplifications reasons.

From this point and once the data are written into the memory the DSP_0 requests a read access to the partition to the MMU_C1. However since the partition is not managed by the MMU_C1 the request is automatically forwarded to the MMU_C0. Once the read request received by the MMU_C0, the DSP_0 read request is acknowledged. From this step the GPP_0 inform the MMU_C0 that it can free the partition. However since all the read operations have not been done the free request is queued.

Then the DSP_0 requests to MMU_C1 to transfer the data contained into the partition. Since the partition is not managed by the MMU_C1 the request is automatically forwarded to the MMU_C0. Once the request received by the MMU_C0 the data are transferred to the DSP_0.

The DSP_0 then inform the MMU_C1 that the read operation is done which transfer the request to the MMU_C0. This enables to release the partition by the MMU_C0 since all the read operation have been done.

It is important to note that even if requests are queued for a partition it is still possible to access the other partitions. The hardware MMU is constructed in a way to ensure that any request is blocked into the FIFO.

Furthermore the positions of the processors inside the architecture do not impact the procedure to access the data. This is a key point of the proposed approach since the communications are independent of the processors positions which allow the creation of a generic communication model for the architecture and partially hides to the programmer the underlying memory hierarchy to ease the mapping and data placement issue.

Moreover since the MMU is hardwired the communications are more efficient. The hardware MMU is able to ensure dynamically the memory consistency and management based on the incoming processor requests. Finally as shown on this example it is possible to read data into remote clusters, but also to write data into remote clusters in a transparent manner.

4.4.6 Experimentation and results

4.4.6.1 Simulation environment

In order to validate our approach, OVP and SpearDE environments were used for modeling and simulation purpose. In our context, OVP was used to model and validate the cluster based MPSoC architecture. The cluster based MPSoC architecture is composed of four clusters, each cluster is organized

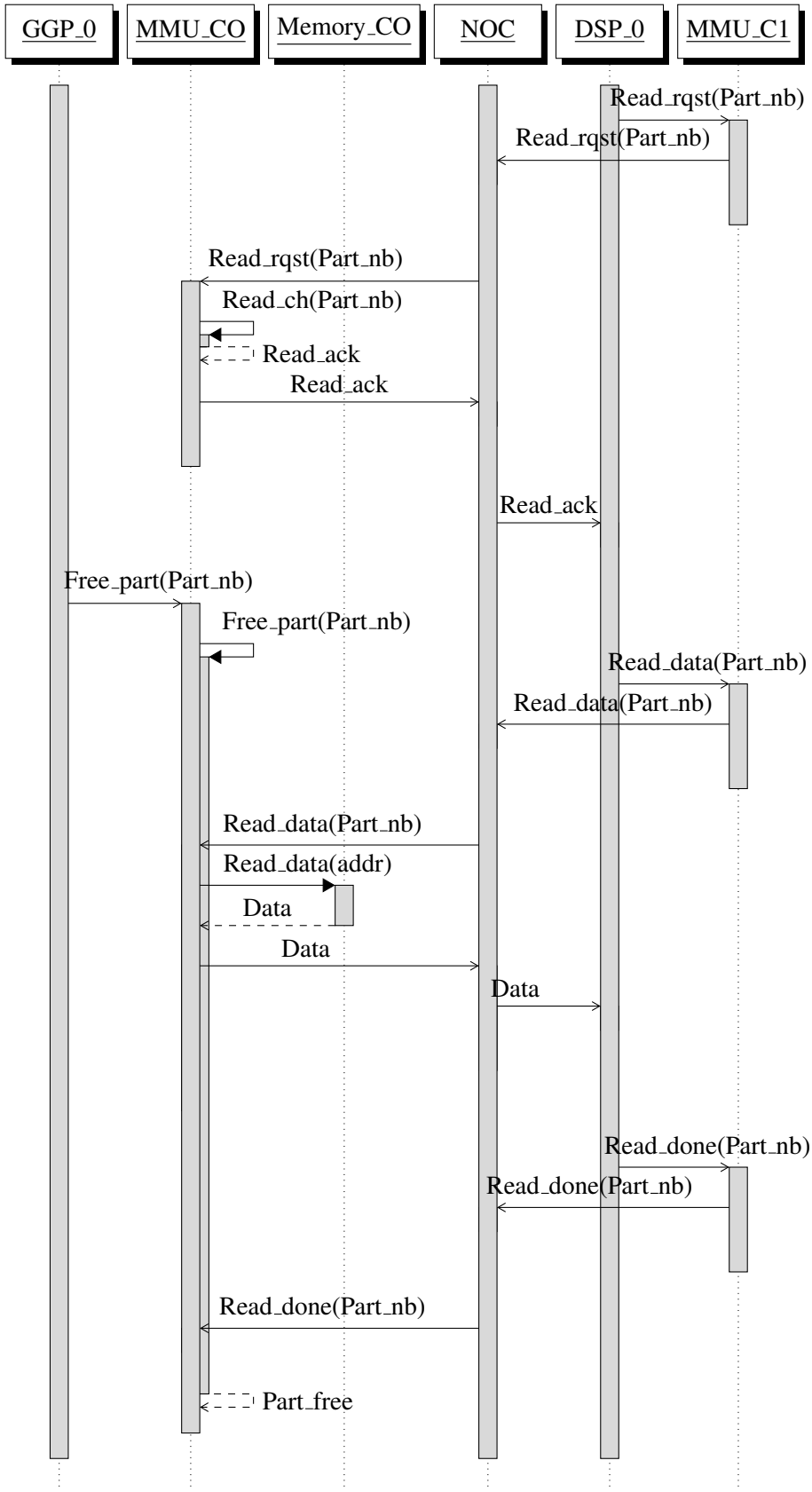


Figure 4.18 – Example where a GPP and a DSP located into remote clusters are communicating through a memory node.

around several processor nodes (2, 4 or 8) and a memory node able to manage four partitions simultaneously. A processor node is composed of a MIPS processor with a local memory and peripherals for synchronization and data transfer purpose. Once the behavior of our solution validated within the OVP framework, SpearDE software environment was used to get performance evaluations.

Regarding the comparison and validation of the results, three architectural models were implemented in SpearDE:

1. Our cluster based architecture with globally distributed locally shared memories (Fig. 4.19).

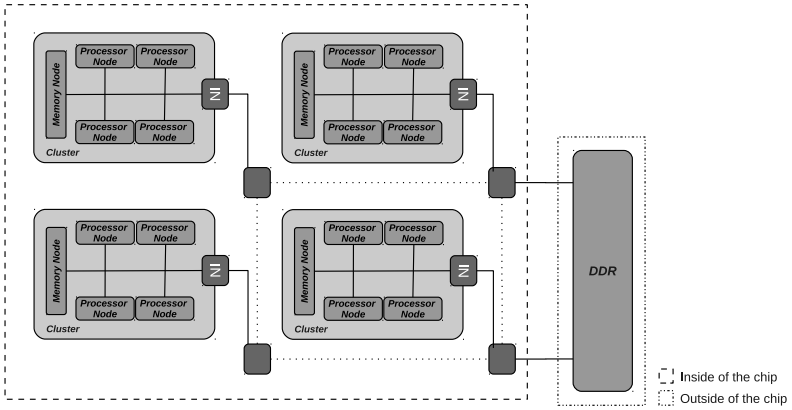


Figure 4.19 – OVP model of cluster based architecture implementing an on-chip MP protocol.

2. The Ter@ops architecture model (Fig. 4.20) used to compare our results with the one of the state of art. The Ter@ops architecture was chosen because it is close from the proposed cluster based architecture which allow a fair comparison. Moreover this allow to show the benefit of the hardware MMU since as stated during the state of the art the memory space of the Ter@ops architecture is managed at the software level.

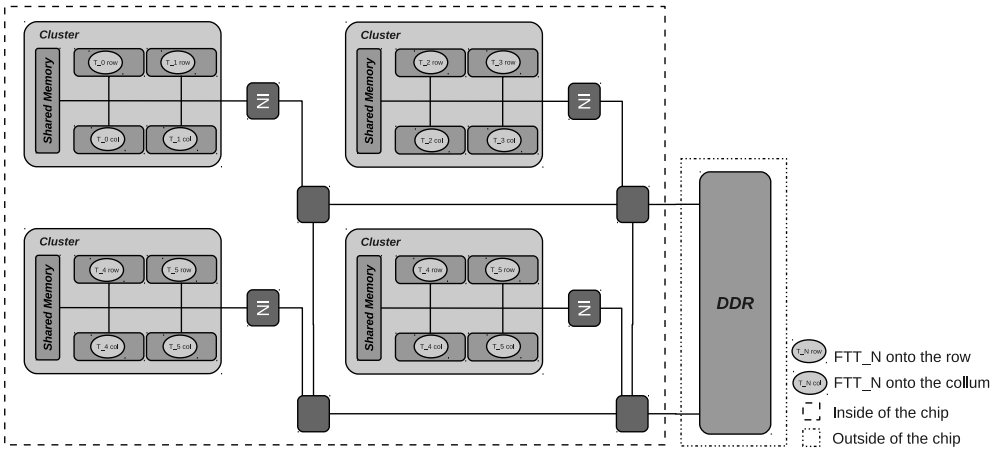


Figure 4.20 – Ter@ops architecture.

3. The architecture shown in Figure 4.21, which is a flat architecture model mostly used in current MPSoC architectures [133, 124]. In this model each processor is associated with its own local

memory and peripherals. The communication protocol between the processors is based on a MP paradigm. The MP paradigm is implemented thanks to the message passing interface [16] standard. Moreover as for the cluster based and the Ter@ops architectures all the processors can access to a DDR memory, this memory is used for data storage purpose when the memory space available on-chip is not sufficient.

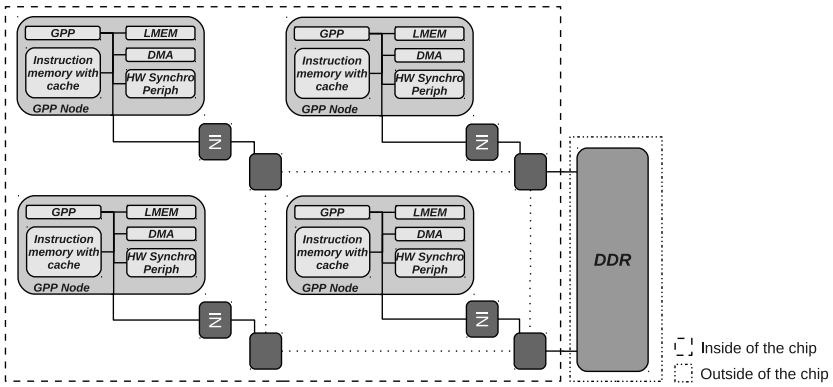


Figure 4.21 – Flat architecture model used to compare the benefits brings by the cluster based architecture.

The set of all the architectures tested and modeled in these experiments are reported in Table 4.6. This table shows that seven different architectures have been tested. Three architectures (Flat_1, Flat_2, Flat_3) are based on the flat architecture model and composed of 8,16 and 32 cores respectively. Three architectures (Clus_1, Clus_2, Clus_3) are based on the cluster based architecture model and composed of 8,16 and 32 cores respectively. One architecture is based on the Ter@ops architecture model. Since the Ter@ops architecture model is composed of 4 cores per cluster the Ter@ops architecture is only compared with the results obtained by the related Clus_2 and Flat_2 architectures.

Table 4.6 – List of evaluated architectures. The number of cores varies from 8 to 32. While the amount of memory present on the chip and the memory bandwidth stay constant.

Architecture Name	Cluster based	Nb of cores	Nb of cores per cluster	Nb of cluster	Mem node bw Mb/s	Size LMEM Kb	Size mem node Kb
Flat_1	No	8	N\A	0	0	288	0
Clus_1	Yes	8	2	4	6400	32	512
Flat_2	No	16	N\A	0	0	160	0
Clus_2	Yes	16	4	4	6400	32	512
Flat_3	No	32	N\A	0	0	64	0
Clus_3	Yes	32	8	4	6400	32	512
Ter@ops	Yes	16	4	4	6400	32	512

Moreover as shown on Table 4.6 the amount of memory present on the chip and the memory bandwidth are kept constant. The bandwidth for the local memory and for the DDR memory of all the architectural models is set respectively to 1600 Mb/s and 25600 Mb/s. This allows a fair comparison of the various architectural models, since each model present the same hardware cost in terms of memory.

4.4.6.2 Application

The applications that were implemented in order to measure the impact of our approach on the performance are a 2D-FFT and a matrix multiplication. These applications were chosen because they present different levels of complexity in the memory access pattern.

To measure the benefit of the cluster based architecture on the data transfer time and when the memory are correctly and incorrectly sized, the two applications were tested with 4 input matrix sizes (128*128, 256*256, 512*512, 1024*1024). All the input data are complex numbers encoded on 64 bits (32 bits for the real part and 32 for the imaginary part).

Moreover we also measure the task overlap of the communication time onto the computation time, to evaluate the benefit of the proposed memory hierarchy onto the performance and onto the data transfers among the platform.

Eq. 4.1 depicts how the task overlap is computed. t_iover is the task overlap, t_ict represent the communication time of task t_i while t_iet represent the time needed by the task t_i to process the data.

$$t_iover = \frac{t_iet}{t_ict}; \tag{4.1}$$

A good memory hierarchy is a memory hierarchy where the task overlap is greater than one. This means that the time needed to compute the data hides the time needed to transfer then.

In the context of these experiments the mapping of the applications on the architecture and the data placement are supposed to be ideal. Which means that the overlap is maximized in regards of the architecture capabilities.

2D-FFT:

The application task graph used for the 2D-FFT application (Figure 4.22) is composed of two tasks: One realizing the FFT onto the lines, and one realizing the FFT onto the column. As can be seen from the application tasks graph a corner turn is needed between the two tasks executions in order to reorder the data this operation is done by a DMA.

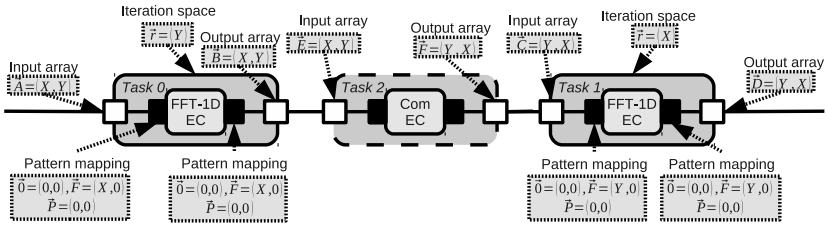


Figure 4.22 – 2D-FFT application task graph.

The results obtained for the 2D-FFT application are presented Figure 4.23, Table 4.7 and 4.8. Our solution realizes an average speedup of 6.98% over a flat architecture, and an average speedup of 3.25% over the Ter@ops architecture. Examples of the 2D-FFT application mapping onto the Clus_2, Flat_2 and Ter@ops architectures are given annex B.2.1.

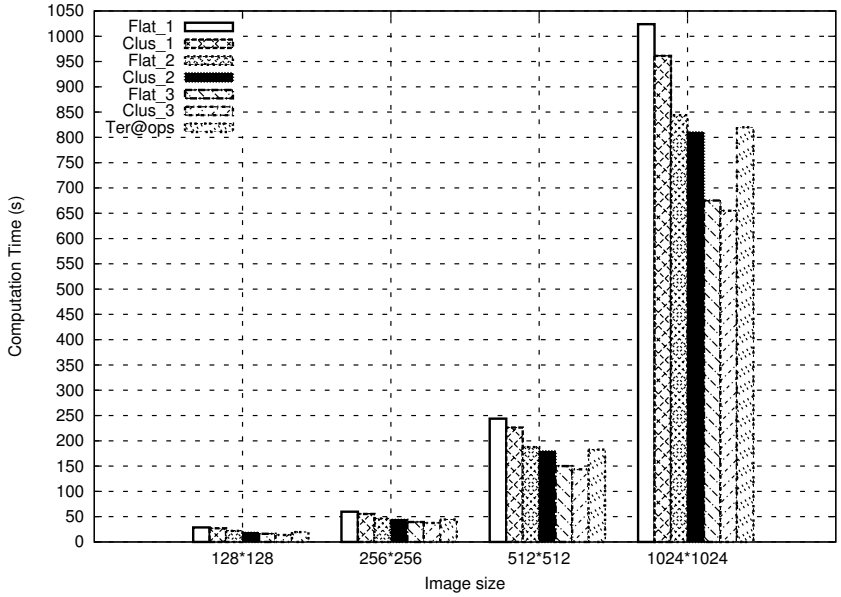


Figure 4.23 – Execution times needed to realize a 2D FFT on 32 images.

Table 4.7 – Speedup of the cluster based architecture over the flat architecture and the Ter@ops on the FFT application.

	Speedup over the classical flat architecture				Speedup
Matrix size	Clus_1/Flat_1	Clus_2/Flat_2	Clus_3/Flat_3	Average	over Ter@ops
128*128	5.9%	17.13%	14.29%	12.4%	8.1%
256*256	7.85%	5.61%	4.82%	6.1%	2.01%
512*512	7.24%	4.31%	3.53%	5%	1.7%
1024*1024	6.13%	4.00%	3.10%	4.41%	1.2%
Average				6.98%	3.25%

In the context of a 2D FFT the impact of the cluster based approach is limited. This is due to the fact that in order to realize the corner turn between the two FFT a complete matrix is needed.

In the context of these experiments over a matrix size of 128*128, a partition of the memory node is not able to store one complete matrix, one of the solutions is then to allocate several partitions. These partitions can be allocated in the local memory node or in a remote memory node.

However this solution is not suitable when the number of cores increase. Indeed some cores would not have any data storage inside the architecture. The unique solution is then to work on sub-part of the input matrix and to write back the obtained results in the DDR memory between the two FFT executions, which strongly impacts the global performance.

Indeed as depicted Table 4.8, when the size of the image is over 128*128 the task overlap is smaller than 1. It is then not possible to hide the communication time with the computation time, and the application execution time is dependent of the data transfer one. Moreover when the number of cores increases the time needed for the communication is getting more important since more bottlenecks on

Table 4.8 – Task overlap for the FFT application.

	Task overlap			
	128*128	256*256	512*512	1024*1024
Flat_1	1.2	0.54	0.39	0.33
Clus_1	1.41	0.71	0.49	0.40
Flat_2	0.72	0.47	0.35	0.32
Clus_2	1.2	0.63	0.43	0.38
Flat_3	0.64	0.4	0.32	0.31
Clus_3	1.03	0.52	0.4	0.37
Ter@ops	0.89	0.41	0.33	0.31

the network and at the memory port implies.

The same problem applies also to the flat architecture model, since as for the memory nodes the LMEMs are not able to contain one complete matrix the results have to be written in the DDR memory between the two FFT executions.

Even if, the memory is incorrectly sized for all the matrix sizes, the cluster based approaches always shows a benefit over the flat architecture model. This is due to the fact that the memory node allows the reduction of the global amount of communication. Moreover when the memory is correctly sized (which is the case for the application running on matrix of 128*128) the observed speedup can go up to 17.13% since the data locality is increased and the data transfers with the DDR memory and with the other processing cores are reduced. Finally when in the context of the flat architecture model, the size of the LMEM is correctly chosen (which is the case for the Flat_1 when working with images of 128*128) the cluster based approach still shows a benefit.

Regarding the benefit of the cluster based approach over the Ter@ops architecture the results shows that the benefit is dependent of the quality of the task mapping and data placement. Indeed when the task overlap of the cluster based architecture is below one the benefit bring is reduced. However even if it is not possible to hide the communication time with the computation time the hardware MMU still allows to optimize the application execution time. Moreover when the task overlap is optimized the benefit bring by the hardware MMU is getting more important.

Matrix multiplication:

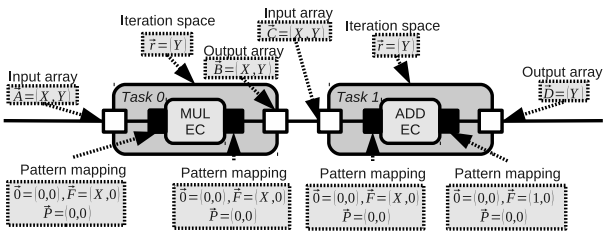


Figure 4.24 – Matrix multiplication application task graph.

For the matrix multiplication (Figure 4.24), the application is also implemented as two sub-tasks one realizing a term to term matrix multiplication and one realizing a sum on the line of the generated matrix.

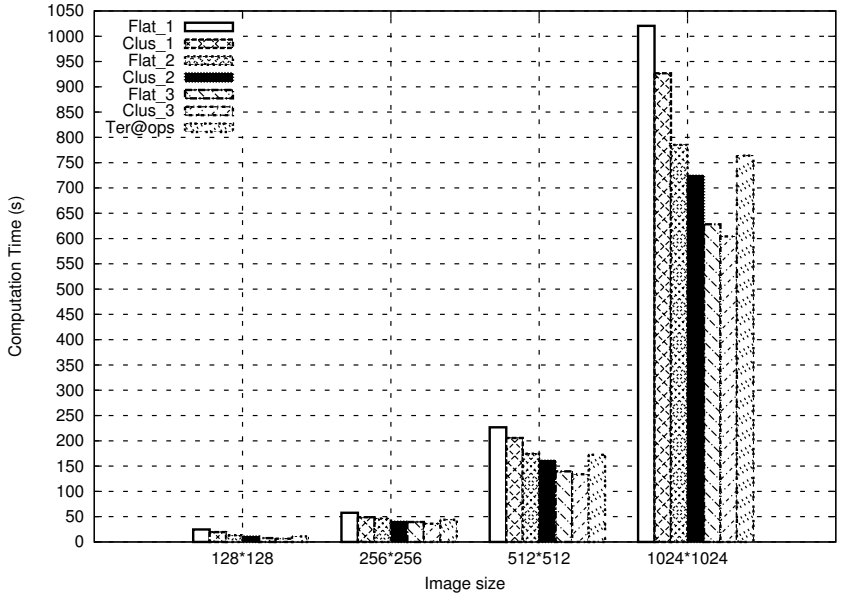


Figure 4.25 – Execution times needed to realize 32 matrix multiplications.

The results obtained for the matrix multiplication application are presented Figure 4.25, Table 4.9 and 4.10. Our solution realizes an average speedup of 14.1% over a flat architecture, and an average speedup of 8.1% over the Ter@ops architecture. Examples of the matrix multiplication application mapping onto the Clus_2, Flat_2 and Ter@ops architectures are given annex B.2.5.

Table 4.9 – Speedup of the cluster based architecture over the flat architecture and the Ter@ops architecture on the matrix multiplication application

	Speedup over the classical flat architecture				Speedup
Matrix size	Clus_1/Flat_1	Clus_2/Flat_2	Clus_3/Flat_3	Average	over Ter@ops
128*128	21.77%	18.84%	15.58%	18.73%	10.2%
256*256	17.40%	15.23%	10.54%	14.39%	9.2%
512*512	13.88%	12.45%	8.80%	11.71%	7.5%
1024*1024	13.73%	12.29%	8.52%	11.51%	5.5%
Average				14.1%	8.1%

In the context of this application, the gains in terms of acceleration are more important. This is due to the fact that only two rows of the complete matrix are required to perform the two tasks (one for each input matrix). In this case it is therefore not necessary to use the DDR memory between the two tasks for data storage. The data locality can be further maintained and allows to take full advantage of the memory node.

