Modélisation flux de données et optimisation pour architecture multi-cœurs de motifs répétitifs

Jonathan Piat

Thèse

THESE INSA Rennes
sous le sceau de l'Université européenne de Bretagne
pour obtenir le titre de
DOCTEUR DE L'INSA DE RENNES
Spécialité : Traitement du signal et des images

présentée par
Jonathan Piat
ECOLE DOCTORALE : *MATISSE*
LABORATOIRE : *IETR*

Modélisation flux de données et optimisation pour architecture multi-cœurs de motifs répétitifs

Thèse soutenue le 16.09.2010
devant le jury composé de :

Mohamed Akil
Professeur des universités, ESIEE Paris, France/ *rapporteur*
Pierre Boulet
Professeur des universités, LIFL Villeneuve d'Ascq, France / *rapporteur*
Shuvra S. Bhattacharyya
Professeur des universités, UMD College Park, USA / *examinateur*
Steven Derrien
Maître de conférence, IFSIC Rennes, France / *examinateur*
Olivier Deforges
Professeur des universités, INSA Rennes, France / *directeur de thèse*
Mickaël Raulet
Ingénieur de recherche, INSA Rennes, France / *encadrant*

Acknowledgements

The work in the thesis is the result of a three-year work at the INSA of Rennes. I really enjoyed spending time working at the lab, and I would like to thank the people that made it possible.

First, I want to thank my supervisor, Pr. Olivier Deforges, for making this work possible and for the time he spent correcting my mistakes in this thesis. I also would like to thanks my co-advisors, Mickaël Raulet and Pr Jean-François Nezan, for giving me support and freedom in my work. I address special thanks to Mickaël Raulet for his contribution and Olivier Déforges' work with correcting my orthography. Thanks to Shuvra S. Bhattacharyya who helped a lot on the research and publications. His contributions during the last two years were valuable and helped me to overcome a lot of difficulties in my work. Thanks to, Pr. Mohamed Akil and Pr. Pierre Boulet, for being the reviewers of this thesis. Moreover, thanks to Steven Derrien and Pr. Shuvra S. Bhattacharyya for their participation as member of the jury.

Spending three years on the same topic would have affected my mental health without following colleagues help on both work and ambiance (not listed in any order of importance): Maxime Pelcat, Mathieu Wipliez, Méderic Blestel, Jérôme Gorin, Fabien Racapé, Emilie Bosc, Pierre Laurent Lagalaye, Mathieu Urvoy, Yohan Pitrey. Being passionate by embedded systems, robotic, and general electronic I found great geek-mates at the lab that helped me full filing my geekyness needs: Sylvain Haese, Eric Bazin, Clément Strauss, François Pasteau, Xavier Perraud. Thanks to Jocelyne Trermier, Denis Simon, Frédéric Garesche and all the technical and administrative staff of the INSA of Rennes. Special thanks to Emilie Besse for enlightening our workdays by presenting the news with her natural softness.

Those three years would not have been possible without a personal and social life. Thanks to Emilie Malenfant for sharing my life the past seven years, bearing with my meaningless technical conversation, standing by my mess and for her support in good and bad times. Thanks to her family for being close and supportive. This work would not have been possible without my family whose presence the last twenty seven years helped constructing the person I am. Thanks to my missing dad, whose positive influence has undoubtedly guided me to this point (and beyond).

Table des matières

Première partie

French Summary

Chapitre 1

Introduction

Devant la demande croissante en puissance de calcul, des applications de traitement du signal et notamment, des applications d'encodage/décodage d'images fixes et vidéos, il est de plus en plus courant de faire appel à la puissance des architectures multi-coeurs/multiprocesseurs. Ce type d'architecture présente une puissance potentielle importante, mais la complexité d'implémentation d'une application est élevée. En effet, ce type d'architecture peut, selon le contexte, faire appel à des types de processeurs différents tels que DSP ou GPU mais aussi à des coeurs généralistes (x86, ARM ...). Ces coeurs de processeur sont interconnectés au travers de réseaux de communication présentant des topologies adaptées.

Afin de parvenir rapidement à une implémentation fiable et efficace sur ce type d'architecture, il est nécessaire de faire appel à des outils de prototypage rapide tel que PREESM, PEaCe ou SynDEx qui proposent des méthodes automatiques pour parvenir à une solution d'implantation d'un algorithme sur une architecture parallèle rapidement. Ces outils s'appuient sur un modèle d'application de type flux de données, qui modélise l'application au travers des dépendances de données qui existent entre les acteurs qui la compose. Un acteur est un élément opératoire qui consomme des données, effectue un calcul et produit des données. Ces acteurs sont connectés au sein d'un réseau au travers d'arcs qui symbolisent les dépendances de données.

L'outil PREESM est l'un des fruits de cette thèse et s'appuie sur le modèle Synchronous Data Flow, ou modèle Flux de Données Synchrone. Ce modèle spécifie pour chaque tâche le nombre de données produites et consommées par chaque acteur pour une exécution de celui-ci. Ces informations permettent d'extraire un ordre total d'exécution des acteurs qui composent le réseau ainsi que la mémoire requise pour la communication entre les acteurs. Dans le modèle flux de données, il existe une représentation hiérarchique du réseau qui permet de spécifier le comportement de chaque acteur au travers d'un réseau flux de données. Dans le modèle SDF (Synchronous

Data Flow), la technique du clustering permet d'extraire une représentation hiérarchique à partir d'une représentation à plat, mais il n'existe pas de modèle spécifique de hiérarchie qui permette la spécification d'un réseau SDF ou chaque acteur peut lui même être spécifié par un réseau SDF.

La première contribution de cette thèse est de définir un tel modèle de hiérarchie en spécifiant des règles qui assurent la composition (comme définit précédemment). De ce fait, il est possible d'extraire un ordonnancement du réseau de plus haut niveau indépendamment de l'ordonnancement des réseaux qui spécifient le comportement des acteurs. Ce modèle s'approche, par son interprétation, du découpage en fonctions et blocks possible dans le code C.