Increasing the size of the matrix and the number of cores, reduce the task overlap (Table 4.10). This is due to a more important traffic on the networks and at the memory ports.

We can see that the benefit of the cluster based approach over the Ter@ops architecture is more important (up to 10.2%) when it is possible to hide the communication time with the computation time.

These results demonstrate that the hardware management of shared memory helps to reduce the communication overhead and increase the performance thanks to a more efficient management of the communications.

Table 4.10 – Task overlap for a matrix multiplication application.

	Matrix Multiplication task overlap			
	128*128	256*256	512*512	1024*1024
Flat_1	3	1.8	1.27	0.93
Clus_1	4	2.16	1.5	1.13
Flat_2	2.6	1.6	1.13	0.88
Clus_2	3.5	1.95	1.41	1.08
Flat_3	2	1.46	1.01	0.82
Clus_3	3.1	1.78	1.22	1.06
Ter@ops	3.2	1.62	1.19	0.95

4.4.6.3 Influence of the data placement and task mapping

In order to measure the benefit of the proposed approach this section proposes to measure the benefit brings by the cluster based approach when the task mapping and the data placement are not ideal.

To that end we propose to vary the task mapping and data placement from an ideal one to a poor one and measure the impact of remote data access in context of the proposed cluster based architecture.

These two experiments are done with the matrix multiplication application when working on image size of 128*128. The application is run on the flat_2 and clus_2 architectures and on the Ter@ops architecture. This application was chosen because it fully optimizes the task overlap which allows us to vary it and measure the benefit of using a hardware MMU. An example of a poor multiplication matrix application mapping is given annex B.2.9.

The obtained results following these experiments are resumed Figure 4.26. As shown the cluster based approach always shows a benefit over the flat architecture on the application execution time even if the data placement and task mapping are not ideal.

Compared to the Ter@ops architecture we can see that when the task overlap is below 0.3 the benefit brings by the proposed approach is less than 2%. This is due to the fact that the processors are not loaded and have enough time to synchronize and ensure the memory consistency.

However when the task overlap is over 0.3 the proposed approach bring a benefit on the application execution time. Indeed because the processors are more loaded the hardware MMU helps to reduce the processor idle state, the synchronization events and optimize the application execution time and data transfers when the data access are local to the cluster.

Regarding the impact of remote data access we see that it implies a loss in performance. However even when the mapping cause remote data access the proposed cluster based approach always show a benefit over the flat architecture and propose the same level of performance as the Ter@ops architecture.

In conclusion it's necessary to take a full benefit of this approach to carefully size the memory and

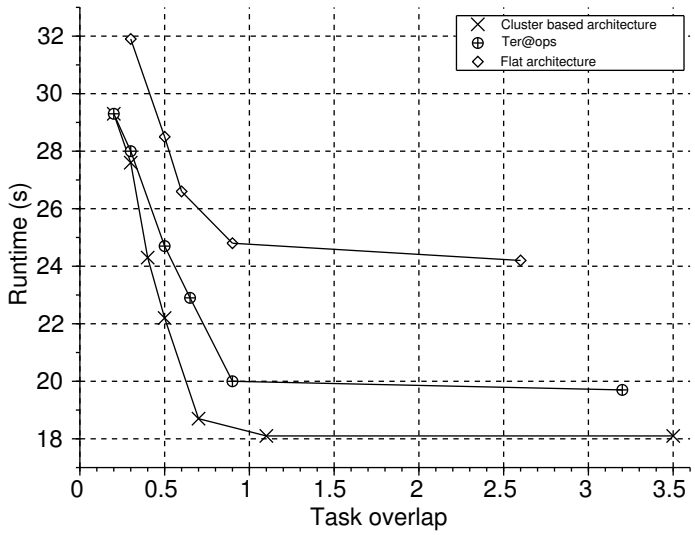


Figure 4.26 – Influence of the task mapping and data placement onto the benefit bring by the cluster based approach.

the number of cores connected to the memory node and to map highly communicating task within the same cluster. This allows the bottlenecks reduction on the network and transfers to the external memory. Nevertheless, despite the fact that the speedup is reduced in case of incorrectly sized memory or remote data transfer, our solution still shows a benefit. This is made possible due to a reduced network traffic, obtained thanks to the cluster organization and the hardware MMU which enable an easy programming thanks to a unified interface.

4.4.7 Hardware Implementation

The results of the implementation of the MMU within a Xilinx Virtex6 SX315T FPGA are resumed Table 4.11.

These results suppose that the MMU is able to manage the consistency of four partitions simultaneously. The request FIFO of each channel is limited to 4 requests. Which turn to be sufficient in the context of the proposed approach. The router, the NI and the processor node used for hardware resources comparison are the same as the one used into section 4.3. The results given for the MMU do not include the protected memory.

Based on these results we can see that the MMU operates at a high frequency while occupying a reduced amount of hardware resources compared to the other component of the architecture. Indeed as shown on Table 4.11 the MMU is equal to the size of a NI and much smaller than the size of a processor and a router in terms of slice.

This is due to the fact that the FSM implemented into each channel to control the memory consistency are quite simple and organized around a reduced number of states. Finally the two arbitrators

Table 4.11 – MMU resources consumption

	Flip-Flop	LUT	Slice	Freq. (Mhz)	MMU/Component consumption ratio (Occupied slice)
MMU	1635/393600 (0.4%)	970/196800 (0.5%)	818/49200 (1.6%)	450	1
Router	2402/393600 (0.6%)	1425/196800 (0.7%)	1202/49200 (2.45%)	406	0.68
Processor Node	4301/393600 (1.1%)	2551/196800 (1.3%)	2152/49200 (4.3%)	220	0.38
NI	1635/393600 (0.4%)	970/196800 (0.5%)	818/49200 (1.6%)	420	1.01

implemented within the architecture to control and limit the access to the memory or to the NoC are also based on simple FSM to manage the incoming requests.

Furthermore from these numbers we can conclude that it is possible to implement within a Virtex6 SX315T FPGA four clusters composed of 4 processors node connected to a shared memory of 6400MB/-cluster.

4.5 Summary

In this chapter the FlexTiles project which is a heterogeneous high performance MPSoC architecture, constructed around hardware and software mechanisms used to abstract the underlying heterogeneity was introduced.

The AI used within the FlexTiles platform to abstract and give an autonomous behavior to the DSP and the hardware accelerators is presented. A real case implementation of this platform and the associated results are given. These experiments show the AI ability to give an autonomous behavior to the accelerators. Moreover these experiments show the AI ability to abstract the heterogeneity of the platform. These results prove the interest of using the AI in the context of PARSE to ease the VHDL code generation and the construction of hardware architectures.

In order to tackle the challenges introduced by the memory wall and ease the programming of MPSoC architecture we propose a cluster based architecture constructed around a hardware memory management unit. This approach has shown its ability to improve the performance by reducing the data transfer among the platform and the bottleneck that occur at the memory port level. Moreover the hardware MMU implementation has also shown its ability to abstract to the programmer the underlying memory hierarchy, in order to ease the programming of MPSoC architectures. These results show the hardware MMU interest in context of PARSE to ease the mapping and data placement solution space exploration.

5

Conclusions and perspectives

Abstract: This chapter summarizes all the contributions presented during this thesis and proposes some perspectives beyond this work.

Contents

5.1 Conclusions

The rising computing need of embedded applications requires the definition of more and more high performance computing architectures. The embedded hardware architectures then have to integrate more and more functionalities to face these increasing demands.

In order to tackle these challenges the designer are moving towards MPSoC architectures. These architectures bring the possibility to integrate within the same chip several complex functionalities all in a restricted power budget. However the conception, the design and the programming of these platforms are difficult and time consuming.

To face these challenges and enable an efficient design and use of the MPSoC architectures this thesis has first proposed to study the state of the art of these architectures and of the framework used to design and program them. This study has shown that there exist a lot of MPSoC architectures which are used into several domains of applications and for different purposes.

These architectures present a lot of heterogeneity both in terms of design and programming models in order to be adapted to the targeted domain of application. This heterogeneity is needed in order to face hard constraints imposed by the applications. However this heterogeneity also greatly increases the complexities to conceive and program MPSoC architectures. The MPSoC conception and programming is then dependent of a set of parameters at the architecture and at the application levels that mutually influence each other. The main parameters are the choice of:

- The interconnection hierarchy and bandwidth
- The memory hierarchy and bandwidth
- The number and types of cores
- The application mapping
- The data placement

To help the designer define and conceive their MPSoC architectures the state of the art has shown that a lot of tools and initiatives have tried to ease the user tasks. The main limitations of these approaches are due to the fact that the exploration of the solution space is mainly done by hand which is time consuming.

Furthermore the DSE tools present into the state of the art are not able to consider at the same time the application and the architecture. Indeed in most of the cases either the application or the architecture is known. This approach then creates a separation of concern between the application and the architecture and leads to a sub-optimal solution and to an inefficient use of the hardware resources.

To tackle these challenges and ease the definition and the programming of MPSoC architectures this thesis has proposed a DSE methodology which consider at the same time the definition of the architecture, the application, the mapping and the data placement of the application onto the architecture (It is however possible as the state of the art tools to start with a known architecture model and focus on the application mapping and data placement).

The proposed methodology is a semi-automated approach where the most exhaustive tasks are performed automatically and the role of the user limited. To automatically go through the solution space a tool called PARSE was defined. PARSE is based on recursive mechanisms to allow the exploration of the solution space in an automated manner. However in order to keep the exploration into a reasonable amount of time and focus on the most promising regions, PARSE takes as input a parsimonious

representation of the solution space.

The goal of this representation is to reduce the size of the solution space based on the user experience. To that end the user defines for each task a set of potential processing IPs, which allows to remove from the solution space the uninteresting design points. This representation is needed considering the size of the solution space generated by the mutual exploration of the architecture and application aspects.

Moreover to have the more accurate parsimonious representation of the solution space PARSE automatically computes a set of metrics prior to the exploration of the solution space. These metrics determines: (1) the bandwidth needs of the application and (2) the load of each core based on the architecture and application mapping. These metrics are then used by PARSE to detect mappings with overloaded cores and which generates too many communications among the platform. The uses of these metrics avoid the accurate exploration of uninteresting design points, help PARSE to focus on the most promising regions of the solution space and to find the best compromise for the application and the architecture.

Moreover thanks to these metrics we have defined a mixed evaluation heuristics. Indeed current techniques are mainly based on an analytic formula for performance evaluation. This is a main drawback since it becomes highly complex to accurately model the communication topology and architecture behavior of a MPSoC architecture with an analytic formula. The use of SystemC simulator to get performance evaluation of an application mapping then becomes mandatory to have an accurate representation of the solution space. However due to the size of the solution space each design points cannot be evaluated by the use of a SystemC simulator due to the exploration runtime overhead.

To that end and based on the metrics results the most promising design points are evaluated thanks to a SystemC simulation while the others designs points are evaluated thanks to an analytic formula to not affect the exploration process. This approach allows the reduction of the exploration runtime while maintaining an accurate representation of the solution space. In the context of this thesis the data placement and mapping heuristics were implemented and tested jointly and in a standalone mode on both benchmarks and real live applications.

The results obtained by the proposed tabu search reduce space framework in a standalone mode have shown that we are close or equal to the optimal solution found by the state of art. Experimental results showed that the proposed framework provides an effective exploration of large design space and results close or equal to exhaustive approach. Moreover the exploration runtime is close to the one get by the state of the art which are based on a simple analytic formula while our approach is based on SystemC simulation which is much more time consuming. This proves the interest of the proposed metrics since we propose a more accurate representation of the solution space without needing more time to explore it.

The results obtained by the genetic algorithm with fusion operator framework in a standalone mode have also shown that the solutions produced are close or equal to the optimal. The fusion operator used to explore the neighborhood of the current population has shown its ability to help the algorithm to converge to the most promising regions of the solution space. Moreover in terms of exploration runtime the GAFO framework exhibits an important speedup over the exhaustive approach without impacting the quality of the final solution.

Finally the results obtained by the TSRS-GAFO framework were of high quality and close or equal to the optimal solution. Moreover the parsimonious representation proves its interest and is ability to reduce the exploration time and the number of solutions explored accurately without having an impact

on the quality of the final solution. The results obtained by the TSRS-GAFO both in terms of quality and exploration runtime have shown that it is feasible to explore at the same time the architecture and the application to find the best compromise for both of them.

Still with the aims to ease the programming and the definition of MPSoC architectures this thesis has also defined two hardware modules. The first hardware module is the accelerator interface which is used to: (1) hide the heterogeneity of the IPs connected onto the interconnect and (2) offload the processor of the fine grain control of the accelerators.

The abstraction of the heterogeneity in context of MPSoC architectures is needed to ease their definition and programming and take a full advantage of the computing power induced by the dedicated processing IPs. With the AI the master processor still sees the same interface and does not have to take care about the targeted IP and the way to program it. This enables the concept of plug and play architectures where it is easy to add or remove IPs onto the interconnect without taking care of their specificity.

The experiments done in the context of the FlexTiles project have demonstrated that the AI is able to target any kind of processing IP and interconnect, without having an impact onto their architecture and onto the performance. Moreover the hardware cost overhead introduced by the use of the AI is limited in regards of the other component of the platform. While the latencies induced by the AI to transfer the data from the NoC to the accelerators are limited and just applied onto the first data transfer since the AI present a pipelined behavior.

Furthermore the AI is conceived in a way that allows the master to be offloaded from management of the accelerator with the other components of the platform and enables it to do other processing in parallel. The AI then allows to fully exploit the computing power of MPSoC architectures by increasing their parallelism. Finally the AI thanks to the abstraction of the heterogeneity allows the reduction of the time to market by easing the programming of the platform.

The second hardware module defined in the context of this thesis is a MMU used to manage the consistency of a shared memory in context of a globally distributed locally shared MPSoC architecture. The main advantages of the proposed approach over the state of the art are: (1) the use of a globally distributed and locally shared memory hierarchy, (2) the definition of a generic communication model enabled thanks to the hardware MMU, (3) the reduction of the communications among the platform.

Thanks to these features, the architecture is able to dynamically allocate and manage the available memory space. Furthermore the task of the programmer is reduced. Indeed the underlying memory hierarchy is transparent since the communications are made generic thanks to the hardware MMU. The experiments done have shown that the proposed approach always shows a benefit over a state of the art architecture and a traditional flat architecture. The benefits on the application execution time bring by this module are due to the fact that the data locality is increased and the data transfer among the platform limited.

Through this thesis and thanks to the definition of the DSE methodology we have proposed solutions to ease the programming and the conception of MPSoC architectures. More particularly we have proved that it is possible to define at the same time the application and the architecture in order to find the best compromise for both of them. Moreover to tackle the MPSoC heterogeneity and programming problems we have introduce the AI and the hardware MMU.

5.2 Perspectives

5.2.1 Mid-term perspectives

During the thesis a complete DSE methodology along with hardware modules have been defined to ease the definition of MPSoC architectures. There exists anyway several future works to enhance our proposal and perspectives beyond this thesis.

In chapter 3 a detailed description of the proposed DSE methodology was given and the data placement and mapping heuristics implemented. Through the implementation and test of the two heuristics we conclude that it is not feasible to allow an accurate exploration of each design points of the solution space.

To reduce the exploration runtime a set of metrics for the mapping heuristics have been defined to avoid for each designs points the call to the data placement heuristic and to the SystemC simulator. However this set of metrics need to be further extended to the architecture exploration heuristic. To that end it seems necessary to be able to generate prior to the exploration of the solution space a set of templates to help the user determine a range for the memory sizes/bandwidth, interconnect topology/bandwidth...

The profiling of the application should then allow to extract from the application tasks graph and the architecture library all the templates needed by PARSE to avoid the accurate exploration of uninteresting design points. To generate these templates, it seems interesting to look at these works [31, 172, 194] for detection of the interconnect contention, [110, 134] for the memory concurrent access evaluation and [119] to define resource aware architecture.

However these approaches are based on already known architectures and mapping models, which in the context of PARSE is not the case. To keep the PARSE philosophy which aims to automate the exhaustive tasks, it is then needed to propose an efficient manner to reduce the architecture solution space prior to the generation of an architecture or a mapping. In the context of the proposed approach it is necessary to identify the potential bottlenecks that can occur at the memory ports or onto the interconnect.

A possible solution can be to profile the access pattern of the task to its local memory. By correlating these results with the processor frequency it is then possible to have for each task an idea of the frequency of the access to the memory and to the interconnect. Since the accesses to the interconnect are dependent of the size of the memory these results will allow to determine for the application a memory size, a memory bandwidth and an interconnect bandwidth range guaranteeing the performance. Moreover by an analysis of the application task graph and parallelism it seems possible to determine if it is more beneficial to guide the exploration towards a shared or a distributed memory architecture.

Through the enunciation of these few points we see that it is possible to determine a set of metrics and templates able to reduce the exploration runtime. However the definition of this framework has to be done and to be validated for different application complexities and domains. Indeed the choice made here will impact the architecture solution space exploration and so the quality of the solution provided as outputs.

Still in order to reduce the exploration time works have to be done at the SystemC simulator level. Indeed this step is the most time consuming task when doing the exploration of the solution space. To reduce the impact of the SystemC simulator generation a possible solution is to determine simplifying

assumptions that allows the generation reduction time while keeping an accuracy that allows a good representation of the solution space to avoid the definition of inefficient solutions.

Another perspective to reduce the exploration runtime is to take benefit of the inherent parallelism of PARSE. To that end it should be beneficial to look at the parallelization of the heuristics used to explore the solution space in order to speed up the exploration runtime. There exist already initiatives which have tried to parallelize genetic algorithm like [30, 77, 129, 143]. However these works need to be further extended to the tabu search heuristics and adapted to the PARSE approach and problematic.

One key aspect of PARSE is also the adjustment of the parameters (number of iteration, memory size, FO rate...) of the TS and GA algorithms. Indeed these parameters are dependent of the application and architecture complexity and have to be tuned regarding this complexity. Since the choice of the parameters have a big impact on the quality of the final solution it is mandatory to determine these parameters automatically to offload the user from this difficult task.

To that end based on the statement of [101, 102] it seems possible to determine templates allowing an automatic definition of these parameters. However since these parameters have a big influence on the quality of the final solution it is mandatory to have a high level of accuracy for the templates. This requires to intensively train these templates. Moreover this requires to clearly identify the architecture and the application aspects determinant for the parameters choice.

5.2.2 Long-term perspectives

PARSE actually defines one architecture for one application. However in more and more embedded system the same architecture is used for different applications. The best compromise then has to be fond for all the applications and for the architecture.

This multiple application exploration then generates a huge solution space either at the architecture level or at the mapping and data placement level. To handle this exploration in a reasonable amount of time the parsimonious representation and the use metrics is mandatory and have to be extended.

Another perspective beyond this DSE methodology is to handle dynamic application workload. Indeed we have seen that more and more application are becoming dynamic and exhibit different behavior based on sensors informations.

The conception of architectures for these kinds of applications then becomes much more difficult since they have to be adapted for different application behavior. To ease the definition of these types of architectures the exploration heuristics then have to define an architecture, a mapping and a data placement adapted to each workload.

This dynamicity therefore increases the number of potential solutions and the size of the solution space explored. The use of a parsimonious representation of the solution space and the generations of templates in the future then becomes crucial in order to keep the exploration into a reasonable amount of time.

Moreover in this thesis we do not consider the dynamically reconfigurable architectures. However these architectures as the FlexTiles architecture are more and more present in the market. Indeed they offer the possibility to adapt its architecture based on the application workload.

To handle this dynamically reconfigurable architectures PARSE have to be able to measure the time needed to reconfigure the architecture. Since the reconfiguration time can be long PARSE will then have to evaluate based on the task workload if it more beneficial to allocate a reconfigurable resources or to run the task on a processor. This decision has to be taken in regard of the user constraints, the task mapping and scheduling and also in regards of the tasks needs in terms of processing IP.

This latter will force us to consider an on future MPSoC. This OS will be part of the solution and should be included in the design of future platforms.

Personal Publications

- **Journal:**
  — R. Brillu, S. Pillement, F. Lemonnier, and P. Millet. Cluster based mpsoc architecture: An on-chip message passing implementation. *Design Automation for Embedded Systems*, 2014
- **Patent:**
  — R. Brillu, S. Pillement, F. Lemonnier, and P. Millet. Design space exploration for hw/sw codesign of mpsoc, 2014
- **Conferences**
  — R. Brillu, S. Pillement, F. Lemonnier, P. Millet, E. Lenormand, M. Bernot, and F. Falzon. Towards a design space exploration tool for mpsoc platforms designs: a case study. In *Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, 2014
  — R. Brillu, S. Pillement, F. Lemonnier, P. Millet, M. Bernot, and F. Falzon. Algorithm-architecture adequacy, an application to the phase diversity algorithm. In *GRETSI*, 2013
  — R. Brillu, S. Pillement, and F. Lemonnier. Environnement de modélisation et de simulation d'architecture mpsoc: Ovp une solution fiable et effective ? In *Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS)*, 2013
- **Workshops**
  — R. Brillu, S. Pillement, A. Abdellah, F. Lemonnier, and P. Millet. Flextiles: A globally homogeneous but locally heterogeneous manycore architecture. In *Workshop on Rapid Simulation & Performance Evaluation: Methods and Tools of the HiPEAC conference*, 2014
  — R. Brillu, S. Pillement, F. Lemonnier, and P. Millet. Accelerator interface, a keystone for heterogenous "mpsoc" architectures. In *DATE Friday Workshop on Reconfigurable Computing V2.0: The Next Generation of Technology*, 2013
- **Seminar**
  — R. Brillu, S. Pillement, F. Lemonnier, and P. Millet. Flextiles a heterogeneous "mpsoc" architecture. In *GDR SoC-SiP*, 2013

Abbreviations

3GPP: 3rd Generation Partnership Project
AGU: Address Generation Unit
AI: Accelerator Interface
ASIP: Application Specific Instruction set Processor
BB: Branch and Bound
CA: Classic approach
CGA: Classical Genetic Algorithm
CMP: Chip Multiprocessor
CMT: Chip Multithreading
COTS: Commercial On the Shelve
CPU: Central Processing Unit
CSDF: Cyclo Static Data Flow
CSDFG: Cyclo Static Data Flow Graph
CTS: Classical Tabu Search
DDR: Double Data Rate
DE: Discrete Event
DMA: Direct Memory Access
DMC: DMA Memory Controller
DSE: Design space exploration
DSL: Domain Specific Language
DSM: Distributed Shared Memory
DSP: Digital Signal Processor
DVB: Digital Video Broadcasting
EC: Elementary Computation
EDGE: Enhanced Data Rates for GSM Evolution
FIFO: First In First Out
FO: Fusion Operator
FPGA: Field Programmable Gate Array
FU: Functional Unit
GA: Genetic Algorithm
GAFO: Genetic Algorithm with Fusion Operator
GOPS: Giga Operation Per Second
GPP: General Purpose Processor
GPRS: General Packet Radio Service
GPU: Graphical Processing Unit
GSM: Global System for Mobile

HDL: Hardware Description Language
HLS: High Level Synthesis
I/O: Input/Output
I_NI: Internal NI
I_NoC: Internal NoC
ILP: Integer Linear Programming
IP: Intellectual Properties
ISA: Instruction Set Architecture
ISS: Instruction Set Simulator
ITRS: International Technology Road-map for Semi-conductors.
KPN: Kahn Process Networks
LMB: Local Memory Buffer
LMEM: Local MEMory
LTE: Long Term Evolution
MARTE: Modeling and Analysis of Real-Time and Embedded systems
MI: Memory Interface
MIMD: Multiple Input Multiple Data
MIU: Mesh Interface Unit
MMU: Memory Management Unit
MOC: Model of Computation
MP: Message Passing
MPSoC: Multiple Processors System on Chip
MPU: Memory Protection Unit
NI: Network Interface
NoC: Network on Chip
OS: Operating System
PARSE: Parsimonious ARchitecture Space Explorer
PE: Processing Element
PSO: Particle Swarm Optimization
QEMU: Quick EMUlator
QoS: Quality of Service
RTL: Register Transfer Level
SBR: Space Based Radar
SDR: Software Define Radio
SDF: Synchronous Data Flow
SDFG: Synchronous Data Flow Graph
SHMEM: Shared MEMory
SMP: Symmetric MultiProcessor
SM: Streaming Pultiprocessor
SP: Streaming Processor
SysML: System Modeling Language
TDMA: Time Division Multiple Access
TOPS: Tera Operation Per seconds
TPC: Texture/Processor Cluster
TS: Tabu Search
TSA: Tabu Search Algorithm
TSRS: Tabu Search Reduce Space

UAV: Unmanned aerial vehicle
UML: Unified Modeling Language
UMTS: Universal Mobile Telecommunications System
VHDL: VHSIC Hardware Description Language
VHSIC: Very High Speed Integrated Circuit
VLIW: Very Long Instruction Word
XML: Extensible Markup Language

Annex A

Contents

A.1 Methodology implementation

The following section proposes an approach to implement the DSE methodology proposed in chapter 3 . The tools proposed in this section are not mandatory and can be replaced by any other tools with the same functionalities.

A.1.1 Library of architecture

To present a standard representation of the IPs, all the IPs are captured with the IP-XACT standard [35]. IP-XACT is an XML format that defines and describes electronic components and their designs and enables automated configuration and integration through tools. This standard is used because it is the main contribution in the field. However due to some semantics limitations [83], the use of IP-XACT implies to make additions to the standard to more efficiently represent the architectures.

A.1.2 Code parallelization

In order to extract the application tasks graph from the sequential C code description, PIPS [19] is used. PIPS is a source-to-source compilation framework for analyzing and transforming C and FORTRAN programs. The goal of the PIPS project is to develop a free, open and extensible workbench for automatically analyzing and transforming scientific and signal processing applications.

In the context of the proposed methodology PIPS is used because it is already used as a front end by the SystemC and binary code generator for parallelism extraction (see section A.1.4).

A.1.3 Profiler

For the profiling, OVP tool [18] is used due to its rich library of standard processors (ARM, Microblaze, PowerPC ...) and high quality API to describe new models. The rich library of standard processors avoids the user to maintain the model library for these standard processors. The only models that need to be included and maintained into the OVP library are the user processing cores.

A.1.4 SystemC simulation and Binary code generation

For the SystemC simulation and binary code generation SpearDE is used. The use of SpearDE allows to covers two needs at the same time, which reduce the time needed to implement the DSE methodology.

Moreover SpearDE have already integrated a code parallelization tool (PIPS) into its framework.

A.1.4.1 HDL code generation

For the HDL code generation, a specific HDL code generator will be defined. This HDL code generator will be able to generate architecture based on the AI.

A.2 Justification of the Tabu search Parameters

The choice of the TS parameters is dependent of the application and architecture complexity [101]. Since these parameters have an important influence on the quality of the final solution, their definition as to be done based on a set of trials demonstrating that the accuracy of the exploration is not improved by a further increase of parameter values [101].

However it is important to note that these experiments do not have to be run for each application and architecture. Indeed as stated by [101] the same parameters value can be reused for applications and architecture which present the same complexity.

In the context of these studies these experiments were ran for all the applications. The rest of this section will depict the experiments only for the stap application. All the experiments done to justify the TS parameters follow the methodology proposed by [101].

A.2.1 Justification of the tabu search memory size and of the number of iteration

The choice of the TS memory size has to be done in regards of the number of iteration [101]. Indeed the memory size is dependent of the number of visited solution. A small tabu memory resulting in a loop phenomenon, while a too large memory result in a dispersion of the search.

To find the appropriate memory size and iteration number the fitness of the final solution is measured. In order to avoid being dependent of the randomly created input solution, the input solution provided for all the tests is the same and the diversification operator is not triggered.

Table A.1 – Evolution of the best solution with the memory size and the number of iteration for the stap application.

		Number of iteration				
Memory size		2	4	6	8	10
TSRS	2	0.42	0.41	0.42	0.46	0.52
	4	0.41	0.39	0.44	0.45	0.47
	6	0.42	0.44	0.40	0.40	0.40
	8	0.43	0.43	0.45	0.40	0.39
	10	0.42	0.43	0.45	0.46	0.41
CTS	2	0.44	0.45	0.41	0.43	0.41
	4	0.46	0.39	0.392	0.398	0.395
	6	0.46	0.41	0.391	0.398	0.391
	8	0.48	0.42	0.40	0.394	0.396
	10	0.49	0.44	0.42	0.399	0.399

The results described Table A.1 show that the CTS and the TSRS perform well with a small number of iteration and a reduce memory size. Indeed the table shows that fitness is minimized for the TSRS framework when the number of iteration is included between 8 and 10 and the memory size between 6 and 8. Thus the number of iteration is set to 10 and the memory size to 8 for the TSRS framework since its the value which gives the better results.

For the CTS framework the fitness is minimized when the number of iteration is between 8 and 10 and when the memory size is between 6 and 8. Thus the number of iteration is set to 10 and the memory size to 6 for the CTS framework since it is the value which gives the better results.

A.2.2 Justification of the tabu search diversification operator level

In order to find the appropriate diversification operator level, we measured the impact of the diversification operator on the final solution. These experiments were performed with a memory size of 8 and 10 iterations for the TSRS and with a memory size of 6 and 10 iterations for the CTS. The value of the diversification operator varies from 2 to 10 by step of 2, since the number of iteration is limited to 10 for the TSRS and the CTS. The obtained results are depicted in Figure A.1.

The results show that the fitness value is minimized for a diversification operator level set between 6 and 8. The value of the diversification operator for the TSRS and the CTS is then set within this interval and is arbitrarily equal to 7 for the two algorithms.

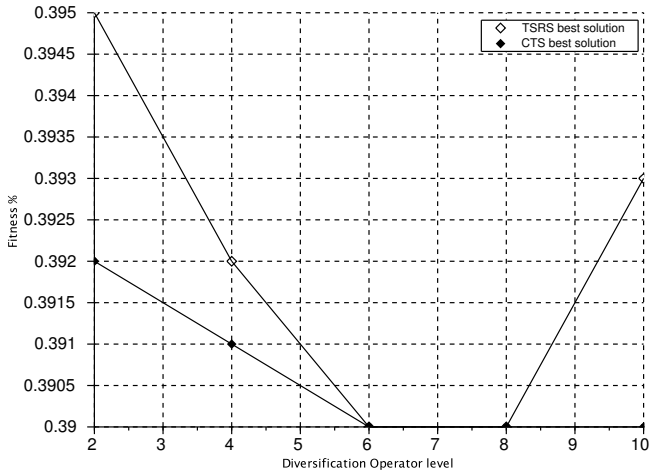


Figure A.1 – Evolution of the best solution with the diversification operator level for the stap application.

A.3 Application benchmarks

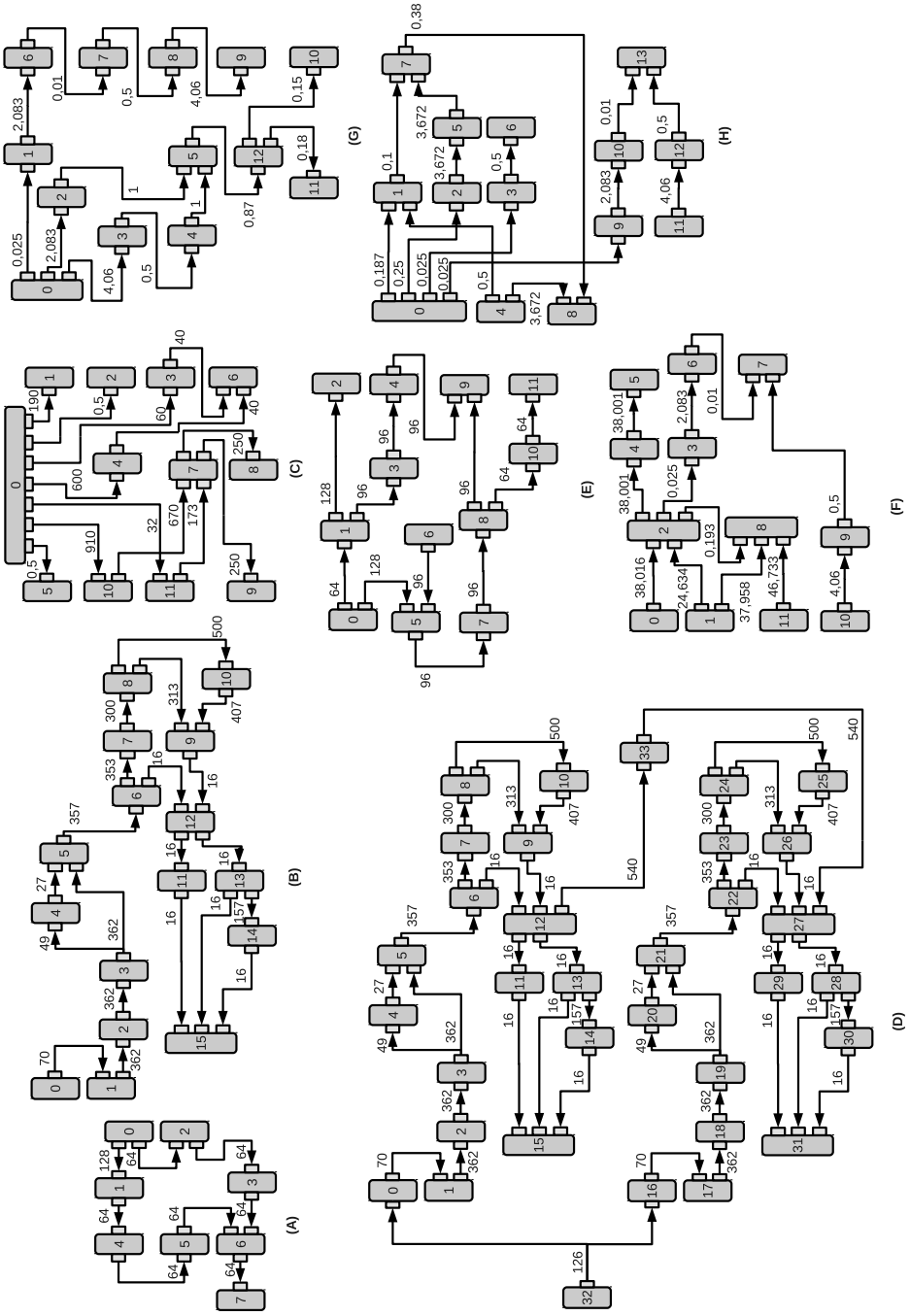


Figure A.2 – (A) PIP, (B) VOPD, (C) MPEG-4, (D) DVOPD, (E) MWD, (F) mp3enc mp3dec, (G) 263enc mp3dec, (H) 263dec mp3dec.

A.4 Benchmarks architecture model

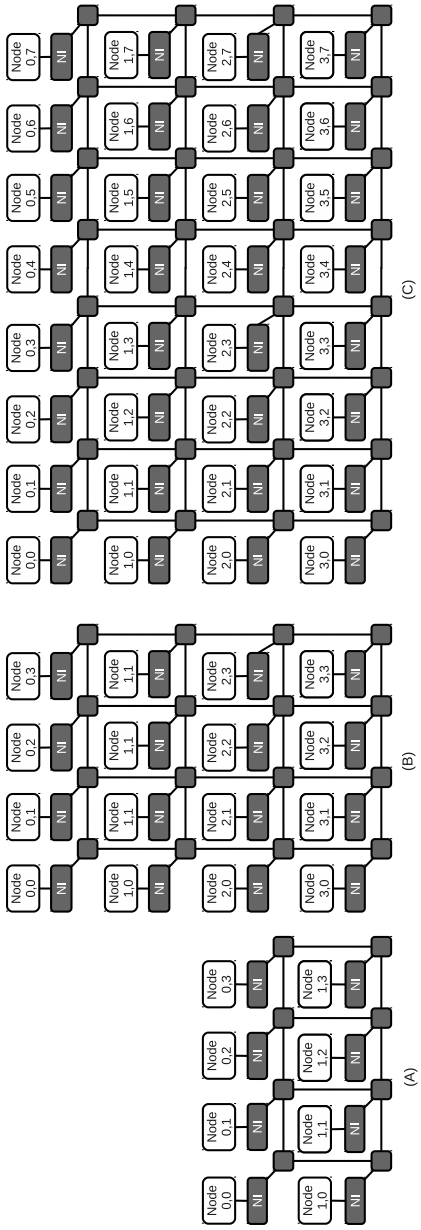


Figure A.3 – (A) PIP architecture model, (B) VOPD, MPEG-4, MWD, 263enc mp3dec, mp3enc mp3dec, 263dec mp3 dec architecture model, (C) DVOPD architecture model.

A.5 Justification of the Genetic algorithm parameters

The experiments done to choose the GA parameters have been done regarding the methodology proposed by [102]. Since the parameters choices have to be done in the worst case these experiments are ran for the stap application.

The parameters settings have been chosen from a set of trials demonstrating that the accuracy of the exploration was not further improved by an increase of parameters values.

A.5.1 Justification of the population size

To observe the evolution of the final solution quality with the population size, several tests were performed for 6 population size: 100, 500, 1000, 1500, 2000, 2500. The results obtained for the 6 population size are resumed into Table A.2. These experiments were done on the stap application.

The results show that under a population of 2000 the quality of final solution found by the GAFO is impacted, while a population composed of more than 2000 individuals does not lead to an important benefit.

In order to keep the execution time in a reasonable amount of time the size of population is limited to 2000.

Table A.2 – Error rate on the stap application for different population size.

Population size	Error rate %	Execution time (s)
100	25	59.8
500	17.4	72.8
1000	13.72	114.7
1500	12.2	223.8
2000	5.88	437.4
2500	5.9	682.2
3000	5.82	1173.4

A.5.2 Justification of the number of generation

To observe the behavior of the algorithm with the evolution of the number of generation, the algorithm runs with only the selection operator. The crossover and the mutation operators are not triggered so in all the cases the best individual is present into the initial population at the beginning. There is no possibility to find a better solution and the GA population will converge to this solution after a number of generations. The size of the initial population is set to 2000.

The measures (Figure A.4) show that the population needs more than 1500 generations to converge to the best solution. In order to be sure that the algorithm has time enough to converge to a near optimal solution the number of generation is set to 2000.

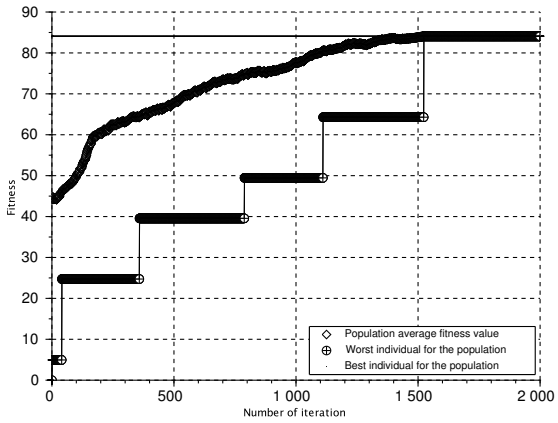


Figure A.4 – Convergence of a population composed of 2000 individuals

A.5.3 Justification of the crossover rate

In order to find the appropriate crossover rate, we measured the impact of the crossover rate on the final solution. The value of the crossover rate varies from 0 to 1 by step of 0.1. These experiments were performed with a population size of 2000 and a number of generation of 2000.

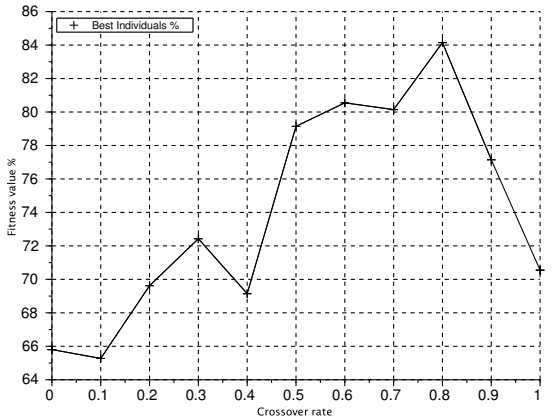


Figure A.5 – Impact of the crossover rate

The obtained results are depicted on Figure A.5. Although the curve is not monotonic, we can observe a general tendency of the algorithm. Indeed a crossover rate included between 0.6 and 0.8 gives the best results. Thus the value of the crossover is set at 0.8 since it is the value which gives the better results.

A.5.4 Justification of the mutation rate

In order to find the appropriate mutation rate, the same experiments as the one performed for the crossover rate are done. The size of the population is set to 2000, the number of generation is set to 2000 and the crossover rate to 0.8.

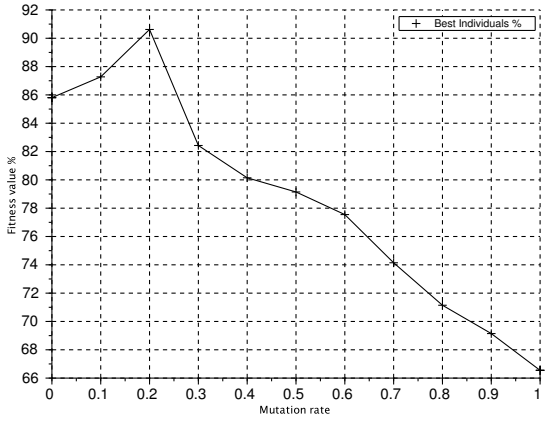


Figure A.6 – Impact of the mutation rate

The results are showed on Figure A.6. These results showed that the fitness of the individual are maximized for a mutation rate included between 0.0 and 0.3. Thus the mutation rate is set to 0.2 since it is the value which gives the better results.

A.5.5 Justification of the FO rate

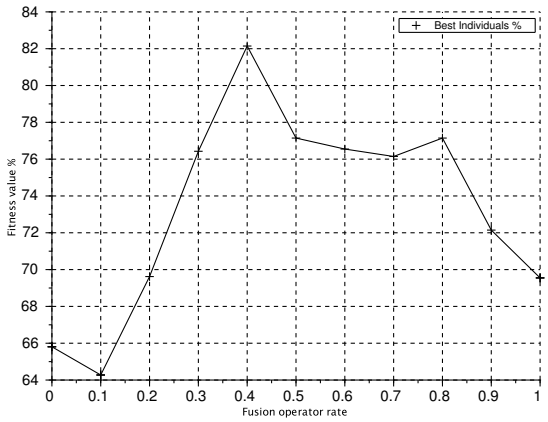


Figure A.7 – Impact of the fusion operator rate

As done for the crossover and mutation rate the same experiments are done for the fusion operator rate. The size of the population is set to 2000, the number of generation to 2000, the crossover rate to 0.8 and the mutation rate to 0.2.

The results depicted in Figure A.7 shows that the fitness of the individual is maximized between 0.3 and 0.8. Thus the FO rate is set to 0.5 since it is the value which gives the better results.

A.6 Application task graph

The fully detailed application task graph for the chirp application, the jpeg application and the stap application are respectively shown Figure A.8, A.9 and A.10.

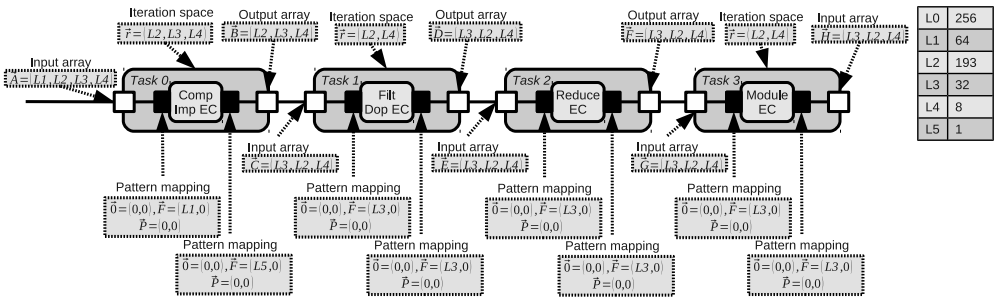


Figure A.8 – Chirp application task graph with loops details.

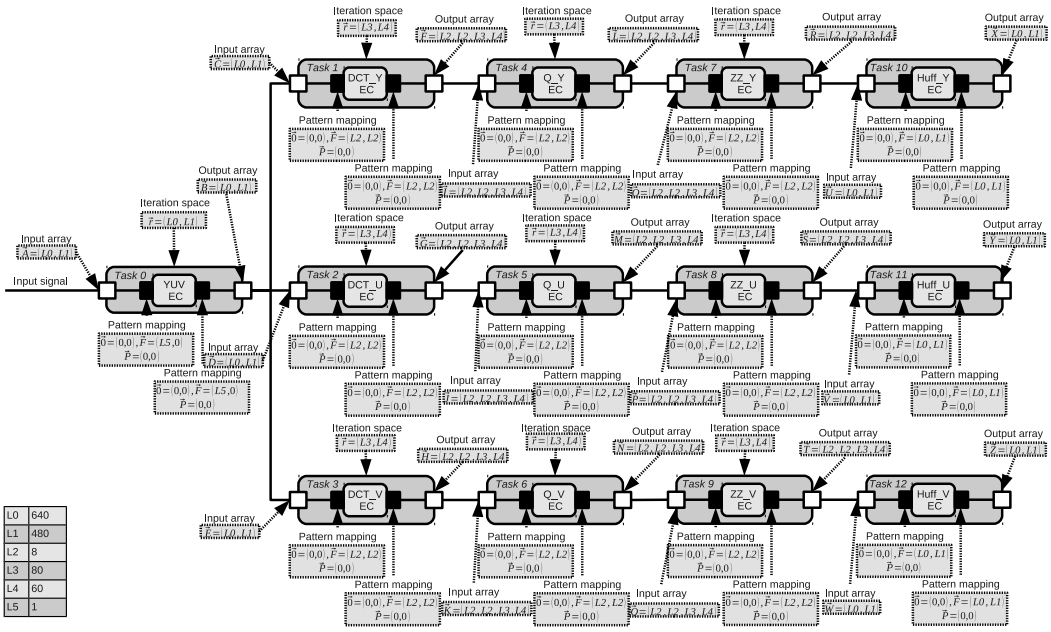


Figure A.9 – Jpeg application task graph with loops details.