Une autre contribution de cette thèse est la partie génération de code C de l'outil PREESM. Cette génération de code a pour but de délivrer un code C dans un format compréhensible et structuré. De plus, la flexibilité de la méthode de génération permet à l'utilisateur de configurer la technique d'allocation mémoire et de facilement s'adapter à différents types de matériels et OS. Il est également envisageable de générer du code dans un langage autre que le C avec peu de modifications.

Enfin, une dernière contribution établit l'utilisation de méthodes d'extraction de parallélisme dans les nids de boucles appliquées au domaine de la modélisation flux de données synchrone. Ce type de transformation permet d'extraire du parallélisme d'une description factorisée de l'application tout en conservant un nombre réduit de sommets à ordonnancer.

Durant cette thèse, j'ai également participé à des contributions dans le domaine du codage vidéo. En effet, l'équipe "<image"> de l'IETR est impliquée dans l'encodage/décodage d'images fixes et de vidéos. L'équipe participe depuis peu au standard MPEG-RVC, et j'ai été impliqué dans des travaux connexes. La première contribution concerne la validation de bit-stream dans RVC. J'ai participé au développement de l'outil de validation, et j'ai apporté ma contribution à une publication sur la procédure de validation d'un bit-stream. J'ai, également, participé au portage sur plateforme ARM d'un décodeur vidéo reconfigurable. Ce décodeur utilise la technologie LLVM pour implémenter un décodeur reconfigurable à partir d'une description d'un réseau d'acteurs CAL. Une description plus complète de l'environnement des contributions est décrite ci-après.

1.1 Contributions du travail de thèse

1.1.1 Prototypage rapide

Les travaux contenus dans cette thèse sont des contributions au logiciel de prototypage rapide PREESM (Parallel Real-tile Embedded Executive Scheduling Method). Le prototypage rapide vise à établir rapidement un prototype d'une application donnée pour une architecture donnée. Ce prototype est généré à partir de descriptions haut niveau de l'architecture et l'application. A partir de ces spécifications haut niveau, le logiciel se charge d'établir un partitionnement de l'application sur les coeurs de l'architecture cible. Il génère, ensuite, un ordonnancement pour chaque coeur. L'objectif final est de générer, à partir de cet ordonnancement, du code compilable et exécutable. Le logiciel prend donc en charge une majeure partie de la complexité afin de faciliter le travail du programmeur.

1.1.2 Reconfigurable Video Coding

Le standard RVC définit par MPEG, s'attache à décrire les actions qui composent un décodeur générique pour en dresser une bibliothèque et pouvoir configurer dynamiquement un décodeur. Les acteurs sont appelés "<Functionnal Units"> (Unités fonctionnels ou FU) et sont instanciés dans un réseau qui définit les échanges de données entre FU. A partir de cette description, il est alors possible de créer dynamiquement une structure de décodage vidéo. Le langage de spécification des acteurs est RVC-CAL.

1.1.3 Bit-Stream Description Language

Dans le cadre du standard RVC, il est nécessaire d'être capable de décrire la structure du flux d'informations d'entrée du décodeur. En effet, le seul élément non standard dans l'ensemble des codecs, est la structure du flux. Afin de valider la structure du flux, il est nécessaire d'avoir recours au standard BSDL qui fournit la syntaxe de description du flux et en décrire la structure dans un fichier de type xslt. Il est également possible de générer dynamiquement un acteur capable d'interpréter ce flux et ainsi l'utiliser dans le cadre de RVC. J'ai travaillé, dans ce contexte, sur les outils qui permettent d'obtenir la description BSDL d'un flux à partir d'une séquence, et la séquence à partir d'une description BSDL. J'ai également participé à la rédaction d'un article sur la méthode de validation de la structure d'un flux.

1.1.4 Organisation de la thèse

La suite de cette thèse s'organise comme suit :
- Cette première partie présente un résumé substantiel du travail effectué en français
- Une deuxième partie dresse un état de l'art en anglais des éléments nécessaires à la compréhension de la suite du document
- Une troisième partie en anglais présente les travaux de recherche effectués.

Chapitre 2

Etat de l'art

2.1 Prototypage rapide

2.1.1 Introduction

Le prototypage rapide s'attache à fournir un ensemble d'étapes automatisées qui permettent, à partir d'une description haut niveau de l'application et de l'architecture, de générer une solution optimisée d'implantation. Les outils de prototypage rapide répondent à la problématique d'implantation d'algorithmes complexes sur des architectures parallèles. Ces architectures embarquent un nombre variable de processeurs, ainsi que différents moyens de communication entre ces processeurs. Le modèle d'application utilisé doit fournir suffisamment d'informations pour extraire du parallélisme sans dénaturer le comportement de l'application. Le modèle flux de données symbolise une application par les dépendances de données qui existe entre ces constituants. A l'aide de ce modèle, il est ainsi aisé d'extraire le parallélisme intrinsèque et différentes informations permettant l'optimisation mémoire et l'ordonnancement.

2.1.2 Systèmes de calculs distribués

Un système de calcul distribué est caractérisé par plusieurs éléments :

- Application cible : un système de calcul distribué est avant tout caractérisé par l'application cible. On distinguera deux grands types d'applications : dominés par le contrôle ou dominés par les données. En effet, une application dominée par le contrôle devra, par exemple, présenter un temps de réaction faible, tandis qu'une application dominée par les données devra être capable d'un important débit et d'une puissance de calcul.
- Consommation/Performance : Les choix architecturaux sont, dans le cas des

applications mobiles, guidés par le choix du meilleur rapport consommation/-performance. En effet, une architecture parallèle offre plus de possibilités pour améliorer ce rapport. La multiplication des coeurs, permet (selon l'application) de baisser la fréquence de ceux-ci et ainsi limiter les pertes joules.
- Elément de calcul : L'élément de calcul, est, dans le cas des architectures multi-coeurs, un coeur du système. Un système multi-coeurs est donc généré par l'interconnexion de multiples éléments de calculs homogènes ou hétérogènes. Cet élément de calcul est caractérisé par le jeu d'instructions qu'il prend en charge et sa microarchitecture. La microarchitecture définit la manière dont est implémenté le jeu d'instruction. L'efficacité de l'élément de calcul dépend grandement de sa microarchitecture.
- Système mémoire : Le système mémoire de l'architecture est caractérisé par : le système de cache, l'interconnexion des éléments de calcul, et la gestion de la cohérence.
- Accélérateurs, Périphériques : Les accélérateurs et périphériques sont des composants non programmables (configurables) de l'architecture. Ces composants effectuent des tâches dédiées avec une grande efficacité et en parallèle des éléments de calcul. Le DMA (Direct Memory Access) peut, par exemple, effectuer des transferts de mémoire à mémoire sans intervention d'un élément de calcul.