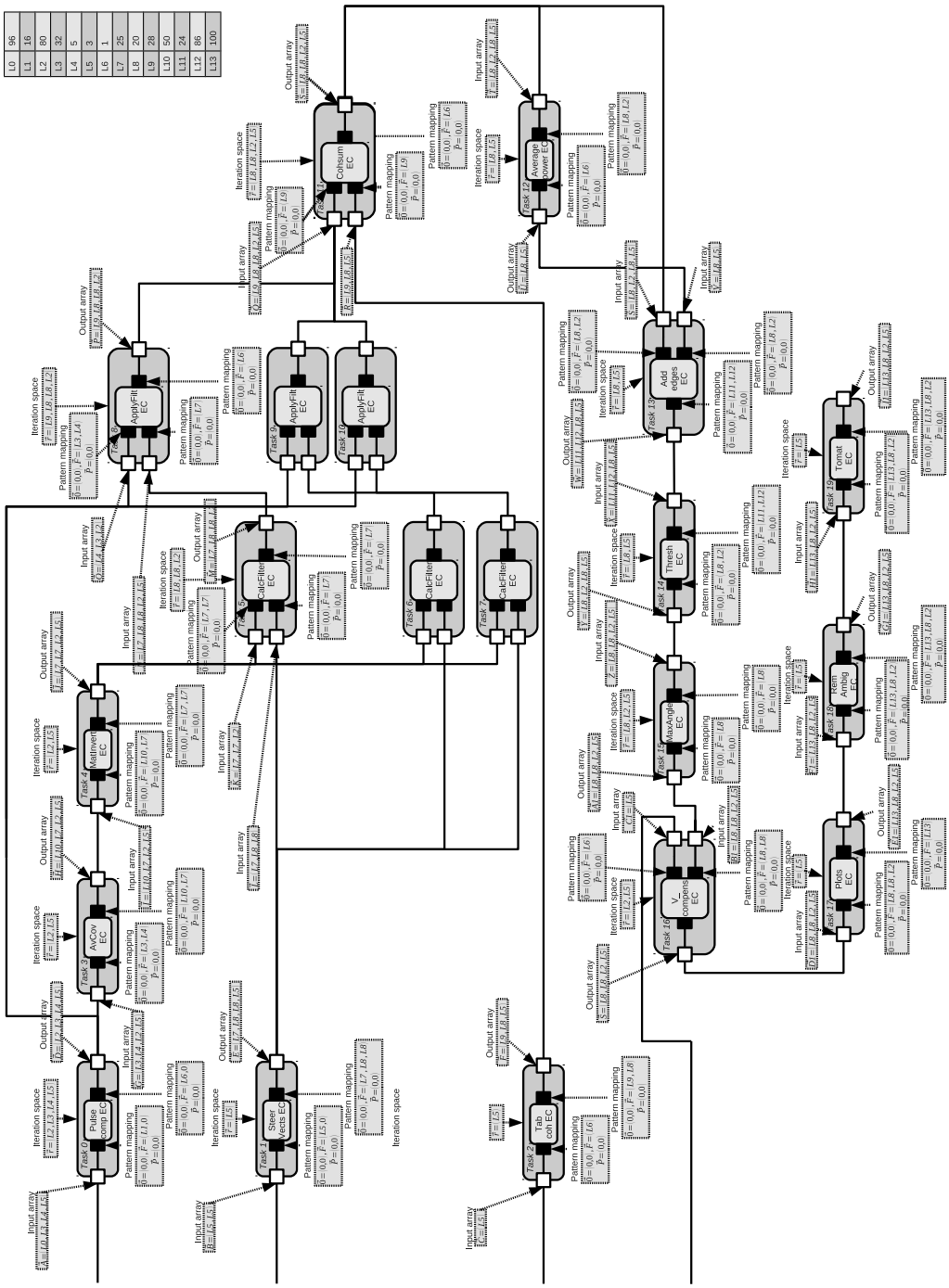


Figure A.10 – Stap application task graph with loops details.

A.7 Data placement representation

A.7.1 Example of data placement obtained for the chirp application

The data placement shown Figure A.11 is the one obtained when the chirp application is mapped onto the architecture depicted in Figure 3.21.A with a memory size of 256 kB. In this experiments all the tasks are mapped onto the node 0,0 (see section 3.5).

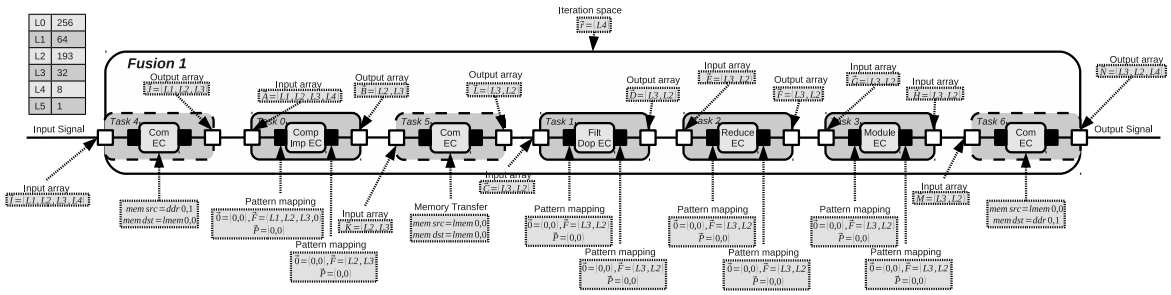


Figure A.11 – Chirp data placement obtained following the exploration done with the GAFO.

A.7.2 Example of data placement obtained for the jpeg application

The data placement shown Figure A.13 is the one obtained for the jpeg application. The architecture model is shown Figure 3.21 and the mapping models given in input is depicted Figure A.12. The size of the on-chip memory is of 720 kB.

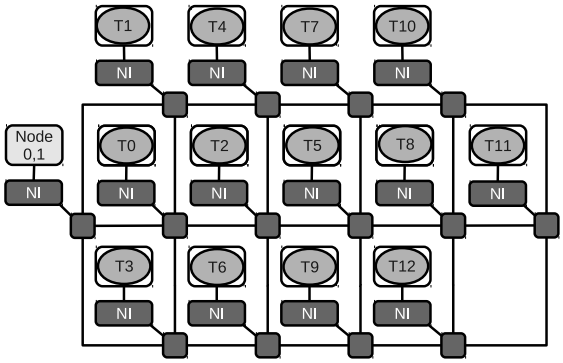


Figure A.12 – Jpeg input mapping used by the GAFO.

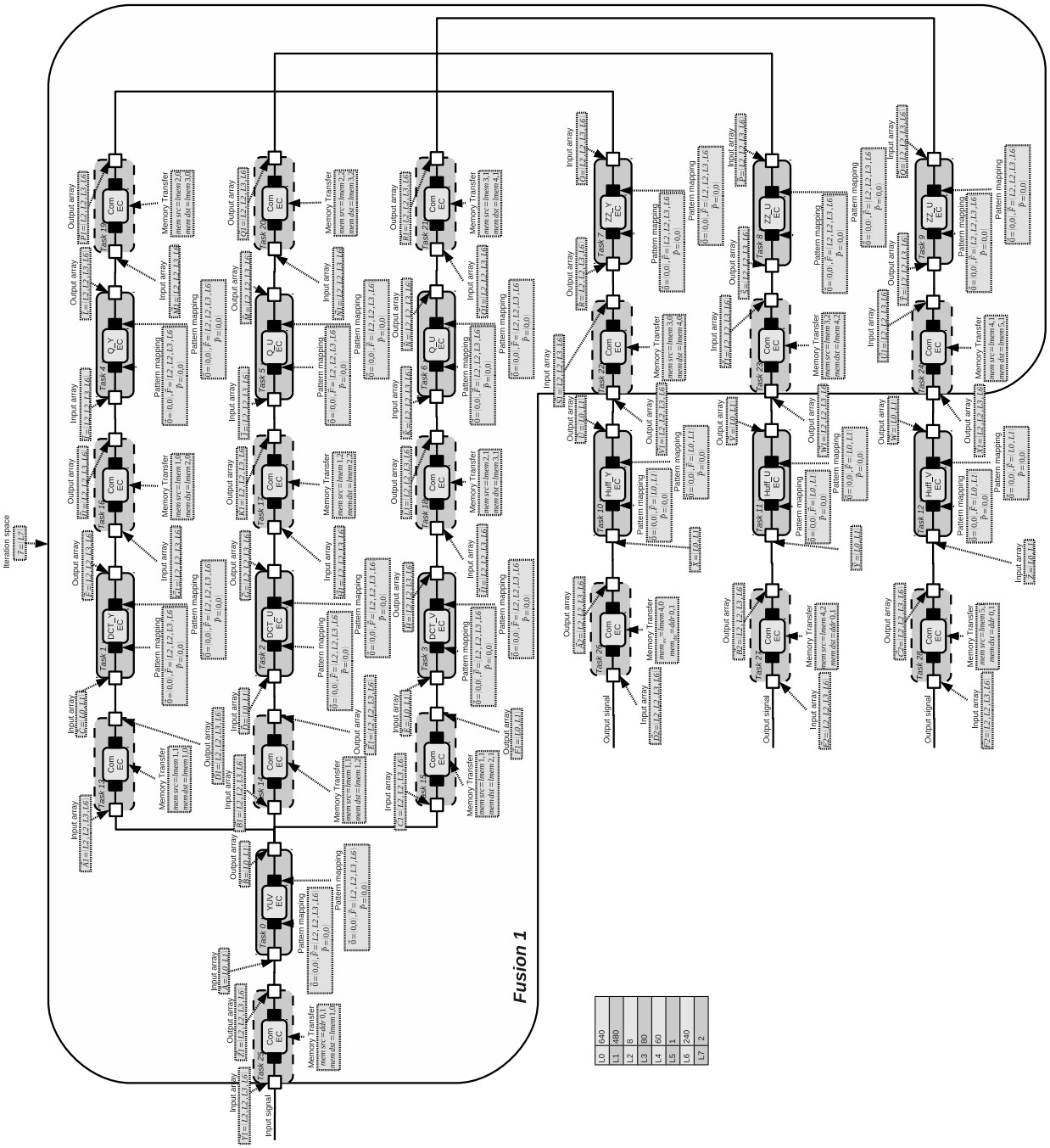


Figure A.13 – Jpeg data placement obtained following the exploration done with the GAFO.

A.7.3 Example of data placement obtained for the stap application

The data placement shown Figure A.15 is the one obtained for the stap application. The architecture and mapping models given in inputs are the one depicted in Figure 2.9 and in Figure A.14. The size of the on-chip memory is of 2 MB per cluster.

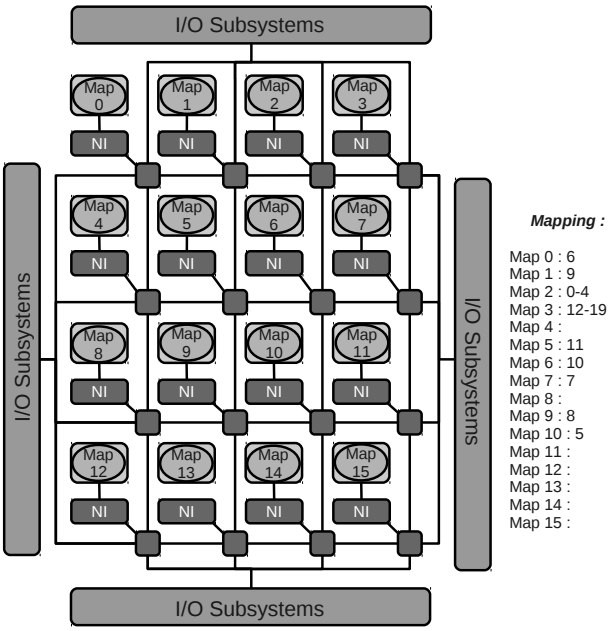


Figure A.14 – Stap input mapping used for the GAFO.

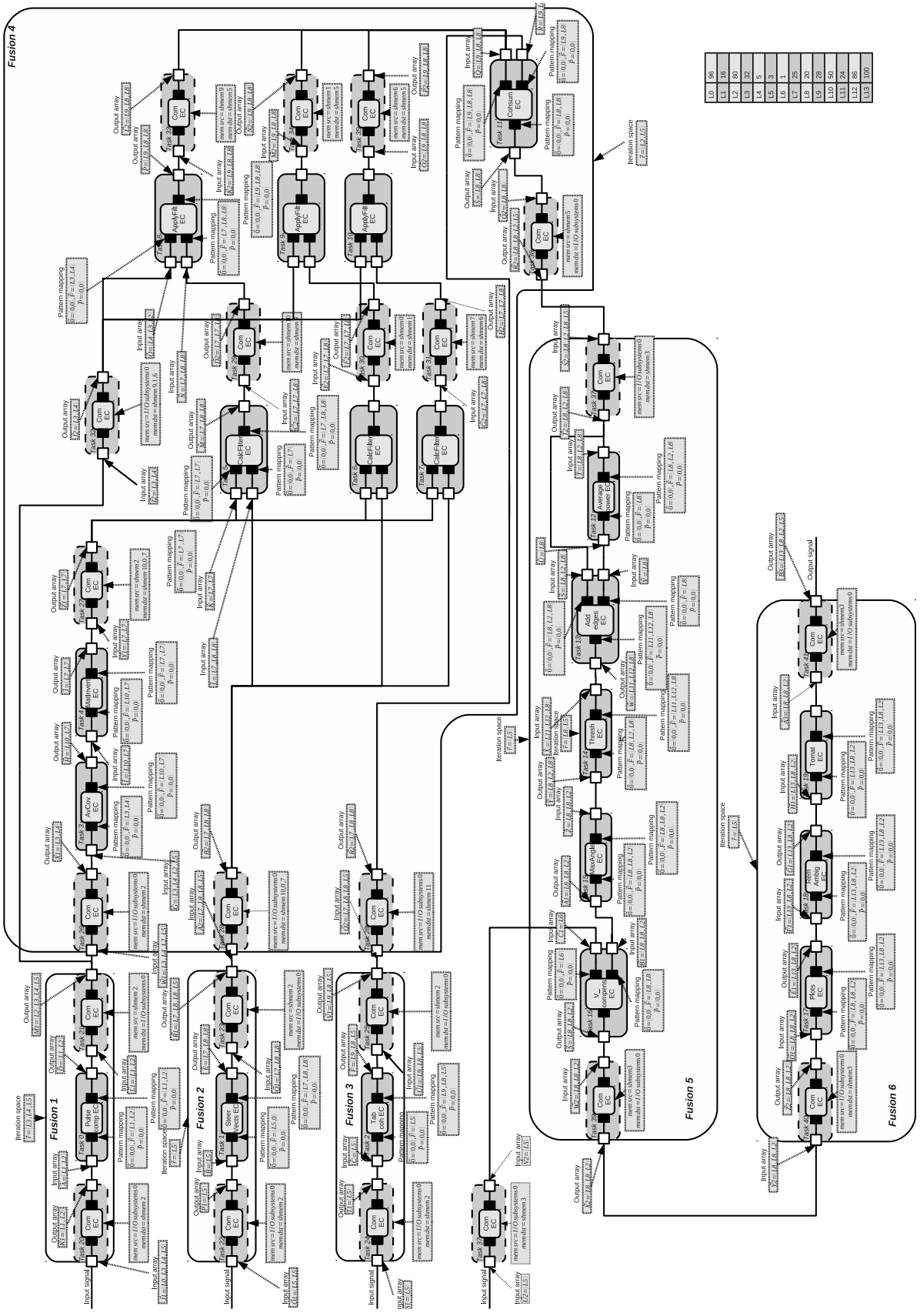


Figure A.15 – Stap data placement obtained following the exploration done with the GAFO.

A.8 TSRS-GAFO

A.8.1 Chirp

A.8.1.1 Mapping

The mappings obtained by the TSRS-GAFO for the architecture models depicted in Figure 3.22.A are shown Figure A.16.

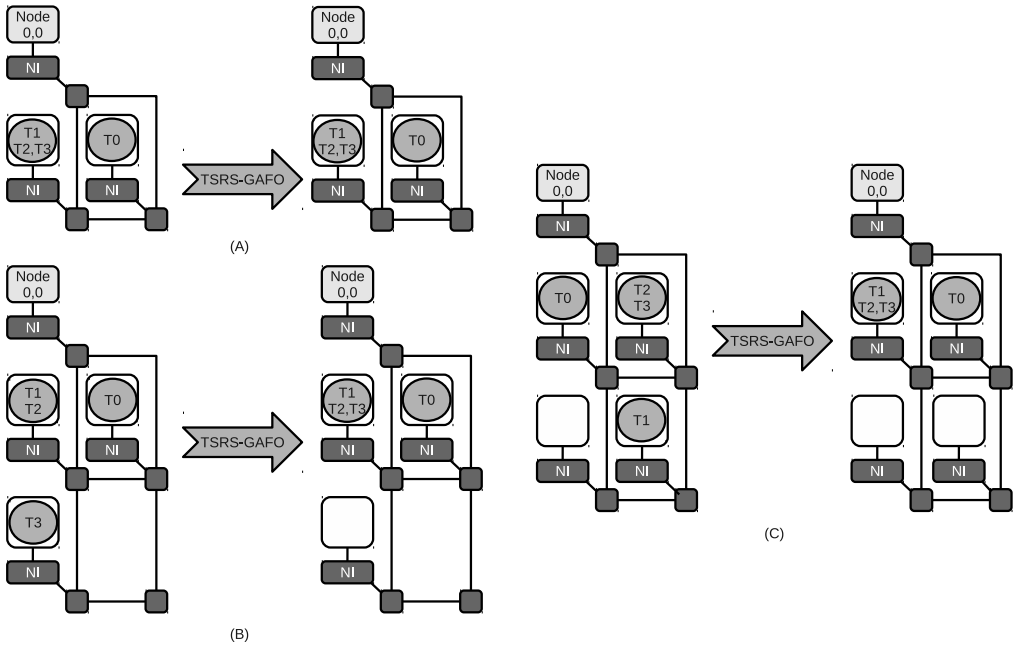


Figure A.16 – Initial and Final mapping obtained for the chirp application by the TSRS-GAFO for the three test case architectures.

A.8.1.2 Data placement

The data placements obtained for the three test case architectures are the same as the one depicted in Figure A.11.

A.8.2 Jpeg

A.8.2.1 Mapping

The mappings obtained by the TSRS-GAFO for the architecture models depicted in Figure 3.22.D and E and in Figure 3.21.B are shown Figure A.17.

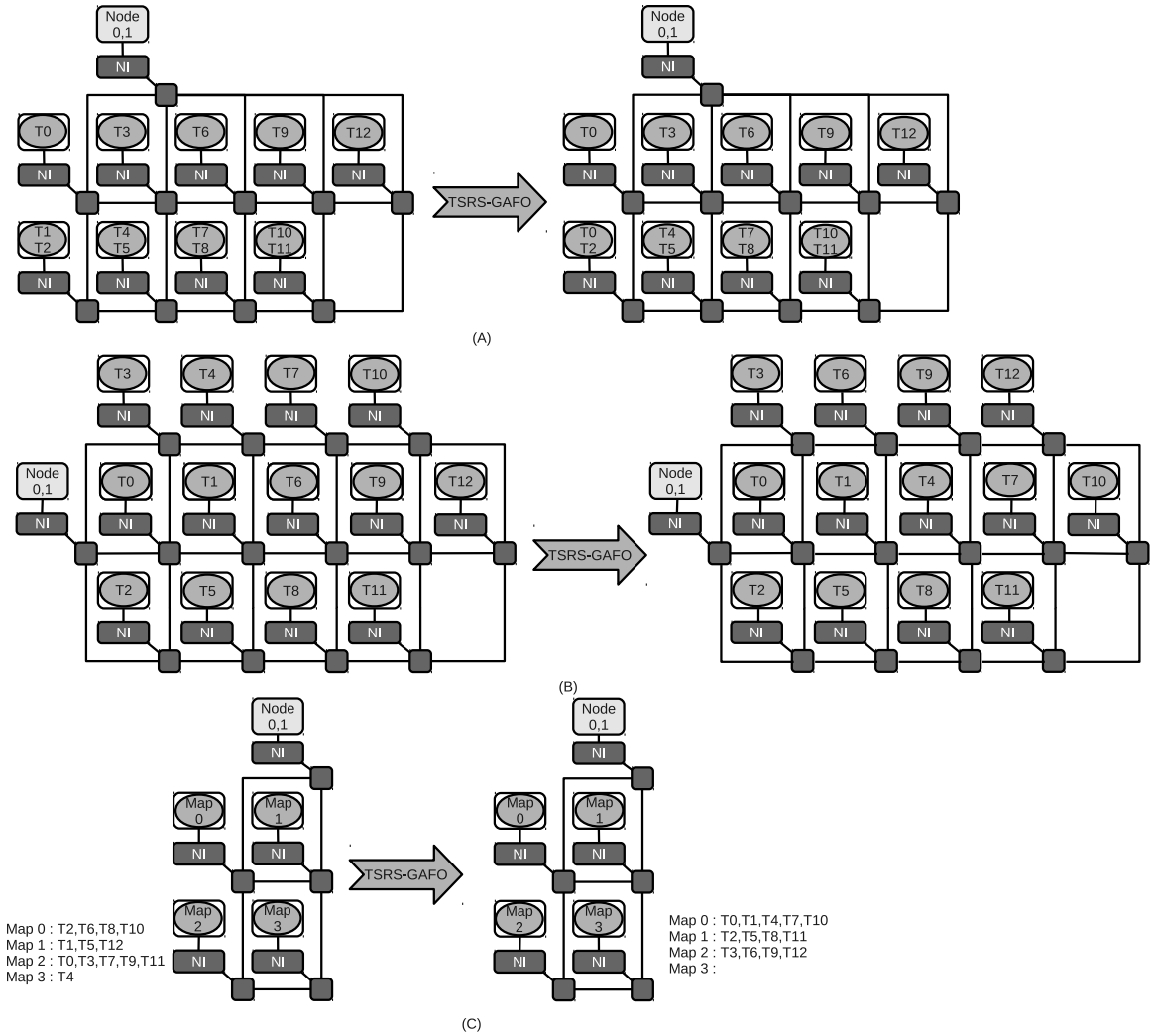


Figure A.17 – Initial and Final mapping obtained for the chirp application by the TSRS-GAFO for the three test case architectures.

A.8.2.2 Data placement

The data placement obtained for the mappings shown Figure A.17.A and C is the one given Figure A.18. The data placement obtained for the mapping depicted in Figure A.17.B is the same as the one given Figure A.13.