Les éléments donnés ci-dessus permettent de caractériser de manière précise un système de calcul distribué de type multi-coeurs. Pour un système de type multi-processeurs, pour lequel la division des coeurs et à la fois logique et spatiale, il est également nécessaire de caractériser le système de communication entre les processeurs. En effet, dans un système multi-coeurs, les éléments de calcul résident dans un seul composant physique. Dans le cas d'un système multiprocesseurs, plusieurs éléments physiques sont connectés par un medium. Ce type d'interconnexion présente des caractéristiques différentes de l'interconnexion sur puce.

2.1.3 Modèle de calcul flux de données

Les modèles de calcul flux de données s'attachent à décrire le comportement d'une application au travers des dépendances de données qui existent entre les tâches qui la composent. La description flux de données repose sur trois éléments :
- Acteur : Un acteur représente une entité de calcul de l'application.
- Jeton de données : Un jeton de données est l'élément d'une donnée atomique que s'échangent les acteurs.
- Arc : Un arc connecte deux acteurs. Un acteur produit des données sur l'arc que l'acteur/consommateur peut ensuite consommer.

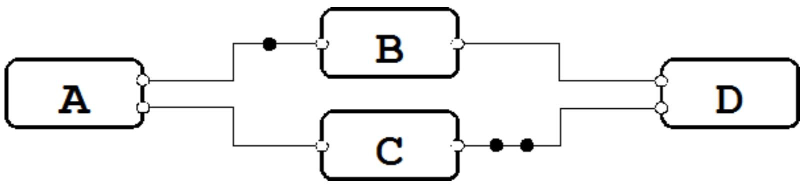


FIGURE 2.1 – *Réseau d'acteurs échangeant des jetons de données*

L'interconnexion des acteurs au travers des arcs constitue un réseau. Le modèle flux de données le plus générique est appelé *Kahn Process Network* (KPN). Dans ce modèle, les acteurs sont connectés par l'intermédiaire de FIFO (First In First Out) unidirectionnels non limités. Le modèle Data flow Process Network limite le modèle KPN en associant aux acteurs des règles de déclenchement (Firing rules). Ces règles spécifient les séquences de données d'entrée permettant l'exécution de l'acteur. D'autres modèles tels que le SDF restreignent l'expressivité en associant pour chaque arc trois nombres entiers représentant le nombre de jetons produits (production rate), le nombre de jetons consommés (consumption rate) et le nombre de jetons initialement présents sur l'arc. Ce modèle permet de déterminer un ordonnancement de l'application avant exécution et ainsi garantir son bon fonctionnement à l'exécution et une consommation mémoire bornée.

2.1.4 Ordonnancement multiprocesseurs

Grâce à un modèle des caractéristiques de l'architecture multi-coeurs et un modèle flux de données de l'application ; il est possible de déterminer la meilleure implémentation de l'application sur l'architecture. Pour y parvenir il est, tout d'abord nécessaire, de répartir les acteurs qui composent la description flux de données sur les différents opérateurs de l'architecture. Cette opération est appelée : allocation. Ensuite, il faut, pour chaque processeur, décrire l'ordonnancement des acteurs et organiser les communications entre opérateurs. Cette implémentation cherche à optimiser un ou plusieurs critères d'exécution tel que le temps d'exécution (cadence ou latence), l'empreinte mémoire et/ou la consommation électrique.

2.2 Transformation des nids de boucles

2.2.1 Boucles imbriquées et ordre séquentiel

Definition On désigne par « nid de boucles »une structure composée de plusieurs boucles imbriquées.

Cette définition peut être restreinte dans le cas des nids de boucles dit parfaits ou parfaitement imbriqués.

Definition On désigne par « nid de boucles parfait »un nid de boucles pour lequel le code de chaque boucle ne contient qu'une autre boucle à l'exception de la boucle la plus interne.

Exemple de nid de boucles imparfait :

```
for $i_1$ := $l_1$ to $u_1$ do
 for $i_2$ := $l_2(i_1)$ to $u_2(i_1)$ do
   ...
   for $i_n$ := $l_n(i_1, i_2, ..., i_{n-1})$ to $u_n(i_1, i_2, ..., i_{n-1})$ do
    $S : \{Instruction1\}$
     ...
    $\{Instructionk\}$
   end
 end
end
```

FIGURE 2.2 – *Example de nid de boucles imparfait*

Chaque boucle définit un vecteur d'itération (ex : $i = 0$ to N) et l'imbrique-ment des boucles définit un domaine d'itération de dimension égale à la profondeur du nid (le nombre de boucles imbriquées). Dans notre exemple, ce nid de boucles définit un domaine dans Z^n inclus dans le polyèdre $l_1 \leq i_1 \leq u_1, l_2(i_1) \leq i_2 \leq u_2(i_1)...l_n(i_1, i_2, ..., i_{n-1}) \leq i_n \leq u_n(i_1, i_2, ..., i_{n-1})$. Le programme exécute chaque opération notée S entourée par une boucle pour toutes les valeurs I du vecteur d'itération dans un ordre séquentiel noté $<_{seq}$ définit par l'ordre lexicographique du vecteur d'itération.