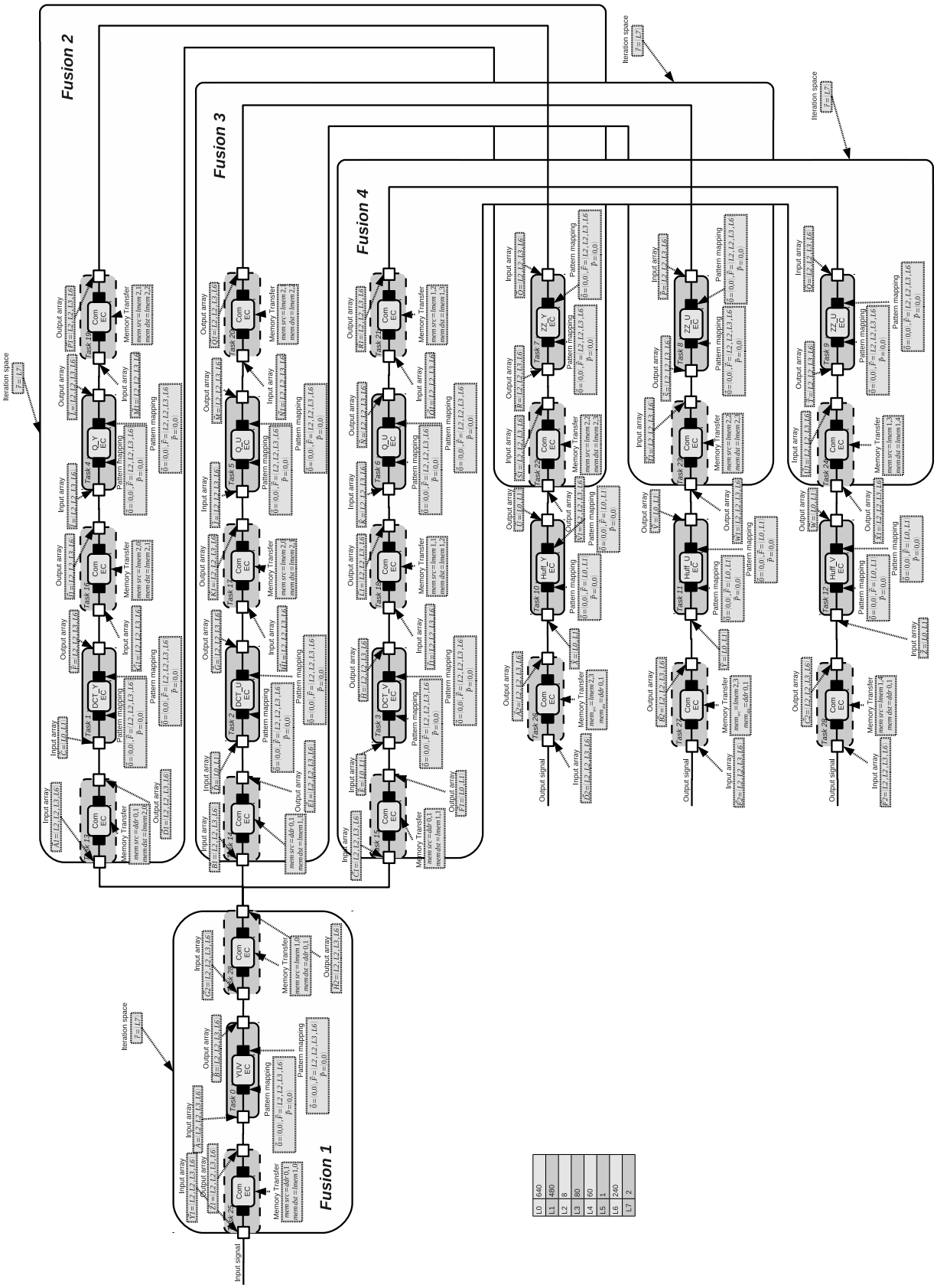


Figure A.18 – Jpeg data placement obtained following the exploration done with the TSRS-GAFO for the mapping shown Figure A.17 A and C.

A.8.3 Stap

A.8.3.1 Mapping

The mapping obtained by the TSRS-GAFO for the MPPA architecture depicted in Figure 2.9 is the one shown Figure A.19.

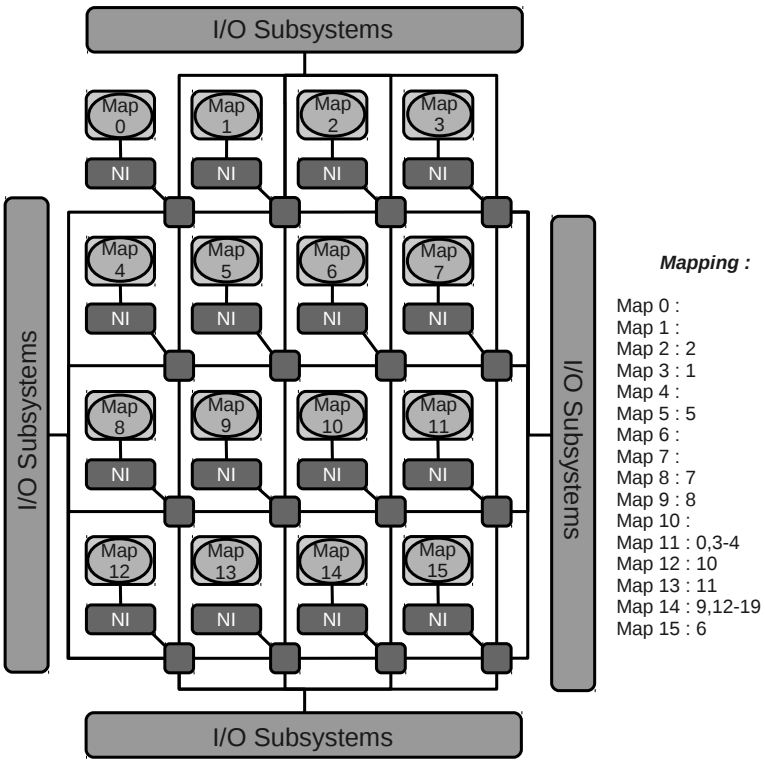


Figure A.19 – Initial and Final mapping obtained for the stap application by the TSRS-GAFO.

A.8.3.2 Data placement

The data placement obtained for the mapping depicted Table A.19 is the one given Figure A.20.

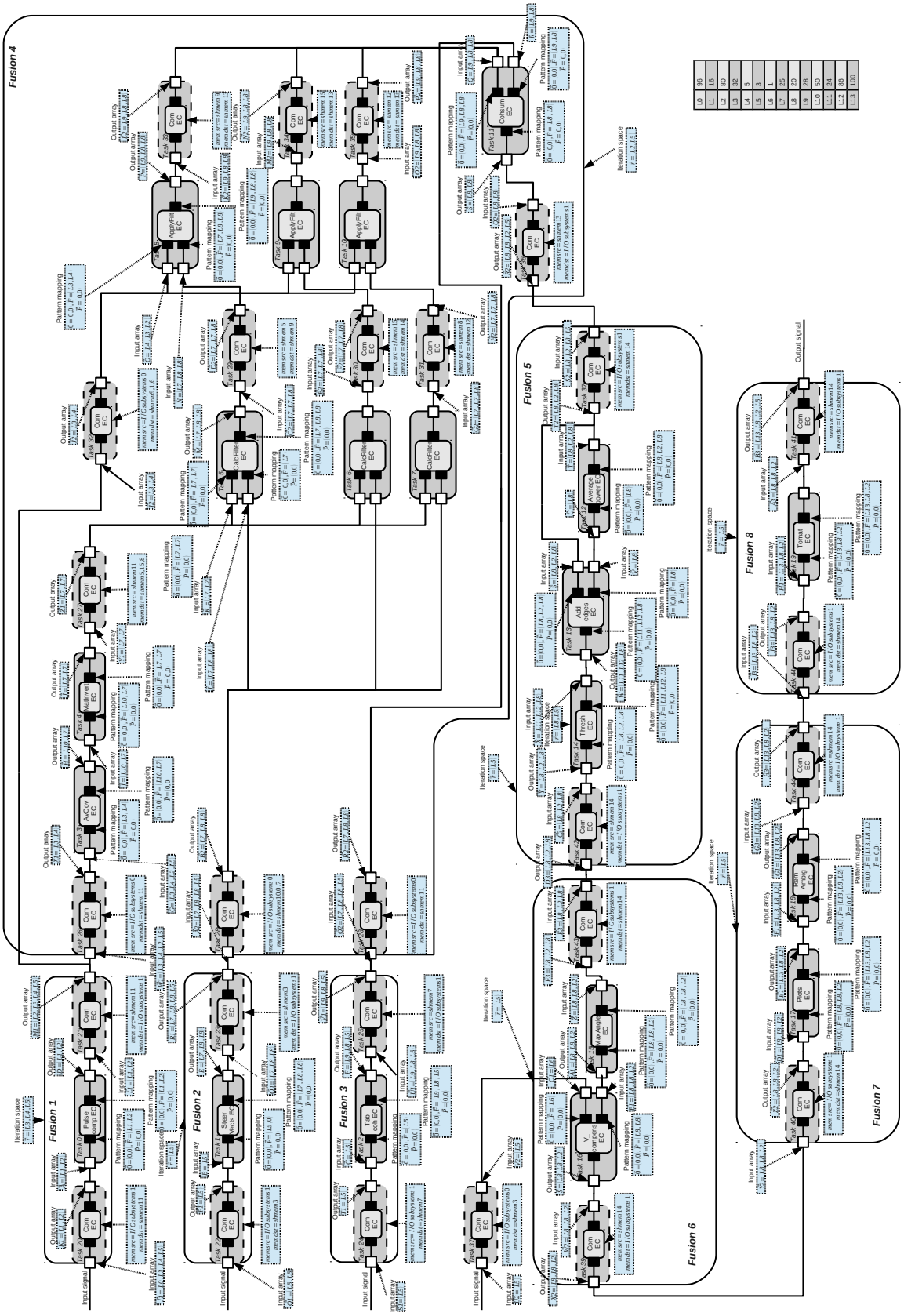


Figure A.20 – Stap data placement obtained following the exploration done with the TSRS-GAFO for the mapping shown Figure A.19

B

Annex B

Contents

B.1 Accelerator Interface

B.1.1 Vehicle registration plate detection synchronization scheme and execution patterns

The complete synchronization scheme and executions patterns for the vehicle registration plate detection application are depicted in Figure B.1, B.2, B.3 and B.4.

The first step needed to run the application is to program the AI of the platform. To that end the GPP node 1 is used to program the AI 0, 1 and 2. The GPP node 2 is used to program the AI 3, 4, and 5.

Once the AI programmed, the data coming from the I/O are fetched by the AI 0 and 3 and by the GPP 2. Since the accelerator 0 and 3 are data-flow one, the task 0 and 2 are fired based on the input data. As soon as the first results become available the AI 0 and 3 automatically transfer the results and synchronize based on the data with the AI 1 and 4 respectively.

Since the accelerators 1 and 4 are micro-programmed accelerators the AI 1 and 4 send a specific synchronization message to inform the accelerators that the data are present into the local memory. These synchronization events implies the firing of the task 1 and 3.

Once done the accelerators 1 and 4 send a synchronization event to their respective AI, to inform that the tasks executions are done. Based on this synchronization event the AI 1 and 4 respectively send the data and a specific synchronization event to the GPP 1. The GPP 1 then stop its current processing and the task 4 is fired. At the end of the task 4 execution the GPP 0 transfer the results to the AI 2. The GPP 1 is then free to do other processings in parallel.

The AI 2 receives the incoming data and fire the task 5 and 6. Once the first results become available the data are sent to the AI 5 to fire the task 7.

Based on the input data the task 7 is started and at the end of the task execution the output data are transferred with a specific synchronization event to the GPP node 2. The GPP 2 which was doing other processing in parallel then stop its current execution to run task 8 and finish the application execution.

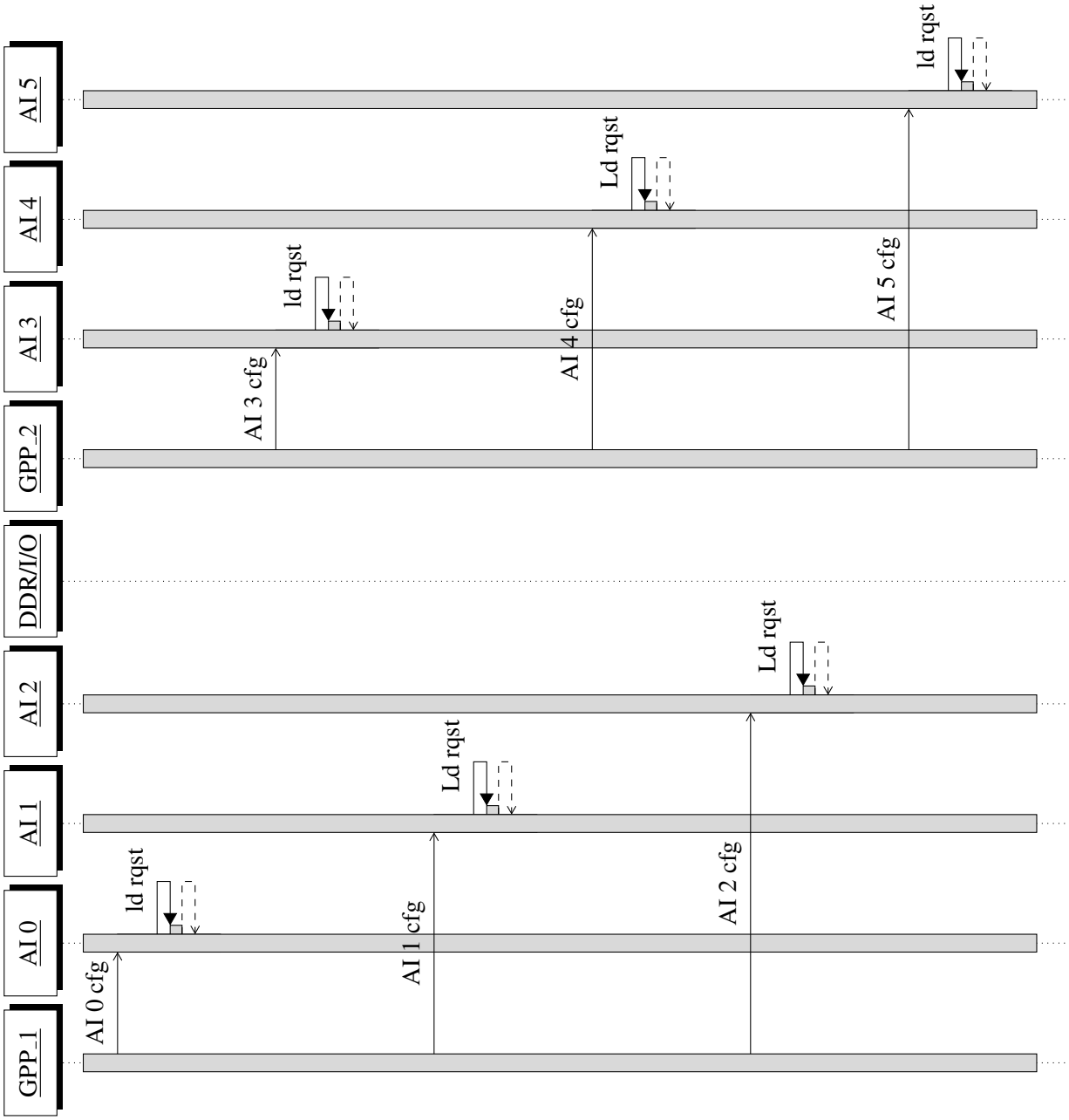


Figure B.1 – Vehicle registration plate synchronization schemes and execution patterns.

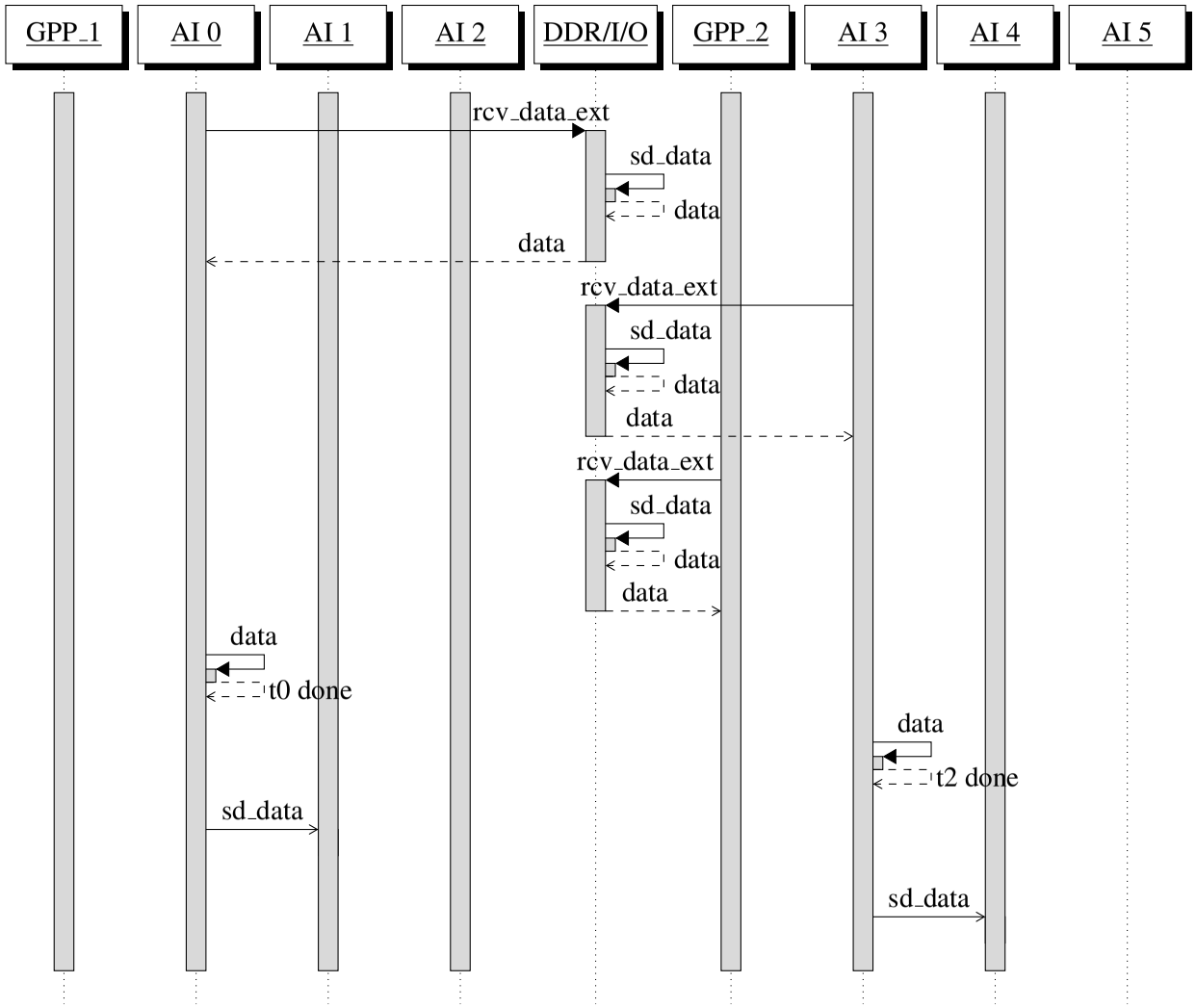


Figure B.2 – Vehicle registration plate synchronization schemes and execution patterns.

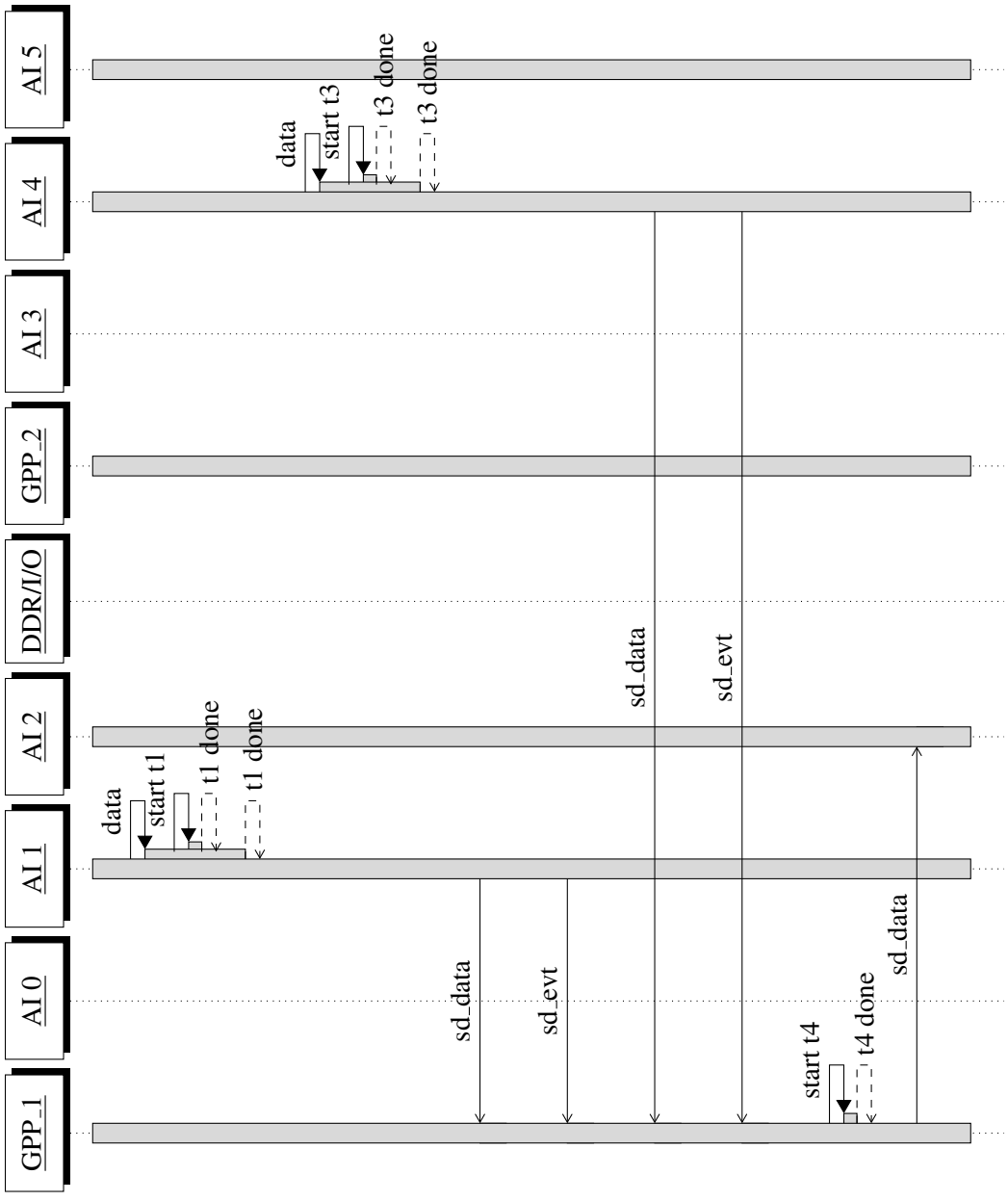


Figure B.3 – Vehicle registration plate synchronization schemes and execution patterns.

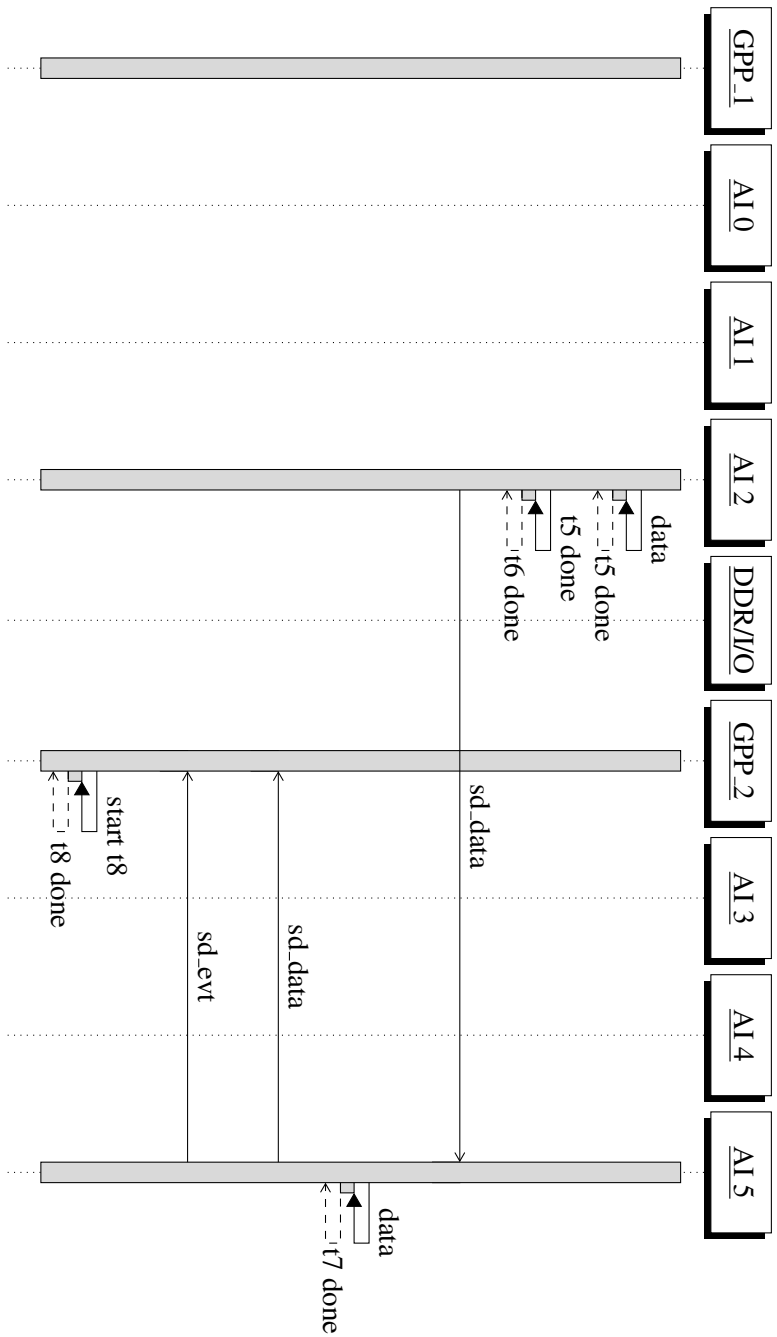


Figure B.4 – Vehicle registration plate synchronization schemes and execution patterns.

B.2 Cluster based architecture a memory abstraction

B.2.1 2D-FFT application mapping

B.2.2 Cluster based architecture mapping

The mapping used with the Clus_2 architecture for the 2D-FFT application is given Figure B.5.

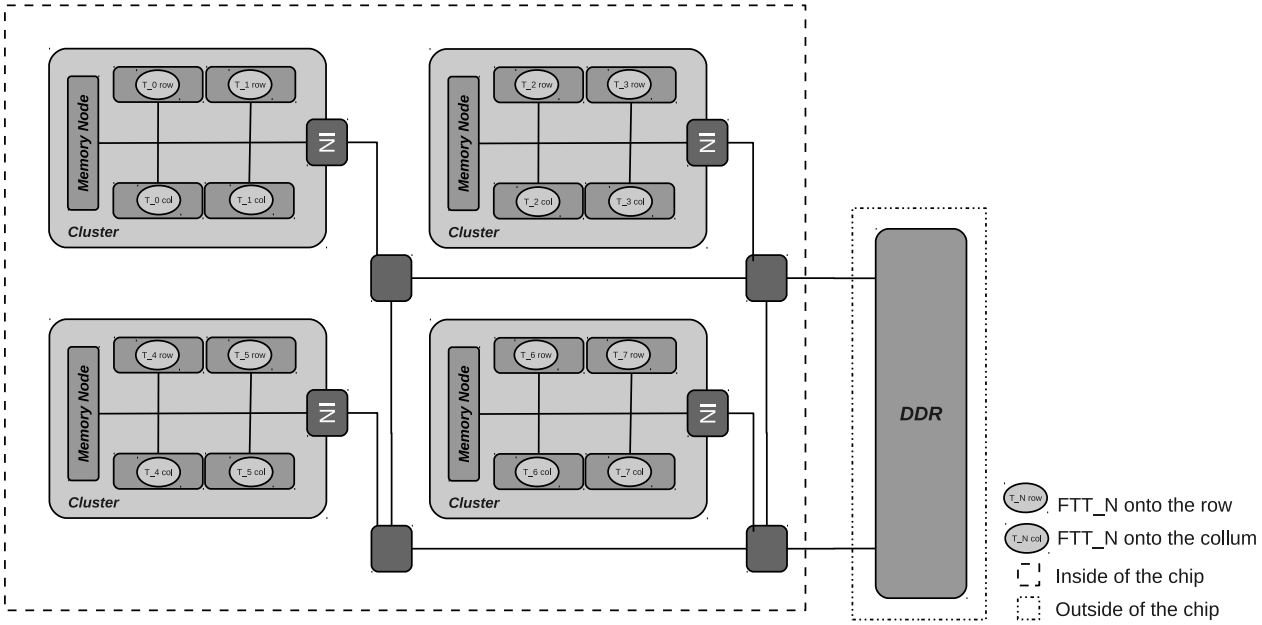


Figure B.5 – 2D-FFT application mapping on the clus_2 architecture

B.2.3 Flat architecture mapping

The mapping used with the Flat_2 architecture for the 2D-FFT application is given Figure B.6.

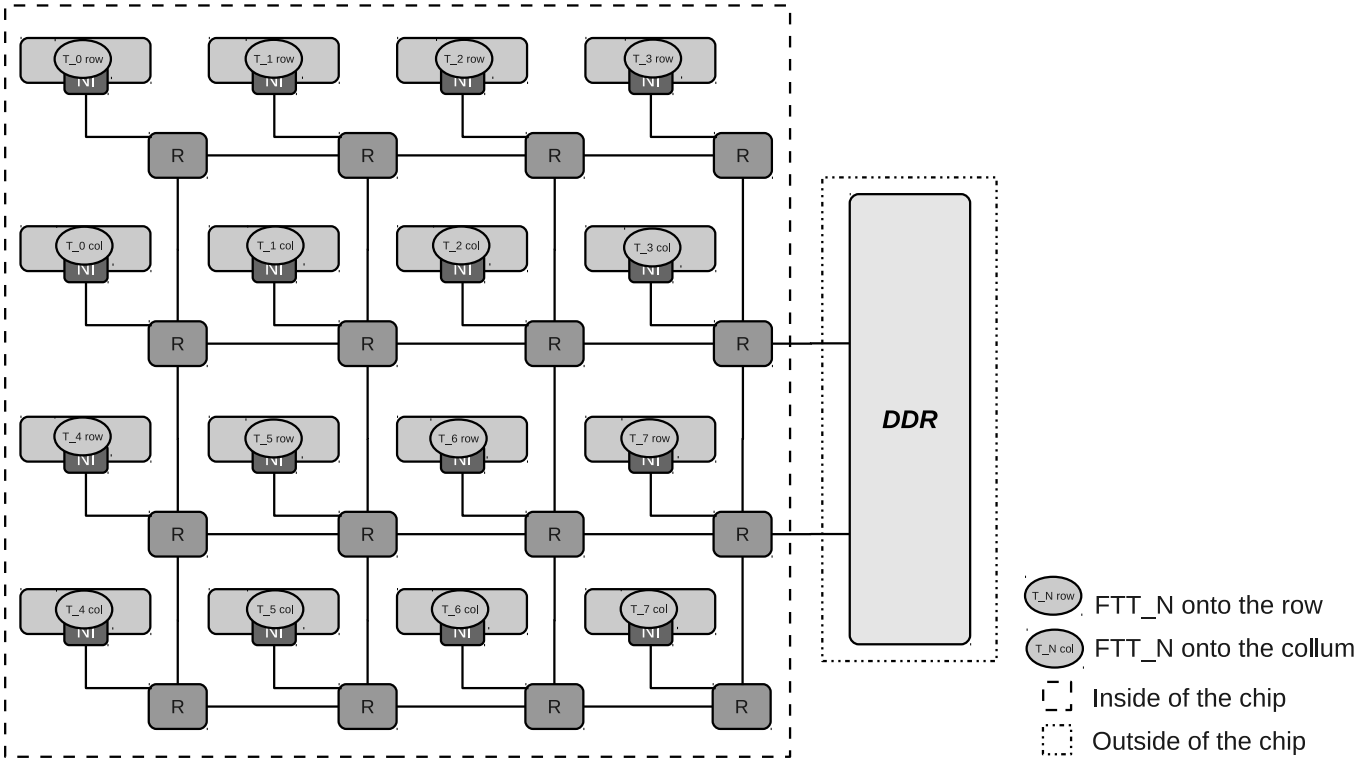


Figure B.6 – 2D-FFT application mapping on the Flat_2 architecture

B.2.4 Ter@ops architecture mapping

The mapping used with the Ter@ops architecture for the 2D-FFT application is given Figure B.7.

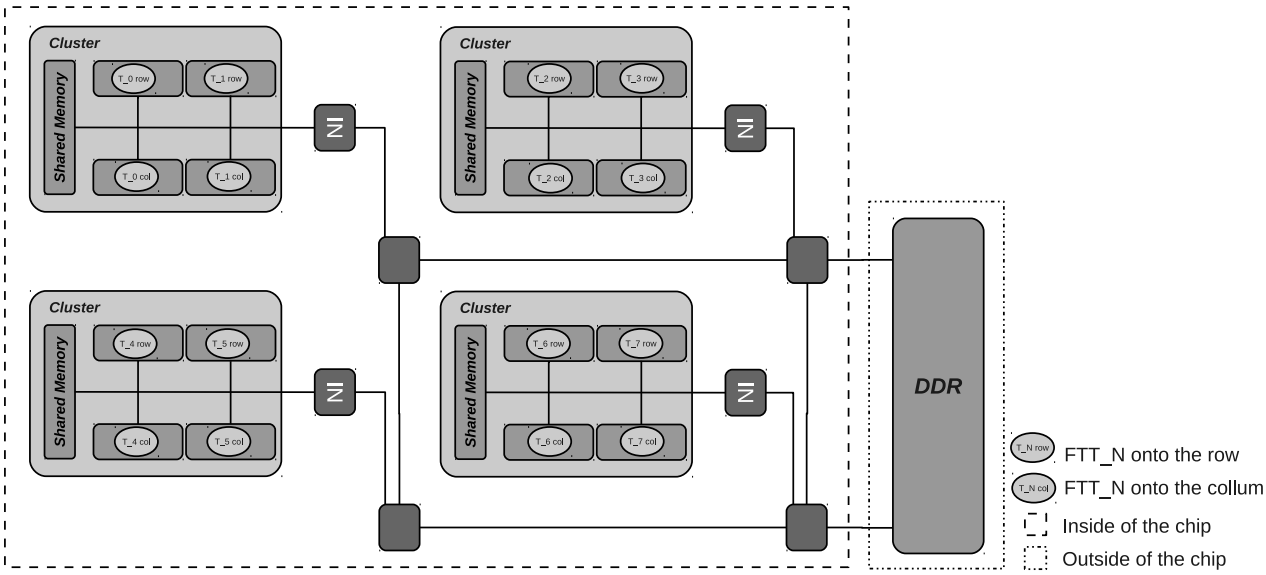


Figure B.7 – 2D-FFT application mapping on the Ter@ops architecture

B.2.5 Matrix multiplication application mapping

B.2.6 Cluster based architecture mapping

The mapping used with the Clus_2 architecture for the matrix multiplication application is given Figure B.8.

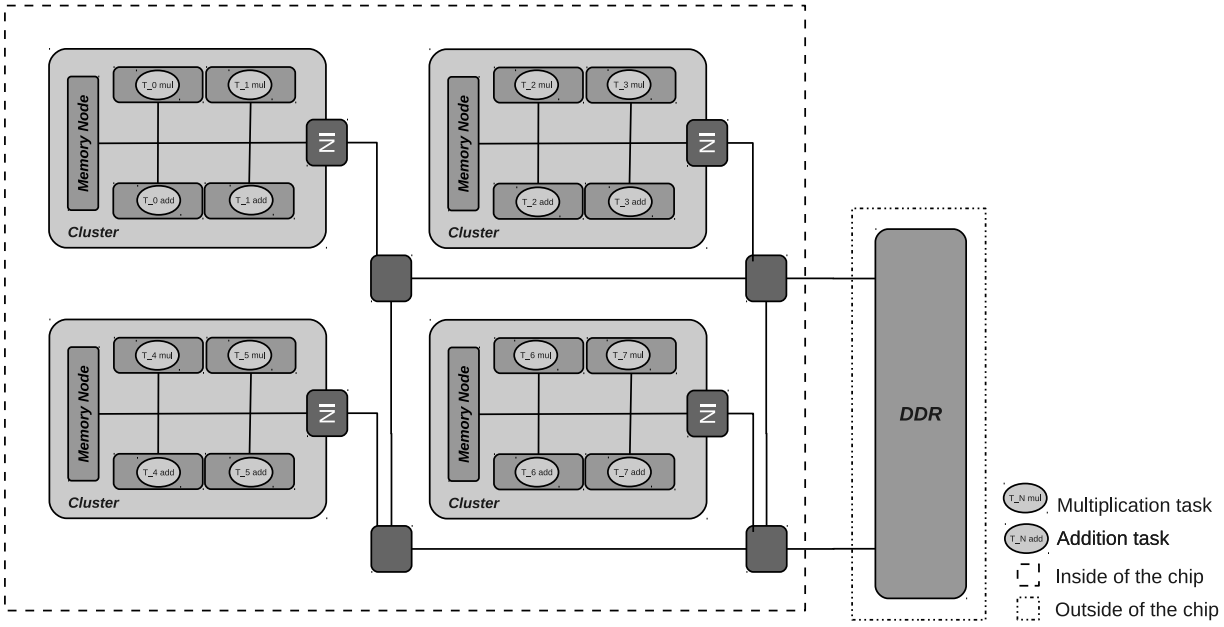


Figure B.8 – Matrix multiplication application mapping on the clus_2 architecture

B.2.7 Flat architecture mapping

The mapping used with the Flat_2 architecture for the matrix multiplication application is given Figure B.9.

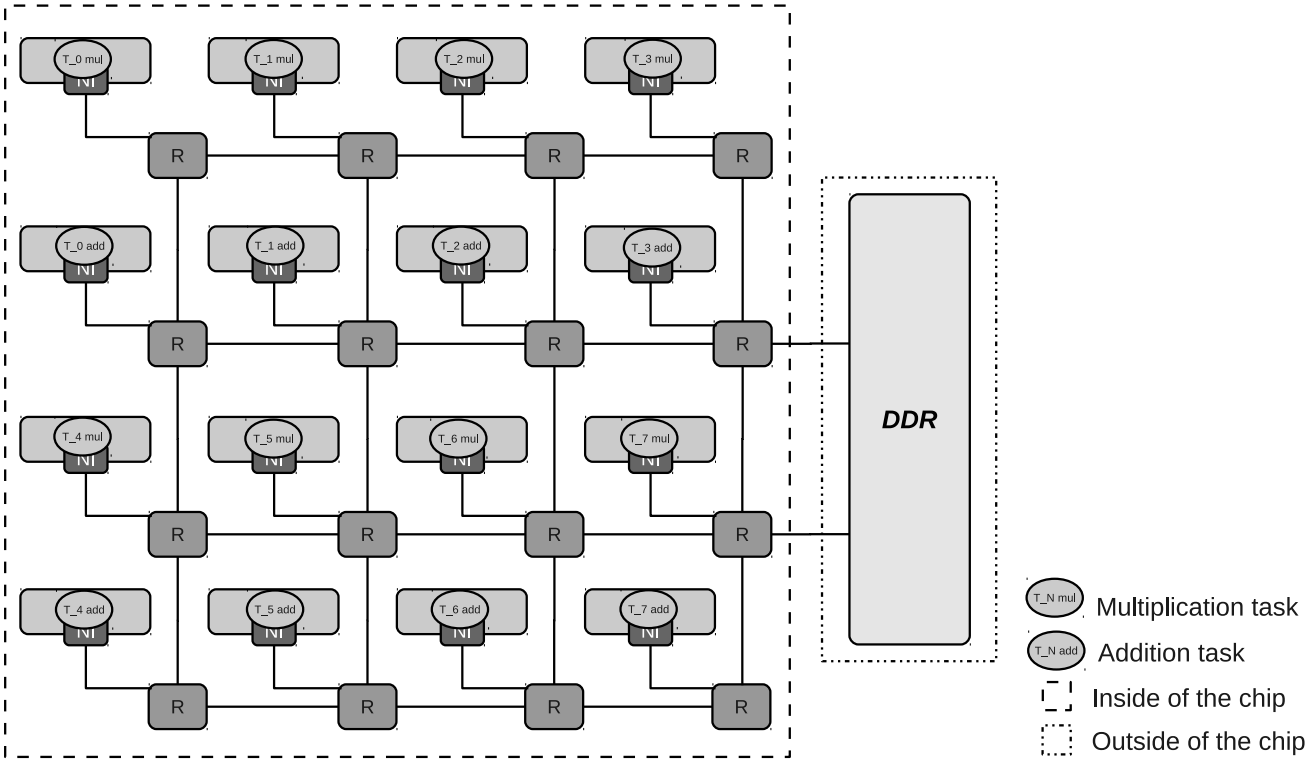


Figure B.9 – Matrix multiplication application mapping on the Flat_2 architecture

B.2.8 Ter@ops architecture mapping

The mapping used with the Ter@ops architecture for the matrix multiplication application is given Figure B.10.

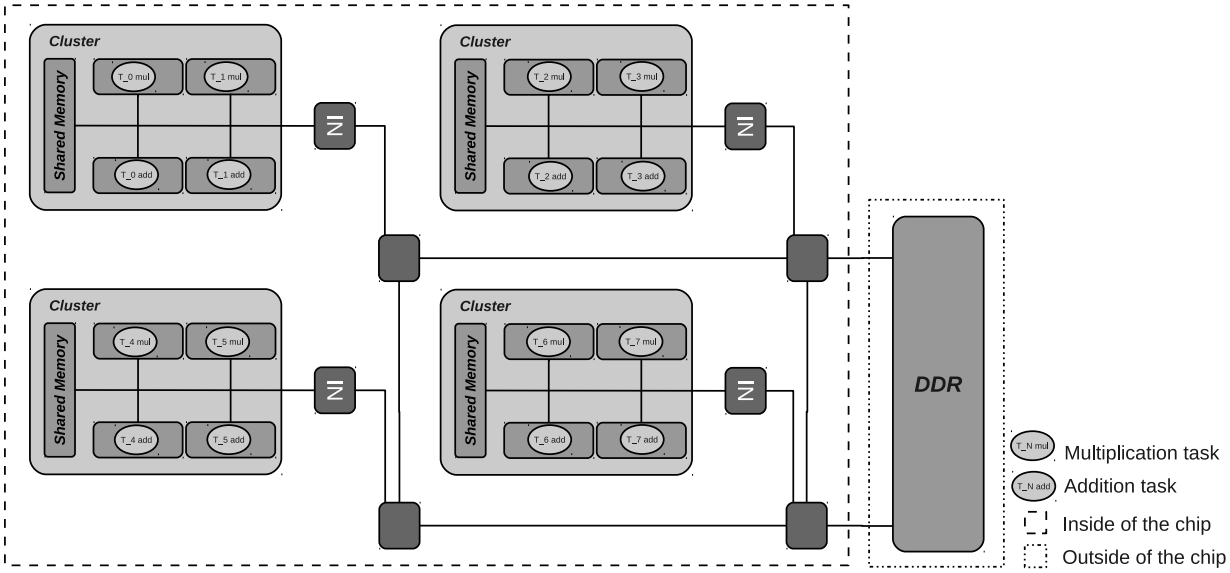


Figure B.10 – Matrix multiplication application mapping on the Ter@ops architecture

B.2.9 Cluster based architecture inefficient mapping

An inefficient mapping of the matrix multiplication application on the Clus_2 architecture is given Figure B.11.