$$S(I) <_{seq} S(J) \leftarrow I <_{lex} J$$

2.2.2 Dépendance de données dans les nids de boucles

Types de dépendances

Les catégories de dépendances identifiées par Bernstein et résumées dans leur livre par Alain Darte, Yves Robert et Frédéric Vivien [AYF00], sont de trois types : Les dépendances de flot, anti-dépendances et dépendances en sortie. Il y a une dépendance de données entre $S(I)$ et $T(J)$ si les deux opérations accèdent au même emplacement mémoire et si au moins un accès est une écriture. La dépendance est dirigée comme dans l'ordre séquentiel, c'est-à-dire qu'il y a dépendance de $S(I) \rightarrow T(J)$

si dans l'ordre séquentiel on à $S(I) <_{seq} T(J)$. Les trois types de dépendances sont ici :

- **Dépendance de flot :** Si l'accès à l'emplacement mémoire commun est une écriture pour $S(I)$ et une lecture pour $T(J)$ et qu'il n'y a aucun accès en écriture sur l'emplacement entre $S(I)$ et $T(J)$ dans l'ordre séquentiel.
- **Anti-dépendance :** Si l'accès à l'emplacement mémoire commun est une lecture pour $S(I)$ et une écriture pour $T(J)$ et qu'il n'y a aucun accès en écriture sur l'emplacement entre $S(I)$ et $T(J)$ dans l'ordre séquentiel.
- **Dépendance en sortie :** Si $S(I)$ et $T(J)$ sont deux écritures consécutives au même emplacement mémoire.

Représentation des dépendances

Les vecteurs de dépendances expriment dans le domaine D des itérations la dépendance entre deux itérations. Le calcul de ces vecteurs se fait par analyse du code des boucles du nid.

```
for $i$ := 0 to $N$ do
 for $j$ := 0 to $N$ do
   $S_1 : a(i,j) = b(i,j-6) + d(i-1,j+3)$
   $S_2 : b(i+1,j-1) = c(i+2,j+5)$
   $S_3 : c(i+3,j-1) = a(i,j-2)$
   $S_4 : d(i,j-1) = a(i,j-1)$
   end
 end
end
```

FIGURE 2.3 – *Exemple de nid de boucles*

Dans l'exemple de nid de boucles (figure 2.3)décrit ici, on peut établir quatre vecteurs de dépendances. Le premier décrit la dépendance existant entre l'instruction S_1 et l'instruction S_3, ce vecteur décrira donc une dépendance de flot.

$$S_1 \rightarrow S_3 \; d_1 = \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

Ce vecteur est obtenu par égalité sur les indices, pour cela on pose l'égalité sur les indices d'itération entre l'instruction S_1 et l'instruction S_3, ce qui nous donne.

$i_3 = i_1 - 2$ et $j_3 = j_1$, la dépendance étant une dépendance de flot le vecteur est orienté de S_3 vers S_1 on a donc $i_1 = i_3 + 2$ et $j_3 = j_1$, ce qui nous donne le vecteur de dépendance. Sur l'exemple, les autres vecteurs de dépendance sont :

$S_3 \rightarrow S_2 \ d_2 = \begin{pmatrix} 1 \\ -6 \end{pmatrix}$

$S_2 \rightarrow S_1 \ d_3 = \begin{pmatrix} 1 \\ 5 \end{pmatrix}$

$S_1 \rightarrow S_4 \ d_4 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$

$S_4 \rightarrow S_1 \ d_5 = \begin{pmatrix} 1 \\ -4 \end{pmatrix}$

Ces vecteurs sont largement utilisés dans les méthodes de transformation de nid de boucles ou d'ordonnancement.

2.2.3 Transformation des nids de boucles

Types de transformation

Afin de maximiser le parallélisme exploitable dans un nid boucles, il est nécessaire de passer par une étape de transformation, rendant explicite le parallélisme implicite. Une transformation est dite légale si elle respecte les dépendances entres les itérations d'un nid boucles et donc si le résultat de l'exécution de la boucle transformée est conforme au résultat attendu. Les transformations peuvent prendre plusieurs formes.

Distribution : Ce type de transformation vise à distribuer un nid de boucles mal structuré afin de faire apparaitre au moins une boucle forall.

Fusion : Ce type de transformation a pour but de fusionner le corps de plusieurs boucles en une seule. Cela a pour effet d'augmenter la localité des données et de minimiser les communications.

Déroulement de la boucle : Dans cette transformation on déroule les itérations de la boucle afin de faire apparaitre un parallélisme maximum.

Partitionnement : On cherche ici à scinder les boucles afin de faire apparaitre des sous ensembles disjoints.

Transformation uni-modulaire : Dans une transformation uni-modulaire on change l'ordre dans lequel les vecteurs d'itérations sont énumérés. Le nouveau vecteur d'itération est déterminé par le produit du vecteur d'itération originel avec une matrice uni-modulaire. $I' = TI$.

Méthode de l'hyperplan

Cette méthode de transformation de boucles dite méthode de l'hyperplan ou méthode de Lamport [Lam74] consiste à chercher à transformer un nid de boucles en un nid de boucles pour lequel un certain nombre de boucles internes peuvent être exécutés en parallèle. Le cas optimal est le cas où toutes les boucles internes peuvent être effectuées en parallèle.

```
for $t$ := 0 to $t_N$ do
 for $i_2$ := $l_2(i_1)$ to $u_2(i_1)$ do
   ...
   for $p \in E(t)$ do in parallel
    $P(p)$
   end
 end
end
```

Figure 2.4 – *Nid de boucles transformées pour permettre l'exécution en parallèle des itérations de la boucle la plus interne*

Dans ce nid la boucle externe correspond à une itération. E(t) est l'ensemble des itérations calculées à l'itération t et P(p) représente l'ensemble des boucles exécutées en parallèles correspondant au vecteur d'itération p. La complexité du partitionnement repose sur la recherche d'une fonction de séquencement. Dans la recherche du partitionnement, il faut faire attention à ce que les dépendances soient respectées afin de garantir l'intégrité du calcul. Lamport [Lam74] propose de se restreindre au cas de fonctions de séquencement linéaire au point du domaine C'est-à-dire qu'il cherche à déterminer une famille d'hyperplans affines parallèles,$H(t)$ tel que l'ensemble des points calculés à l'instant t soit $E(t) = H(t) \cap D$ l'intersection de l'hyperplan et du domaine. Le passage d'un hyperplan à un autre $H(t) \to H(t+1)$ se fait par translation de l'hyperplan d'un vecteur π appelé vecteur de temps. Dans le cas d'une boucle uniforme, les vecteurs de dépendances sont positifs, il est aisé de construire un vecteur de temps solution. Il suffit pour cela de déterminer un vecteur π tel que pour tout vecteur de dépendance d on ait $\pi d \geq 1$. Les points p calculés à l'instant t forment l'hyperplan définit par $E(t) = \{p; Ap \leq b; [\pi p] = t\}$, et on peut calculer le temps de calcul total d'exécution de la boucle ainsi contractée en calculant t pour le dernier point du domaine $t_{execution} = \pi p_{max}$

Lamport propose de faire de la façon suivante, : soit $D = (d_1, ..., d_m)$ la matrice des vecteurs de dépendances, de taille $n \times m$ ou n est la profondeur du nid et m est le nombre de vecteurs de dépendances. On suppose D organisé selon l'ordre lexicographique croissant (de la dépendance la plus courte à la dépendance la plus longue). Soit k_1 la première composante non nulle de d_1 le premier vecteur de D. Comme d_1 est positif, $d_{1,k_1} \geq 0$. On pose π_{k_1} et $\pi_k = 0$ pour $k_1 \neq k \neq n$ .Soit maintenant k_2 l'indice de la première composante non nulle de d_2. Comme d_1 est lexico graphiquement plus petit que d_2 , $k_2 \leq k_1$. On pose $\pi_k = 0$ pour $k_2 < k < k_1$, et on prend pour π_{k_2} le plus petit entier positif tel que $\pi d_2 > 0$, en modifiant éventuellement π_{k_1} si $k_2 = k_1$. En réitérant le processus on obtient le vecteur de temps solution.

Pour des nids de boucles uniformes, plusieurs cas particuliers de la méthode de l'hyperplan ont été proposés, chacun d'eux privilégie un certain type de vecteur de temps.

2.2.4 Partitionnement des répétitions

Méthode de projection

Cette méthode propose de résoudre le problème du partitionnement des applications en utilisant l'algèbre linéaire. En effet, le but de cette méthode est de produire automatiquement un réseau systolique calculant un problème exprimé par un système d'équations récurrentes uniformes. L'idée générale est de profiter de la cyclicité et de l'uniformité des dépendances pour obtenir un réseau dans lequel la circulation de données est synchrone et formée de cellules modulaires. Les calculs sont répartis sur le réseau par une fonction d'allocation linéaire et séquencés par une fonction de temps.

Principe : La construction du réseau se fait par projection du domaine de calcul selon un vecteur s primitif choisi en fonction de la taille du réseau à générer et la position des entrées-sorties. Le choix du vecteur s doit également respecter les contraintes temporelles imposées par le vecteur temporel τ.

Le domaine de calcul de l'équation récurrente se fait dans le domaine D définissant le polyèdre $D = \{x | Ax \leq b\}$ dans Z^n.

Les dépendances sont représentées par une matrice correspondant aux vecteurs valuant les arrêtes du graphe réduit de l'équation récurrente : $D = (d_i, ..., d_n)$

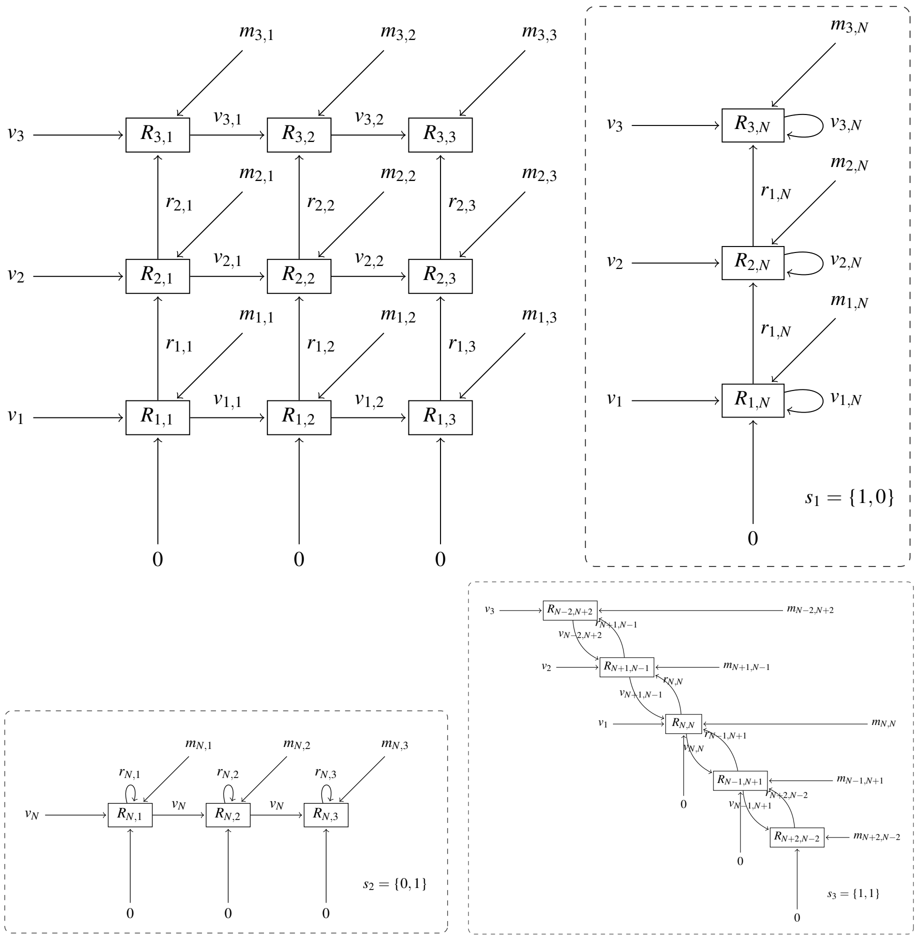


Figure 2.5 – *Exemple de domaine d'itération et plusieurs solutions de projection*

Méthode de pavage

La méthode de pavage (ou tiling) cherche à grouper plusieurs points de calcul du domaine d'itération afin de les exécuter de manière atomique. Cette technique s'assure que le calcul d'un block ne nécessite pas de communication afin de minimiser le coût des synchronisations entre blocs de calcul.