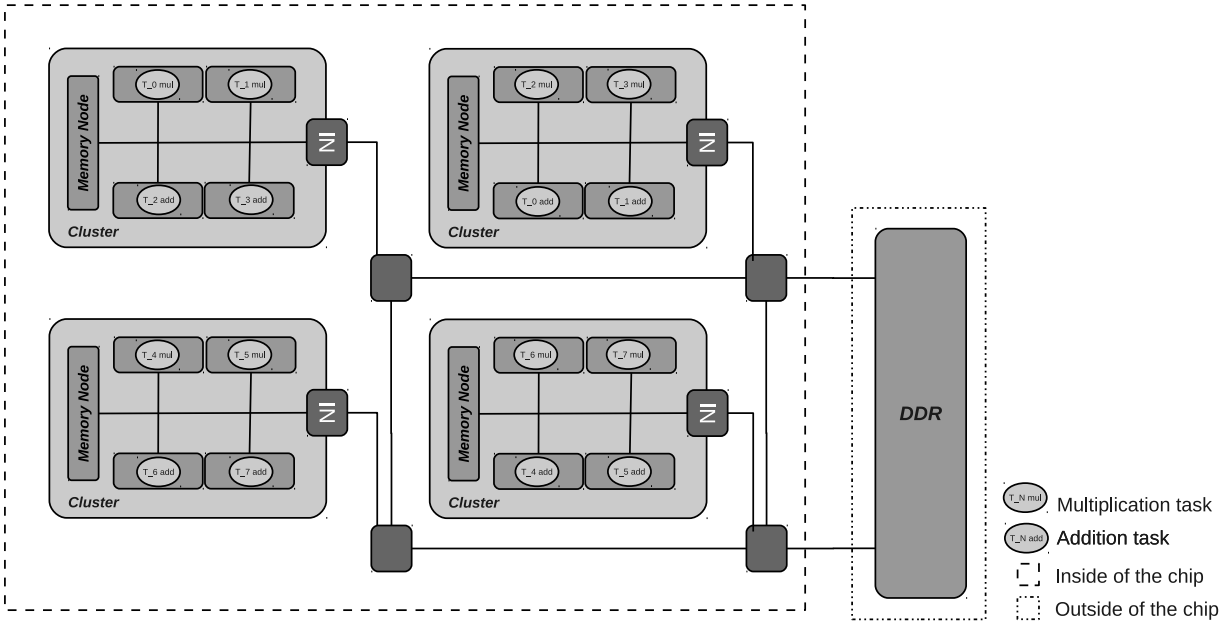


Figure B.11 – Inefficient matrix multiplication application mapping on the clus_2 architecture

Bibliography

[1] Adapteva, http://www.adapteva.com/. 16

[2] Arm, http://www.arm.com/. 14, 20

[3] Azul systems, http://www.azulsystems.com/products/vega/processor. 14

[4] Cadence c to silicon, http://www.cadence.com/. 33

[5] Clearspeed, http://www.clearspeed.com/. 18

[6] Cofluent studio, http://www.cofluentdesign.com/index.php?page=home. 15, 28, 29

[7] Cool, http://ls12-www.cs.tu-dortmund.de/daes/de/forschung/hwsw-co-design/cool.html. 34

[8] Flextiles board, http://www.flextiles.biz/index.php. 93

[9] FlexTiles FP7 project, http://flextiles.eu. 82

[10] Freescale, http://www.freescale.com/. 16

[11] Gaut, http://hls-labsticc.univ-ubs.fr/. 34

[12] Greensoc, http://www.greensocs.com/. 31

[13] Intel scc, http://www.intel.com/content/www/us/en/research/intel-labs-single-chip-cloud-computer.html. 16, 98

[14] Jacquard computing, http://www.jacquardcomputing.com/roccc/. 34

[15] Metropolis, http://embedded.eecs.berkeley.edu/metropolis/metamodel.html. 26

[16] Mpi, http://www.mpi-forum.org/docs/. 108

[17] Ocp, http://www.ocpip.org/. 87

[18] Ovp, http://www.ovpworld.org/. 32, 132

[19] Pips, http://www.cri.ensmp.fr/pips/. 132

[20] Plurality, http://www.plurality.com/. 18

[21] Powerpc 7748, http://www.msc-ge.com/en/921-www/version/default/part/attachmentdata/data/pc7448.pdf. 68, 73

[22] Soclib, http://www. soclib. fr. 31

[23] Space codesign, http://www.spacecodesign.com/. 15, 30

[24] Stretch incorporated, http://www.stretchinc.com/. 20

[25] Synflow, https://www.synflow.com/. 34

[26] Teraops, http://www.systematic-paris-region.org/en/projets/teraops. 85, 98, 99

[27] Texas instrument, http://www.ti.com/. 20

[28] Tilera, http://www.tilera.com/. 16, 85

[29] Xilinx virtex 6, http://www.xilinx.com/products/silicon-devices/fpga/virtex-6/. 93

[30] Enrique Alba and Marco Tomassini. Parallelism and evolutionary algorithms. *Evolutionary Computation*, 6(5):443–462, 2002. 122

[31] Najla Alfaraj, Junjie Zhang, Yang Xu, and H Jonathan Chao. Hope: Hotspot congestion control for clos network on chip. In *International Symposium on Networks on Chip (NoCS)*, pages 17–24, 2011. 121

[32] VSI Alliance, On-Chip Bus Development Working Group, et al. Virtual component interface standard version 2 (ocb 2 2.0). Technical report, Alliance, 2001. 31

[33] Erik Anderson, Jason Agron, Wesley Peck, Jim Stevens, Fabrice Baijot, Ed Komp, Ron Sass, and David Andrews. Enabling a uniform programming model across the software/hardware boundary. In *Field-Programmable Custom Computing Machines (FCCM)*, pages 89–98, 2006. 87

[34] Joel Zvi Apisdorf and Sam Brandon Sandbote. System and method for instruction-level parallelism in a programmable multiple network processor environment, September 27 2005. US Patent 6,950,927. 64

[35] Tero Arpinen, Tapio Koskinen, Erno Salminen, Timo D Hämäläinen, and Marko Hännikäinen. Evaluating uml2 modeling of ip-xact objects for automatic mp-soc integration onto fpga. In *Design Automation and Test in Europe (DATE)*, pages 244–249, 2009. 44, 87, 132

[36] Giuseppe Ascia, Vincenzo Catania, and Maurizio Palesi. A multi-objective genetic approach to mapping problem on network-on-chip. *J. UCS*, 12(4):370–394, 2006. 51

[37] Ivan Augé, Frédéric Pétrot, François Donnet, and Pascal Gomez. Platform-based design from parallel c specifications. *Computer-Aided Design of Integrated Circuits and Systems*, 24(12):1811–1826, 2005. 26, 27

[38] Brian Bailey and Grant Martin. *ESL Models and their Application: Electronic System Level Design and Verification in Practice*. Springer, 2010. 29

[39] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *Computer*, 36(4):45–52, 2003. 26

[40] Christos Baloukas, Lazaros Papadopoulos, Dimitrios Soudris, Sander Stuijk, Olivera Jovanovic, Florian Schmoll, Peter Marwedel, Daniel Cordes, Robert Pyka, Arindam Mallik, et al. Mapping embedded applications on mpsocs: The mnemee approach. In *ISVLSI*, pages 512–517, 2010. 64

[41] Mohamed A Bamakhrama, Jiali Teddy Zhai, Hristo Nikolov, and Todor Stefanov. A methodology for automated design of hard-real-time embedded streaming systems. In *Design, Automation Test in Europe (DATE)*, pages 941–946, 2012. 64

[42] Daniel Bartholomew. Qemu a multihost multitarget emulator. *Linux Journal*, 2006(145):3, 2006. 31

[43] Twan Basten, Emiel Van Benthum, Marc Geilen, Martijn Hendriks, Fred Houben, Georgeta Igna, Frans Reckers, Sebastian De Smet, Lou Somers, Egbert Teeselink, et al. Model-driven design-space exploration for embedded systems: the octopus toolset. In *Leveraging applications of formal methods, verification, and validation - Volume Part I*, pages 90–105. Springer-Verlag, 2010. 64

[44] Markus Becker, Henning Zabel, and Wolfgang Müller. Qemu/systemc cosimulation at different abstraction levels. In *1st International QEMU Users Forum*, 2011. 32

[45] Luca Benini, Eric Flamand, Didier Fuin, and Diego Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Design, Automation and Test in Europe (DATE)*, pages 983–987, 2012. 20, 85, 99

[46] Kshitij Bhardwaj and Rabindra K Jena. Energy and bandwidth aware mapping of ips onto regular noc architectures using multi-objective genetic algorithms. In *International Symposium on System-on-Chip (SOC)*, pages 027–031, 2009. 51

[47] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011. 32

[48] Nathan L Binkert, Ronald G Dreslinski, Lisa R Hsu, Kevin T Lim, Ali G Saidi, and Steven K Reinhardt. The m5 simulator: Modeling networked systems. *Micro*, 26(4):52–60, 2006. 32

[49] Luciano Bononi and Nicola Concer. Simulation and analysis of network on chip architectures: ring, spidergon and 2d mesh. In *Design, automation and test in Europe (DATE)*, pages 154–159, 2006. 85

[50] Shekhar Borkar. Thousand core chips: a technology perspective. In *Design Automation Conference (DAC)*, pages 746–749, 2007. ii, 3, 4

[51] Youcef Bouchebaba, Bruno Girodias, Gabriela Nicolescu, El Mostapha Aboulhamid, Bruno Lavigueur, and Pierre Paulin. Mpsoc memory optimization using program transformation. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12(4):43, 2007. 64

[52] Pierre Boulet. Array-OL Revisited, Multidimensional Intensive Signal Processing Specification. Research Report RR-6113, INRIA, 2007. 26

[53] R. Brillu, S. Pillement, A. Abdellah, F. Lemonnier, and P. Millet. Flextiles: A globally homogeneous but locally heterogeneous manycore architecture. In *Workshop on Rapid Simulation & Performance Evaluation: Methods and Tools of the HiPEAC conference*, 2014.

[54] R. Brillu, S. Pillement, and F. Lemonnier. Environnement de modélisation et de simulation d'architecture mpsoc: Ovp une solution fiable et effective ? In *Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS)*, 2013.

[55] R. Brillu, S. Pillement, F. Lemonnier, and P. Millet. Accelerator interface, a keystone for heterogenous "mpsoc" architectures. In *DATE Friday Workshop on Reconfigurable Computing V2.0: The Next Generation of Technology*, 2013.

[56] R. Brillu, S. Pillement, F. Lemonnier, and P. Millet. Flextiles a heterogeneous "mpsoc" architecture. In *GDR SoC-SiP*, 2013.

[57] R. Brillu, S. Pillement, F. Lemonnier, and P. Millet. Cluster based mpsoc architecture: An on-chip message passing implementation. *Design Automation for Embedded Systems*, 2014.

[58] R. Brillu, S. Pillement, F. Lemonnier, and P. Millet. Design space exploration for hw/sw codesign of mpsoc, 2014.

[59] R. Brillu, S. Pillement, F. Lemonnier, P. Millet, M. Bernot, and F. Falzon. Algorithm-architecture adequacy, an application to the phase diversity algorithm. In *GRETSI*, 2013.

[60] R. Brillu, S. Pillement, F. Lemonnier, P. Millet, E. Lenormand, M. Bernot, and F. Falzon. Towards a design space exploration tool for mpsoc platforms designs: a case study. In *Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, 2014.

[61] J-Y Brunel, WM Kruijtzer, HJHN Kenter, Frédéric Pétrot, L Pasquier, Erwin A de Kock, and WJM Smits. Cosy communication ip's. In *Design Automation Conference (DAC)*, pages 406–409, 2000. 26

[62] Horia Calborean, Ralf Jahr, Theo Ungerer, and Lucian Vintan. A comparison of multi-objective algorithms for the automatic design space exploration of a superscalar system. In *Advances in Intelligent Control Systems and Computer Science*, pages 489–502. Springer, 2013. 47

[63] Horia Calborean and Lucian Vintan. An automatic design space exploration framework for multicore architecture optimizations. In *Roedunet*, pages 202–207. IEEE, 2010. 48

[64] Pedro Angel Castillo, Juan Julián Merelo, Miquel Moreto, Francisco J Cazorla, Mateo Valero, Antonio Miguel Mora, Juan Luís Jiménez Laredo, and Sally A McKee. Evolutionary system for prediction and optimization of hardware architecture performance. In *Congress on Evolutionary Computation, CEC.*, pages 1941–1948, June 2008. 48

[65] Mario R Casu, Massimo Ruo Roch, Sergio V Tota, and Maurizio Zamboni. A noc-based hybrid message-passing/shared-memory approach to cmp design. *Microprocessors and Microsystems*, 35(2):261–273, 2011. 17, 98, 99

[66] Jichuan Chang and Gurindar S Sohi. *Cooperative caching for chip multiprocessors*, volume 34. IEEE Computer Society, 2006. 14

[67] Yin Chao and Wang Hongxia. Developed dijkstra shortest path search algorithm and simulation. In *International Conference On Computer Design and Applications (ICCDA)*, volume 1, pages 1–116, 2010. 53

[68] Yancang Chen, Lunguo Xie, and Jinwen Li. An energy-aware heuristic constructive mapping algorithm for network on chip. In *International Conference on ASIC, (ASICON)*, pages 101–104, 2009. 16, 51, 60

[69] Chen-Ling Chou and Radu Marculescu. Contention-aware application mapping for network-on-chip communication architectures. In *International Conference on Computer Design, (ICCD)*, pages 164–169, 2008. 50

[70] Naveen Choudhary, MS Gaur, V Laxmi, and Virendra Singh. Energy aware design methodologies for application specific noc. In *NORCHIP, 2010*, pages 1–4, 2010. 51

[71] Rosilde Corvino, Erkan Diken, Abdoulaye Gamatié, and Lech Jozwiak. Transformation-based Exploration of Data-Parallel Architecture for Customizable Hardware: A JPEG Encoder Case Study. In *Euromicro Conference on Digital System Design (DSD)*, 2012. 64, 68, 73

[72] Rosilde Corvino, Abdoulaye Gamatié, Marc Geilen, and Lech Jozwiak. Design space exploration in application-specific hardware synthesis for multiple communicating nested loops. In *Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS)*, 2012. 64, 68, 70, 73, 76

[73] Alain Darte. On the complexity of loop fusion. *Parallel Computing*, 26:149–157, 1999. 66

[74] Nicolas Ventroux—Raphaël David. Les architectures parallèles sur puce. *Journal techniques et sciences informatiques*, 29:345–378, 2010. 15, 13

[75] Benoît Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. A distributed run-time environment for the kalray mppa®-256 integrated manycore processor. *Procedia Computer Science*, 18:1654–1663, 2013. 19, 71, 73, 99

[76] Ewerson Luiz de Souza Carvalho, Ney Laert Vilar Calazans, and Fernando Gehm Moraes. Dynamic task mapping for mpsocs. *Design & Test of Computers*, 27(5):26–35, 2010. 51

[77] Francisco Fernndez de Vega, Jos Ignacio Hidalgo Prez, and Juan Lanchares. *Parallel Architectures and Bioinspired Algorithms*. Springer, 2012. 122

[78] Giuseppe Di Guglielmo, Franco Fummi, Graziano Pravadelli, Mark Hampton, and Florian Letombe. On the functional qualification of a platform model. In *Defect and Fault Tolerance in VLSI Systems, DFT*, pages 182–190, 2009. 31

[79] Andrew Duller, Daniel Towner, Gajinder Panesar, Alan Gray, and Will Robbins. Picoarray technology: the tool's story. In *Design Automation and Test in Europe (DATE)*, pages 106–111, 2005. 17

[80] Stephen A Edwards. Cocentric system studio, 2000. 27

[81] Stephen A Edwards. Kahn process networks. In *Languages for Digital Embedded Systems*, pages 189–195. Springer, 2000. 26

[82] Stephen A Edwards. Shim: A language for hardware/software integration. *Synchronous Programming-SYNCHRON*, pages 1–6, 2004. 87

[83] Amin El Mrabti, Frédéric Pétrot, and Aimen Bouchhima. Extending ip-xact to support an mde based approach for soc design. In *Design Automation and Test in Europe (DATE)*, pages 586–589, 2009. 87, 132

[84] Haytham Elmiligi, Ahmed A Morgan, M Watheq El-Kharashi, and Fayez Gebali. Power-aware topology optimization for networks-on-chips. In *International Symposium on Circuits and Systems (ISCAS)*, pages 360–363, 2008. 51

[85] Haytham Elmiligi, Ahmed A Morgan, M Watheq El-Kharashi, and Fayez Gebali. Power optimization for application-specific networks-on-chips: A topology-based approach. *Microprocessors and Microsystems*, 33(5):343–355, 2009. 51

[86] Ahmed A Eltawil, Michael Engel, Bibiche Geuskens, Amin Khajeh Djahromi, Fadi J Kurdahi, Peter Marwedel, Smail Niar, and Mazen AR Saghir. A survey of cross-layer power-reliability tradeoffs in multi and many core systems-on-chip. *Microprocessors and Microsystems*, 37(8):760–771, 2013. ii, 4

[87] Zhao Er-Dun, Qi Yong-Qiang, Xiang Xing-Xing, and Chen Yi. A data placement strategy based on genetic algorithm for scientific workflows. In *Computational Intelligence and Security (CIS)*, pages 146–149, 2012. 64

[88] ACE Associated Compiler Experts. The cosy compiler development system, 2007. 86

[89] Tom Feist. Vivado design suite, 2012. 34

[90] Leandro Fiorin, Slobodan Lukovic, and Gianluca Palermo. Implementation of a reconfigurable data protection module for noc-based mpsocs. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2008. 98

[91] Jose Flich and Davide Bertozzi. *Designing Network On-Chip Architectures in the Nanoscale Era*. Chapman & Hall/CRC, 2010. 6

[92] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language*. Access Online via Elsevier, 2011. 28

[93] Joshua Friedrich, Bradley McCredie, Norman James, Bill Huott, Brian Curran, Eric Fluhr, Gaurav Mittal, Eddie Chan, Yuen Chan, Donald Plass, et al. Design of the power6 microprocessor. In *Solid-State Circuits Conference (ISSCC)*, pages 96–97, 2007. 22

[94] Philip Garcia and Katherine Compton. A reconfigurable hardware interface for a modern computing system. In *Field-Programmable Custom Computing Machines (FCCM)*, pages 73–84, 2007. 87

[95] Rafael Garibotti, Luciano Ost, Rémi Busseuil, Chris Adeniyi-Jones, Gilles Sassatelli, Michel Robert, et al. Simultaneous multithreading support in embedded distributed memory mpsocs. In *Design Automation Conference (DAC)*, page 83. ACM, 2013. 98

[96] Fen Ge and Ning Wu. Genetic algorithm based mapping and routing approach for network on chip architectures. *Chinese Journal of Electronics*, 19(1):91–96, 2010. 16, 60

[97] M. Geilen, S. Stuijk, and T. Basten. Predictable dynamic embedded data processing. In *Embedded Computer Systems*, pages 320–327, 2012. 64

[98] Andreas Gerstlauer and Daniel D Gajski. System-level abstraction semantics. In *International symposium on System Synthesis*, pages 231–236, 2002. 26

[99] Pavel Ghosh, Arunabha Sen, and Alexander Hall. Energy efficient application mapping to noc processing elements operating at multiple voltage levels. In *International Symposium on Networks-on-Chip (NoCS)*, pages 80–85, 2009. 50

[100] Calin Glitia, Pierre Boulet, Eric Lenormand, and Michel Barreteau. Repetitive model refactoring strategy for the design space exploration of intensive signal processing applications. *Journal of Systems Architecture*, 57(9):815–829, 2011. 27, 39, 68

[101] Fred Glover and Manuel Laguna. *Tabu search*, volume 22. Springer, 1997. 9, 51, 52, 53, 58, 122, 134

[102] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman, 2013. 9, 47, 48, 65, 69, 122, 138

[103] Susanne Graf, Sébastien Gérard, Øystein Haugen, Iulian Ober, and Bran Selic. Modeling and analysis of real-time and embedded systems. In *Satellite Events at the MoDELS*, volume 3844, pages 58–66. Springer Berlin Heidelberg, 2006. 26

[104] Mentor Graphics. Catapult c synthesis, 2013. 33

[105] Alain Greiner. Tsar: a scalable, shared memory, many-cores architecture with global cache coherence. In *9th International Forum on Embedded MPSoC and Multicore (MPSoC'09)*, 2009. 15

[106] Yan Gu and Samarjit Chakraborty. Control theory-based dvs for interactive 3d games. In *Design Automation Conference (DAC)*, pages 740–745, 2008. 3

[107] Alexandre Guerre. *Approche hiérarchique pour la gestion dynamique des tâches et des communications dans les architectures massivement parallèles programmables*. PhD thesis, Paris 11, 2010. 15, 2, 12

[108] Tom R Halfhill. Ambric's new parallel processor. *Microprocessor Report*, 20(10):19–26, 2006. 17

[109] Haidar M Harmanani and Rana Farah. A method for efficient mapping and reliable routing for noc architectures with minimum bandwidth and area. In *Northeast Workshop on Circuits and Systems and TAISA Conference (NEWCAS-TAISA)*, pages 29–32, 2008. 51

[110] Damien Hedde and Frédéric Pétrot. A non intrusive simulation-based trace system to analyse multiprocessor systems-on-chip software. In *International Symposium on Rapid System Prototyping (RSP)*, pages 106–112, 2011. 121

[111] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012. 15, ii, 4

[112] D. Henry, S. Cheramy, J. Charbonnier, P. Chausse, M. Neyret, C. Brunet-Manquat, S. Verrun, N. Sillon, L. Bonnot, X. Gagnard, and E. Saugier. 3d integration technology for set-top box application. In *3D System Integration Conference*, pages 1–7, 2009. 82

[113] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van der Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 108–109, 2010. 16, 85, 98

[114] Jingcao Hu and R. Marculescu. Energy-aware mapping for tile-based noc architectures under performance constraints. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 233–239, 2003. 16, 60

[115] Jingcao Hu and R. Marculescu. Exploiting the routing flexibility for energy/performance aware mapping of regular noc architectures. In *Design Automation and Test in Europe (DATE)*, pages 688–693, 2003. 16, 60

[116] Jingcao Hu and Radu Marculescu. Energy-and performance-aware mapping for regular noc architectures. *Computer-Aided Design of Integrated Circuits and Systems*, 24(4):551–562, 2005. 16, 50, 60

[117] Q. Hu, P. G. Kjeldsberg, A. Vandecappelle, M. Palkovic, and F. Catthoor. Incremental hierarchical memory size estimation for steering of loop transformations. *ACM Trans. Des. Autom. Electron. Syst.*, 12(4), September 2007. 64

[118] Jia Huang, Christian Buckl, Andreas Raabe, and Alois Knoll. Energy-aware task allocation for network-on-chip based heterogeneous multiprocessor systems. In *International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 447–454, 2011. 50

[119] Michael Hübner and Jürgen Becker. *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*. Springer, 2010. 121

[120] Open SystemC Initiative. Systemc version 2.0 user's guide, 2001. 30

[121] Aws Ismail and Lesley Shannon. Fuse: Front-end user framework for o/s abstraction of hardware accelerators. In *Field-Programmable Custom Computing Machines (FCCM)*, pages 170–177, 2011. 87

[122] Wooyoung Jang and David Z Pan. A3map: architecture-aware analytic mapping for networks-on-chip. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 17(3):26, 2012. 16, 60

[123] Majid Janidarmian, Ahmad Khademzadeh, and Misagh Tavanpour. Onyx: A new heuristic bandwidth-constrained mapping of cores onto tile-based network on chip. *IEICE Electronics Express*, 6(1):1–7, 2009. 16, 51, 60

[124] Axel Jantsch, Xiaowen Chen, Abdul Naeem, Yuang Zhang, Sando Penolazzi, and Zhonghai Lu. Memory architecture and management in an noc platform. In *Scalable Multi-core Architectures*, pages 3–31. Springer, 2012. 98, 107

[125] Z.J. Jia, A. Núñez, T. Bautista, and A.D. Pimentel. A two-phase design space exploration strategy for system-level real-time application mapping onto mpsoc. *Microprocessors and Microsystems*, 38(1):9 – 21, 2014. 15, 25

[126] Yongjoo Kim, Jongeun Lee, Jinyong Lee, ToanX. Mai, Ingoo Heo, and Yunheung Paek. Exploiting both pipelining and data parallelism with simd reconfigurable architecture. In *Reconfigurable Computing: architectures, tools and applications*, (ARC), pages 40–52. Springer-Verlag, 2012. 64

[127] Yongjoo Kim, Jongeun Lee, Aviral Shrivastava, and Yunheung Paek. Operation and data mapping for cgras with multi-bank memory. In *ACM Sigplan Notices*, volume 45, pages 17–26. ACM, 2010. 64

[128] Yongjoo Kim, Jongeun Lee, Aviral Shrivastava, Jonghee Yoon, and Yunheung Paek. Memory-aware application mapping on coarse-grained reconfigurable arrays. In *High Performance Embedded Architectures and Compilers*, pages 171–185. Springer, 2010. 64

[129] Z. Konfrst. Parallel genetic algorithms: advances, computing trends, applications and perspectives. In *Parallel and Distributed Processing Symposium,*, pages 162–, April 2004. 122

[130] Krzysztof Kosciuszkiewicz, Fearghal Morgan, and Krzysztof Kepa. Run-time management of reconfigurable hardware tasks using embedded linux. In *International Conference on Field-Programmable Technology (ICFPT)*, pages 209–215, 2007. 87