Principe : Cette méthode repose sur la construction d'une « tuile »dont la forme englobe plusieurs points du domaine d'itération. Une fois la forme de la tuile déterminée, celle-ci peut être mise à l'échelle afin d'optimiser le volume de calcul de chaque tuile et le volume de communication aux interfaces de la tuile. Cette tuile, dite tuile canonique est ensuite translatée par un vecteur de temps valide afin de paver l'intégralité du domaine.

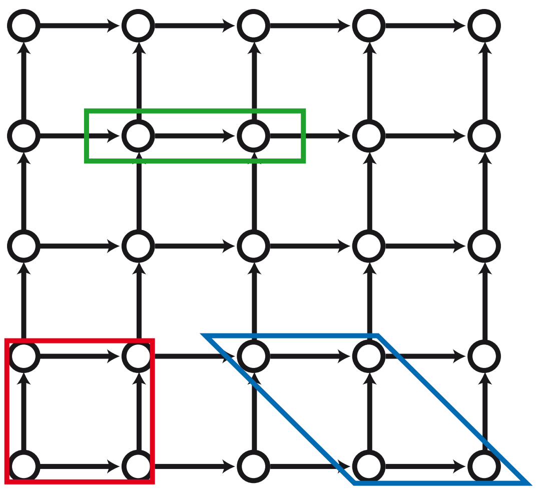


FIGURE 2.6 – *Exemple de domaine d'itération et plusieurs solutions de pavage.*

Chapitre 3

Contributions

3.1 Représentation hiérarchique dans le modèle SDF

Le modèle SDF dispose de plusieurs modes de représentation hiérarchique :

- Clustering : Ce mode de représentation hiérarchique s'attache à regrouper plusieurs acteurs d'une représentation SDF afin qu'ils puissent être exécutés de manière atomique. Cet ensemble d'acteurs est appelé cluster, et peut être représenté et traité comme un acteur unique. Ce regroupement s'appuie sur un ensemble de règles qui garantissent l'exécution atomique du cluster sans perturber le comportement de l'application. Cette hiérarchie part d'une représentation à plat pour en extraire des niveaux de hiérarchie.
- Parameterized SDF : Le modèle PSDF s'attache à décrire la structure d'applications dont le comportement de sous-systèmes varie à l'exécution. Dans ce modèle, la topologie de ces sous-systèmes est décrite à l'aide de facteur de production/consommation et d'acteurs paramétrés. Ces paramètres sont résolus à l'exécution par l'intermédiaire de deux acteurs spécifiques. L'acteur *init* est exécuté pour chaque instanciation du sous-système et il peut affecter les paramètres qui conditionnent l'ordonnancement du réseau parent du sous-système. L'acteur sub_{init} peut affecter les paramètres qui n'affectent que la topologie du sous-système et/ou le comportement des acteurs. Ce type de hiérarchie s'appuie sur la sémantique du modèle SDF tout en ajoutant une couche de paramétrage du modèle qui est résolu à l'exécution. Ce paramétrage augmente l'expressivité du modèle tout en augmentant sa complexité et en diminuant sa vérificabilité.

Ces deux modèles permettent de spécifier de manière hiérarchique une application. Le clustering construit automatiquement la représentation hiérarchique en partant d'une représentation à plat de l'application. Le modèle PSDF permet de décrire des applications dont le comportement de certains sous-systèmes peut varier à l'exécution. Nous proposons un modèle qui respecte la sémantique SDF et son caractère statique, tout en permettant la spécification hiérarchique de l'application (voir [PBR09]). Cette hiérarchie ne serait pas extraite d'une représentation à plat comme dans le clustering, mais permettrait au concepteur de raffiner une représentation gros grain en spécifiant le comportement de certains acteurs au travers d'une représentation SDF. Afin de permettre cette spécification, il est nécessaire de définir les propriétés du modèle permettant la composition de spécifications SDF. Ce modèle s'appuie sur l'ajout d'interfaces aux sous-systèmes SDF pour garantir la cohérence entre les niveaux de hiérarchie. Le comportement de ces interfaces s'appuie sur un ensemble de règles qui garantissent l'atomicité du sous-système et qui assurent l'indépendance en terme d'ordonnancement des différents niveaux de hiérarchie.

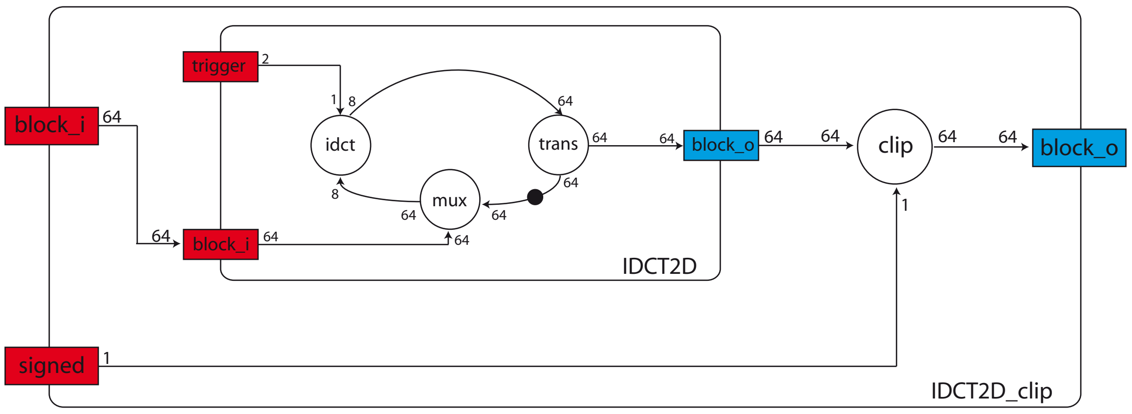


FIGURE 3.1 – *Exemple d'application d'IDCT2D utilisant le modèle hiérarchique*

Cette hiérarchie permet d'impliquer le designer dans le processus de composition de la hiérarchie et ainsi de structurer cette hiérarchie selon la hiérarchie existant dans la sémantique des données et/ou des opérations. De plus, ce type de hiérarchie autorise des optimisations mémoires et opératoires basées sur les choix du designer. Enfin, la hiérarchie spécifiée par l'utilisateur peut être utilisée pour générer le code de l'application dans une forme structurée et facilement interprétable par le designer.

3.2 Génération de code pour le prototypage rapide

Une des contribution de cette thèse consiste à générer du code multiprocesseur à partir du modèle de hiérarchie décrit précédemment (voir [PBPR09]). Cette génération de code cible les architectures multi-coeur embarqués grâce à une allocation mémoire statique et une empreinte de code réduite. Cette génération de code est implémentée dans l'outil PREESM développée depuis 2007 à l'Institut d'Electronique et de Télécommunication de Rennes dans l'équipe image et télédétection. Cet outil s'appui sur l'environnement de développement Eclipse pour proposer un outil de prototypage rapide couvrant les étapes de spécification de l'application (plugin Graphiti), spécification de l'architecture, prototypage automatique et génération de code. Le prototypage de l'application sur l'architecture spécifié ce fait par une suite de traitements séquentiels sur les informations d'entrée pour terminer par générer du code multiprocesseur compilable pour les différents processeurs/coeurs de l'architecture. Cette suite de traitement est dirigée par l'utilisateur, par l'intermédiaire d'un workflow qui spécifie les traitements à appliquer, sélectionnés dans une bibliothèque, et les dépendances de données entre traitement. La génération de code est la dernière phase de cette suite de traitement et vise à produire du code optimisé pour l'architecture dans un format compréhensible par l'utilisateur. Cette génération de code implémente un modèle d'exécution dans lequel chaque processeur/coeur exécute plusieurs processus légers synchronisés par un protocole basé sur l'utilisation de sémaphores. Un processus léger exécute l'ordonnancement des acteurs sur le processeur. Pour chaque bus de communication utilisé, un processus léger gère l'ordonnancement des communications. Le processus de calcul et synchronisé aux différents processus de communication par des sémaphores indiquant pour un envoi la disponibilité de la donnée et pour l'envoi, et la disponibilité de la donnée pour le calcul, et pour une réception de données, la disponibilité de la donnée pour le calcul, et la disponibilité de la donnée pour la réception (voir figure 3.2).

Cette génération se découpe en plusieurs phases :

Ce découpage en phase permet la séparation des problématiques pour traiter chacune d'elles de manière optimale. Pour une meilleure adaptabilité, la phase d'allocation mémoire bénéficie d'une bibliothèque extensible de politiques d'allocation permettant de s'adapter aux contraintes mémoire de l'application et d'évaluer différents algorithmes d'optimisation mémoire.

La génération de code produit un code générique dans une syntaxe xml qu'il est ensuite possible de transformer en une syntaxe spécifique pour cibler l'architecture en utilisant la technologie XSLT (Xml Syntax Language Transformation). Cette transformation utilise trois types de ressources :

- Transformations spécifiques au Système d'exploitation : transformations spécifiques aux appels systèmes et primitives de synchronisation.
- Transformations spécifiques à l'architecture : transformations spécifiques à l'utilisation des ressources matérielles de l'architecture.
- Transformations spécifiques au langage : transformations spécifiques à la syntaxe du langage.

Le code ainsi généré cible un composant de l'architecture et cette génération de code peut facilement être étendue pour supporter d'autres langages, systèmes

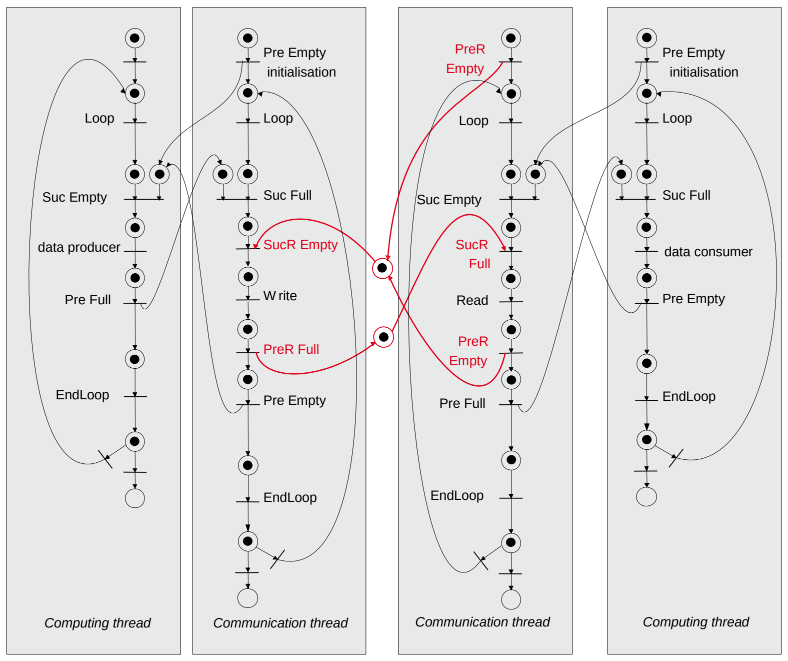


Figure 3.2 – *Réseau de petri symbolisant la synchronisation des processus de calcul et de communication*

d'exploitation, architectures.