[131] Nectarios Koziris, Michael Romesis, Panayotis Tsanakas, and George Papakonstantinou. An efficient algorithm for the physical mapping of clustered task graphs onto multiprocessor architectures. In *Parallel and Distributed Processing*, pages 406–413, 2000. 16, 60

[132] Sami Ktata and Faouzi Benzarti. License plate detection using mathematical morphology. In *Sciences of Electronics, Technologies of Information and Telecommunications*, pages 735–739, 2012. 93

[133] Rakesh Kumar, Dean M Tullsen, and Norman P Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *Parallel architectures and compilation techniques*, pages 23–32, 2006. 81, 107

[134] Sofiane Lagraa, Alexandre Termier, and Frédéric Pétrot. Data mining mpsoc simulation traces to identify concurrent memory access patterns. In *Design Automation and Test in Europe (DATE)*, pages 755–760. EDA Consortium, 2013. 121

[135] Christophe Lavarenne, Omar Seghrouchni, Yves Sorel, and Michel Sorine. The syndex software environment for real-time distributed systems design and implementation. In *European Control Conference*, volume 2, pages 1684–1689. Citeseer, 1991. 27

[136] Eugene L Lawler and David E Wood. Branch-and-bound methods: A survey. *Operations research*, 14(4):699–719, 1966. 50

[137] Choonseung Lee et al. A systematic design space exploration of mpsoc based on synchronous data flow specification. *Journal of Signal Processing Systems*, 58(2):193–213, 2010. 64

[138] F. Lemonnier, P. Millet, G. Marchesan Almeida, M. Hubner, J. Becker, S. Pillement, O. Sentieys, M. Koedam, S. Sinha, K. Goossens, C. Piguet, M. Morgan, and R. Lemaire. Towards future adaptive multiprocessor systems-on-chip: an innovative approach for flexible architectures. In *Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS)*, 2012. ii, iii, 3, 7

[139] Eric Lenormand and Gilbert Edelin. An industrial perspective: A pragmatic high end signal processing design environment at thales. In *Workshop on Systems, Architectures, Modeling and Simulation (SAMOS)*, pages 52–57, 2003. 27, 86

[140] Paul Lieverse, Pieter Van Der Wolf, Kees Vissers, and Ed Deprettere. A methodology for architecture exploration of heterogeneous signal processing systems. *Journal of VLSI signal processing systems for signal, image and video technology*, 29(3):197–207, 2001. 26

[141] Ting-Jung Lin, Shu-Yen Lin, and An-Yeu Wu. Traffic-balanced ip mapping algorithm for 2d-mesh on-chip-networks. In *Signal Processing Systems, (SiPS)*, pages 200–203, 2008. 50

[142] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008. 22

[143] Yan Y. Liu and Shaowen Wang. A scalable parallel genetic algorithm for the generalized assignment problem. *Parallel Computing*, (0):–, 2014. 122

[144] Nicolas Loménie. Visual point set processing with lattice structures: Application to parsimonious representations of digital histopathology images. In Frank Nielsen and Frédéric Barbaresco, editors, *Geometric Science of Information*, volume 8085 of *Lecture Notes in Computer Science*, pages 837–844. Springer Berlin Heidelberg, 2013. 46

[145] Zhonghai Lu, Lei Xia, and Axel Jantsch. Cluster-based simulated annealing for mapping cores onto 2d mesh networks on chip. In *Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pages 1–6, 2008. 16, 60

[146] Martin Lukasiewycz, Martin Streubühr, Michael Glaß, Christian Haubelt, and Jürgen Teich. Combined system synthesis and communication architecture exploration for mpsocs. In *Design Automation and Test in Europe*, (DATE), pages 472–477, 2009. 64

[147] Cao Man, Xie Bin, Qiao Fuming, Shi Qingsong, Chen Tianzhou, and Yan Like. Distributed memory management units architecture for noc-based cmps. In *Computer and Information Technology (CIT)*, pages 54–61, 2010. 98

[148] César AM Marcon, José CS Palma, Ney LV Calazans, Fernando G Moraes, Altamiro A Susin, and Ricardo Reis. Modeling the traffic effect for the application cores mapping problem onto nocs. In *Vlsi-Soc: From Systems To Silicon*, pages 179–194. Springer, 2007. 51

[149] Erik Jan Marinissen, Betty Prince, D Keltel-Schulz, and Yervant Zorian. Challenges in embedded memory design and test. In *Design, Automation and Test in Europe (DATE)*, pages 722–727, 2005. ii, 4

[150] Grant Martin and Steve Leibson. Commercial configurable processors and the mescal approach. In *Building ASIPS: The Mescal Methodology*, pages 281–310. Springer, 2005. 32

[151] Milo MK Martin, Daniel J Sorin, Bradford M Beckmann, Michael R Marty, Min Xu, Alaa R Alameldeen, Kevin E Moore, Mark D Hill, and David A Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, 2005. 32

[152] Sumit Mohanty, Viktor K Prasanna, Sandeep Neema, and J Davis. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. *ACM SIGPLAN Notices*, 37(7):18–27, 2002. 27

[153] Matteo Monchiero, Gianluca Palermo, Cristina Silvano, and Oreste Villa. Exploration of distributed shared memory architectures for noc-based multiprocessors. In *Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 144–151, 2006. 98

[154] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Moller, and Luciano Ost. Hermes: an infrastructure for low area overhead packet-switching networks on chip. *Integration, the VLSI Journal*, 38(1):69 – 93, 2004. 16, 48, 95

[155] Ralph Moritz, Tamara Ulrich, and Lothar Thiele. Evolutionary exploration of e/e-architectures in automotive design. In *Operations Research Proceedings 2011*, pages 361–366. Springer, 2012. 48

[156] Laurent Moss, M-A Cantin, Guy Bois, and El Mostapha Aboulhamid. Automation of communication refinement and hardware synthesis within a system-level design methodology. In *Rapid System Prototyping,RSP'08*, pages 75–81, 2008. 29

[157] Laurent Moss, Hubert Guérard, Gary Dare, and Guy Bois. Rapid design exploration on an esl framework featuring hardware-software codesign for arm processor-based fpga's. *Space*, 1, 2012. 29

[158] Srinivasan Murali, Luca Benini, and Giovanni De Micheli. Mapping and physical planning of networks-on-chip architectures with quality-of-service guarantees. In *Design Automation Conference (DAC)*, volume 1, pages 27–32, 2005. 50

[159] Srinivasan Murali and Giovanni De Micheli. Bandwidth-constrained mapping of cores onto noc architectures. In *Design automation and test in Europe (DATE)*, page 20896, 2004. 58

[160] Razvan Nane, Vlad-Mihai Sima, Bryan Olivier, Roel Meeuws, Yana Yankova, and Koen Bertels. Dwarv 2.0: A cosy-based c-to-vhdl hardware compiler. In *Field Programmable Logic and Applications (FPL)*, pages 619–622. IEEE, 2012. 34

[161] Anshuman Nayak, Malay Haldar, Alok Choudhary, and Prithviraj Banerjee. Accurate area and delay estimators for fpgas. In *Design automation and test in Europe (DATE)*, page 862, 2002. 33

[162] T.M. Parks, J.L. Pino, and E.A. Lee. A comparison of synchronous and cycle-static dataflow. In *Signals, Systems and Computers.*, volume 1, pages 204–210 vol.1, 1995. 26, 85

[163] Ahmad Patooghy, Hamed Tabkhi, and Seyed Ghassem Miremadi. Rmap: a reliability-aware application mapping for network-on-chips. In *International Conference on Dependability (DEPEND)*, pages 112–117, 2010. 51

[164] Dac Pham, Shigehiro Asano, Mark Bolliger, Michael N Day, H Peter Hofstee, C Johns, J Kahle, Atsushi Kameyama, John Keaty, Yoshio Masubuchi, et al. The design and implementation of a first-generation cell processor. In *International Solid-State Circuits Conference (ISSCC)*, pages 184–592, 2005. 18

[165] Andy D Pimentel, LO Hertzbetger, Paul Lieverse, Pieter van der Wolf, and EE Deprettere. Exploring embedded-systems architectures with artemis. *Computer*, 34(11):57–63, 2001. 26

[166] Ruxandra Pop and Shashi Kumar. A survey of techniques for mapping and scheduling applications to network on chip systems. *School of Engineering, Jonkoping University, Research Report*, 4:4, 2004. 6

[167] Joël Porquet, Alain Greiner, and Christian Schwarz. Noc-mpu: A secure architecture for flexible co-hosting on shared memory mpsocs. In *Design, Automation and Test in Europe (DATE)*, pages 1–4, 2011. 98

[168] Adrien Prost-Boucle, Olivier Muller, and Frédéric Rousseau. A fast and autonomous hls methodology for hardware accelerator generation under resource constraints. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 201–208. IEEE, 2013. 34

[169] Xiaogang Qiu and Michel Dubois. Moving address translation closer to memory in distributed shared-memory multiprocessors. *Parallel and Distributed Systems*, 16(7):612–623, 2005. 98

[170] L. Rabiner. The chirp z-transform algorithm-a lesson in serendipity. *Signal Processing Magazine*, 21(2):118–119, 2004. 68

[171] Talal Rahwan, Sarvapali D Ramchurn, Nicholas R Jennings, and Andrea Giovannucci. An anytime algorithm for optimal coalition structure generation. *Journal of Artificial Intelligence Research*, 34(2):521, 2009. 64

[172] Rohit Sunkam Ramanujam and Bill Lin. Destination-based adaptive routing on 2d mesh networks. In *Architectures for Networking and Communications Systems (ANCS)*, pages 1–12, 2010. 121

[173] Antti Rasmus, Ari Kulmala, Erno Salminen, and TD Hamalainen. Ip integration overhead analysis in system-on-chip video encoder. In *Design and Diagnostics of Electronic Circuits and Systems (DDECS'07)*, pages 1–4. IEEE, 2007. 87

[174] Midia Reshadi, Ahmad Khademzadeh, and Akram Reza. Elixir: A new bandwidth-constrained mapping for networks-on-chip. *IEICE Electronics Express*, 7(2):73–79, 2010. 16, 60

[175] Pradip Kumar Sahu and Santanu Chattopadhyay. A survey on application mapping strategies for network-on-chip design. *Journal of Systems Architecture*, 59(1):60–76, 2013. 16, ii, 6, 50, 51, 57, 58, 59, 60, 61, 63

[176] Pradip Kumar Sahu, Nisarg Shah, Kanchan Manna, and Santanu Chattopadhyay. A new application mapping algorithm for mesh based network-on-chip design. In *India Conference (INDICON)*, pages 1–4, 2010. 16, 60, 63

[177] Pradip Kumar Sahu, Putta Venkatesh, Sunilraju Gollapalli, and Santanu Chattopadhyay. Application mapping onto mesh structured network-on-chip using particle swarm optimization. In *Annual Symposium on VLSI (ISVLSI)*, pages 335–336, 2011. 51

[178] Prabhat Kumar Saraswat, Paul Pop, and Jan Madsen. Task mapping and bandwidth reservation for mixed hard/soft fault-tolerant embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 89–98, 2010. 51

[179] Wein-Tsung Shen, Chih-Hao Chao, Yu-Kuang Lien, and An-Yeu Andy Wu. A new binomial mapping and optimization algorithm for reduced-complexity mesh-based on-chip network. In *First International Symposium on Networks-on-Chip*, pages 317–322, 2007. 16, 51, 60

[180] Keng Siau and Terence Aidan Halpin. *Unified modeling language*. IGI Global, 2001. 28

[181] Cristina Silvano, William Fornaciari, Gianluca Palermo, Vittorio Zaccaria, Fabrizio Castro, Marcos Martinez, Sara Bocchio, Roberto Zafalon, Prabhat Avasare, Geert Vanmeerbeeck, et al. Multicube: Multi-objective design space exploration of multi-core architectures. In *VLSI 2010 Annual Symposium*, pages 47–63. Springer, 2011. 47

[182] Simulink. Hdl coder. 33

[183] D Singh. Implementing fpga design with the opencl standard. Technical report, Altera whitepaper, 2011. 34

[184] Nicolas Siret. *Etude de l'implémentation automatisée sur plateforme matérielle: logicielle d'applications de traitement du signal*. PhD thesis, Rennes, INSA, 2011. 16, 34

[185] Won So and Alexander G. Dean. Software thread integration for instruction-level parallelism. *ACM Trans. Embed. Comput. Syst.*, 13(1):8:1–8:23, 2013. 64

[186] Krishnan Srinivasan and Karam S Chatha. A technique for low energy mapping and routing in network-on-chip architectures. In *International symposium on Low power electronics and design*, pages 387–392, 2005. 16, 60

[187] Krishnan Srinivasan, Karam S Chatha, and Goran Konjevod. Linear-programming-based techniques for synthesis of network-on-chip architectures. *Very Large Scale Integration (VLSI) Systems*, 14(4):407–420, 2006. 50

[188] Joshua Suettlerlein, Stéphane Zuckerman, and Guang R. Gao. An implementation of the codelet model. In *International Conference on Parallel Processing*, pages 633–644, 2013. 98

[189] Xian-He Sun and Yong Chen. Reevaluating amdahl's law in the multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183 – 188, 2010. ii, 4

[190] Javid Taheri and Albert Y Zomaya. A pareto frontier for optimizing data transfer and job execution in grids. In *International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW),*, pages 2130–2139. IEEE, 2012. 64

[191] Domitian Tamas-Selicean and Paul Pop. Task mapping and partition allocation for mixed-criticality real-time systems. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 282–283, 2011. 51

[192] Misagh Tavanpour, Ahmad Khademzadeh, and Majid Janidarmian. Chain-mapping for mesh based network-on-chip architecture. *IEICE Electronics Express*, 6(22):1535–1541, 2009. 16, 60

[193] Misagh Tavanpour, Ahmad Khademzadeh, Somayyeh Pourkiani, and Mehdi Yaghobi. Gbmap: An evolutionary approach to mapping cores onto a mesh-based noc architecture. *Journal of Communication and Computer*, 7(3):1–7, 2010. 16, 60

[194] Leonel Pablo Tedesco, Thiago Rosa, Fabien Clermidy, Ney Calazans, and Fernando Gehm Moraes. Implementation and evaluation of a congestion aware routing algorithm for networks-on-chip. In *Symposium on integrated circuits and system design*, pages 91–96. ACM, 2010. 121

[195] Ungerer Theo, Robic Borut, and Silc Jurij. Multithreaded processors. *The Computer Journal*, 45:320–348, 2002. 21

[196] Anita Tino and Gul N Khan. Multi-objective tabu search based topology generation technique for application-specific network-on-chip architectures. In *Design Automation and Test in Europe (DATE)*, pages 1–6, 2011. 51

[197] S. Tosun, O. Ozturk, and M. Ozen. An ilp formulation for application mapping onto network-on-chips. In *Application of Information and Communication Technologies (AICT)*, pages 1–5, 2009. 16, 50, 60, 63

[198] Suleyman Tosun. Cluster-based application mapping method for network-on-chip. *Advances in Engineering Software*, 42(10):868–874, 2011. 16, 50, 60

[199] Suleyman Tosun. New heuristic algorithms for energy aware application mapping and routing on mesh-based nocs. *Journal of Systems Architecture*, 57(1):69–78, 2011. 16, 51, 60

[200] Sergio V Tota, Mario R Casu, Massimo Ruo Roch, Luca Rostagno, and Maurizio Zamboni. Medea: a hybrid shared-memory/message-passing multiprocessor noc-based architecture. In *Design Automation Test in Europe (DATE)*, pages 45–50, 2010. 99

[201] S.V. Tota and others. A case study for noc-based homogeneous mpsoc architectures. *Very Large Scale Integration (VLSI)*, 17(3):384–388, 2009. 98, 99

[202] Marc Tremblay and Shailender Chaudhry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread cmt sparc® processor. In *Solid-State Circuits Conference (ISSCC)*, pages 82–83, 2008. 22

[203] Frank Vahid, Tony Givargis, and John Wiley. *Embedded system design: a unified hardware/software introduction*, volume 4. John Wiley & Sons New York, NY, 2002. 44, 88

[204] Sriram Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Priya Iyer, Arvind Singh, Tiju Jacob, et al. An 80-tile 1.28 tflops network-on-chip in 65nm cmos. In *International Solid-State Circuits Conference (ISSCC)*, pages 98–589, 2007. 16

[205] Nicolas Ventroux and Raphaël David. Scmp architecture: an asymmetric multiprocessor system-on-chip for dynamic applications. In *International Forum on Next-Generation Multicore/Manycore Technologies*, page 6, 2010. 19, 98

[206] Karl Viring et al. Application task and data placement in embedded many-core numa architectures. In *Workshop on Optimizations for DSP and Embedded Systems*, (ODES), pages 37–44, 2013. 64

[207] M. Vuletic, L. Pozzi, and P. Ienne. Virtual memory window for application-specific reconfigurable coprocessors. *Very Large Scale Integration (VLSI) Systems*, 14(8):910–915, Aug 2006. 87

[208] M Vuletid, Laura Pozzi, and Paolo Ienne. Seamless hardware-software integration in reconfigurable computing systems. *Design & Test of Computers*, 22(2):102–113, 2005. 87

[209] Jian Wang, Yubai Li, Song Chai, and Qicong Peng. Bandwidth-aware application mapping for noc-based mpsocs. *Journal of Computational Information Systems*, 7(1):152–159, 2011. 16, 60

[210] Xiaohang Wang, Mei Yang, Yingtao Jiang, and Peng Liu. A power-aware mapping approach to map ip cores onto nocs under bandwidth and latency constraints. *ACM Transactions on Architecture and Code Optimization (TACO)*, 7(1):1, 2010. 51

[211] Zhou Wenbiao, Zhang Yan, and Mao Zhigang. A link-load balanced low energy mapping and routing for noc. In Yann-Hang Lee, Heung-Nam Kim, Jong Kim, Yongwan Park, LaurenceT. Yang, and SungWon Kim, editors, *Embedded Software and Systems*, volume 4523 of *Lecture Notes in Computer Science*, pages 59–66. Springer Berlin Heidelberg, 2007. 51

[212] Linda Wilson. *International Technology Roadmap for Semiconductors (ITRS)*. Semiconductor Industry Association, 2013. i, 3

[213] Matthieu Wipliez, Ghislain Roquier, and Jean-François Nezan. Software code generation for the rvc-cal language. *Journal of Signal Processing Systems*, 63(2):203–213, 2011. 34

[214] Wayne Wolf. The future of multiprocessor systems-on-chips. In *Design Automation Conference (DAC)*, pages 681–685, 2004. 20

[215] Jiong Xie, Shu Yin, Xiaojun Ruan, Zhiyang Ding, Yun Tian, James Majors, Adam Manzanares, and Xiao Qin. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In *International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW),*, pages 1–9. IEEE, 2010. 66

[216] Bo Yang, Thomas Canhao Xu, Tero Santti, and Juha Plosila. Tree-model based mapping for energy-efficient and low-latency network-on-chip. In *Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pages 189–192, 2010. 51

[217] Yang Yang et al. Exploring trade-offs between performance and resource requirements for synchronous dataflow graphs. In *Embedded Systems for Real-Time Multimedia,*, pages 96–105, 2009. 64

[218] Ming-Yan Yu, Ming Li, Jun-Jie Song, Fang-Fa Fu, and Yu-Xin Bai. Pipelining-based high throughput low energy mapping on network-on-chip. In *Euromicro Conference on Digital System Design, Architectures, Methods and Tools, (DSD)*, pages 427–432, 2009. 51

[219] Jun Zhang, Tan Deng, Qiuyan Gao, Qingfeng Zhuge, and Edwin H-M Sha. Optimizing data placement of loops for energy minimization with multiple types of memories. *J. Signal Process. Syst.*, 72(3):151–164, 2013. 64

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

THALES

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Romain BRILLU

Conception et programmation efficace de systèmes multiprocesseurs sur puce

Efficient design and programming of Multiple Processors System on Chip architectures

Résumé

Les applications embarquées incorporent de plus en plus de fonctionnalités impliquant différents types de traitement à réaliser. L'impact majeur de cette demande est l'évolution croissante des systèmes embarqués que cela soit en terme de performances et de capacité mémoire. Ces systèmes doivent en effet trouver un compromis entre leurs capacités (puissance de calcul, dynamicité) et les contraintes du domaine d'application. Face à cette évolution les architectures MPSoC apparaissent actuellement comme les principaux promoteurs de la révolution industrielle des semi-conducteurs. Cependant, la conception d'une architecture "MPSoC" faible consommation et supportant les performances requises, n'est pas aisée. Cet équilibre dépend en effet de nombreux paramètres tels que le nombre de cœurs de calcul, l'enveloppe énergétique globale, le type de réseau d'interconnexion, l'architecture de la hiérarchie mémoire, le déploiement de l'application sur le système.

Tous ces défis durant la conception des architectures MPSoC mettent en lumière le besoin de processus automatisé aidant l'utilisateur à définir et à programmer ces architectures. Dans le cadre de cette thèse, notre contribution est la définition d'une méthodologie d'exploration d'espace de conception. Cette méthodologie a pour but de définir à la fois une architecture matérielle et son code binaire exécutable à partir de trois entrées : (1) Le code C séquentiel d'une application, (2) Une librairie d'architectures, (3) Un fichier de contraintes. De plus, étant donné que nous souhaitons explorer et générer des architectures matérielles, notre seconde contribution est la définition de deux modules matériels. Le premier module matériel défini une unité de management mémoire servant à facilter la programmation des architectures MPSoC et permettant d'augmenter leurs performances. Le second module matériel est l'"accelerator interface" qui est utilisé pour abstraire l'hétérogénéité des plateformes MPSoC , afin de faciliter leur conception et leur programmation.

Mots clés

Systèmes embarquées, MPSoC, Hétérogénéité, Généricité, Mémoire partagée, Mémoire distribuée, Algorithme génétique, Algorithme de Tabu Search.

Abstract

The embedded applications come up with more and more functionalities inducing various kinds of computation to realize. The major impact of these new application needs is the steadily evolution of the embedded systems performances in terms of computing power and memory capacity. These systems have to find a trade-off between their capacity (computing power, dynamicity) and the embedded system constraints (silicium, consumption). To face these hard constraints MPSoC architectures have appeared as a major promoter of the industrial revolution of semiconductors. However, designing a low power MPSoC architecture, supporting the required performance is not easy. This balance depends on the effects of various parameters such as the number of cores, the overall energy envelope, the type of interconnection network, the architecture of the memory hierarchy, the deployment of the application on the system.

All these challenges during the definition of MPSoC architectures spotlight the needs of an automatic design process to help the user design and program these architectures. In the context of this thesis our contributions is the definition of a design space exploration methodology. This methodology aims to define a hardware architecture and the associated executable binary code based on three inputs: (1) An application C code, (2) An architecture library and (3) A constraints file. Moreover because we aims to explore and generates hardware architectures our second contribution is the definition of two hardware modules. The first hardware module defines a hardware memory management unit used to ease the programming of the MPSoC architectures and increase their performances. The second hardware module is the accelerator interface which is used to abstract the heterogeneity of the heterogeneous MPSoC architectures, ease their definition and programming.

Key Words

Embedded systems, MPSoC, Heterogeneity, Genericity, Shared memory, Distributed memory, Genetic algorithm, Tabu search algorithm.