3.3 Optimisation des nids de boucles dans le modèle SDF

La spécification d'application en utilisant le modèle de hiérarchie basée sur les interfaces décrit précédemment, permet une spécification compacte et factorisée. Cette compacité présente, facilite la saisie par l'utilisateur tout en permettant la même compacité dans la génération de code. Le principal inconvénient de ce modèle réside dans le fait qu'une majorité du parallélisme existant dans l'application reste enfoui dans les niveaux de hiérarchie et ne peut être exploité par l'allocation. Pour pallier à ce défaut, il est possible d'effectuer une mise à plat de la hiérarchie et ainsi remonter le parallélisme enfoui au plus haut niveau, le rendant ainsi exploitable par l'allocation. Cependant, dans le cadre de motifs répétés un grand nombre de fois, cette mise à plat de la hiérarchie, s'accompagne d'une explosion du nombre d'acteur à allouer/ordonnancer. La complexité de l'allocation/ordonnancement dépendant principalement du nombre d'acteur, l'augmentation du nombre de sommet

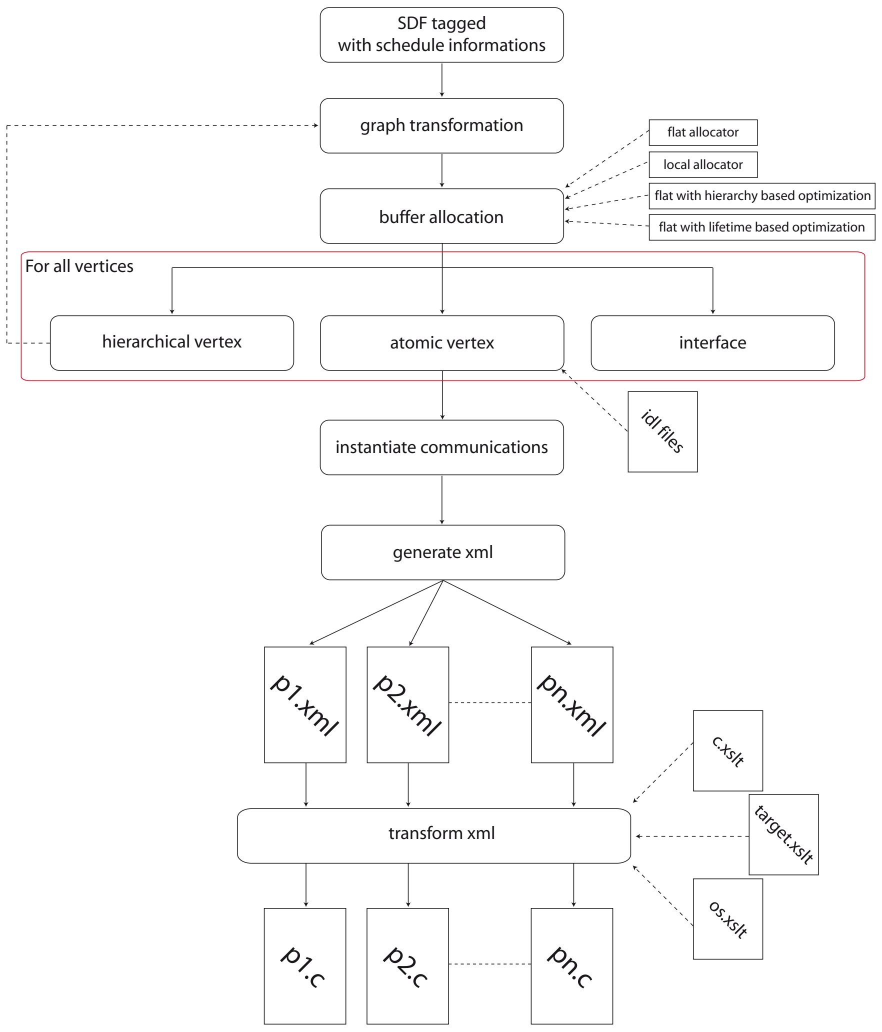


Figure 3.3 – *Etapes de la génération de code*

s'accompagne d'une augmentation de la complexité et donc du temps nécessaire à cette étape. Lors de cette thèse, ont été développées des techniques de partitionnement des motifs répétitifs dans le modèle SDF en utilisant l'analogie existante avec la structure des motifs de boucles imbriqués (voir [PBR10]).

Dans un premier temps, il a été nécessaire de déterminer comment extraire les vecteurs de distance d'itération de la représentation hiérarchique. L'extraction de ces vecteurs repose sur l'interprétation des jetons de données initialement présent sur les arcs. Une fois ces vecteurs extraits, les techniques de projection et pavage peuvent être utilisées. Lors de cette thèse, il a également été déterminé comment générer le nouveau réseau d'acteurs à partir des données de pavage ou de projection. Ces deux techniques permettent toutes deux d'extraire du parallélisme de la représentation hiérarchique. Cependant la technique de pavage permet plus de flexibilité. En effet, pour la technique de projection, la taille du réseau d'acteur produit, dépend des dimensions du domaine d'itération. Dans le cas de la technique de pavage, la taille du réseau produit, dépend uniquement de la taille de la tuile de pavage choisie.

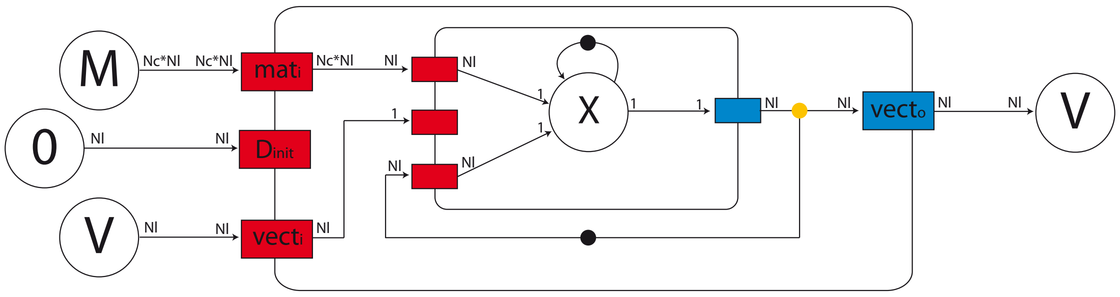


FIGURE 3.4 – *Exemple de produit matrice vecteur en présentation SDF factorisée*

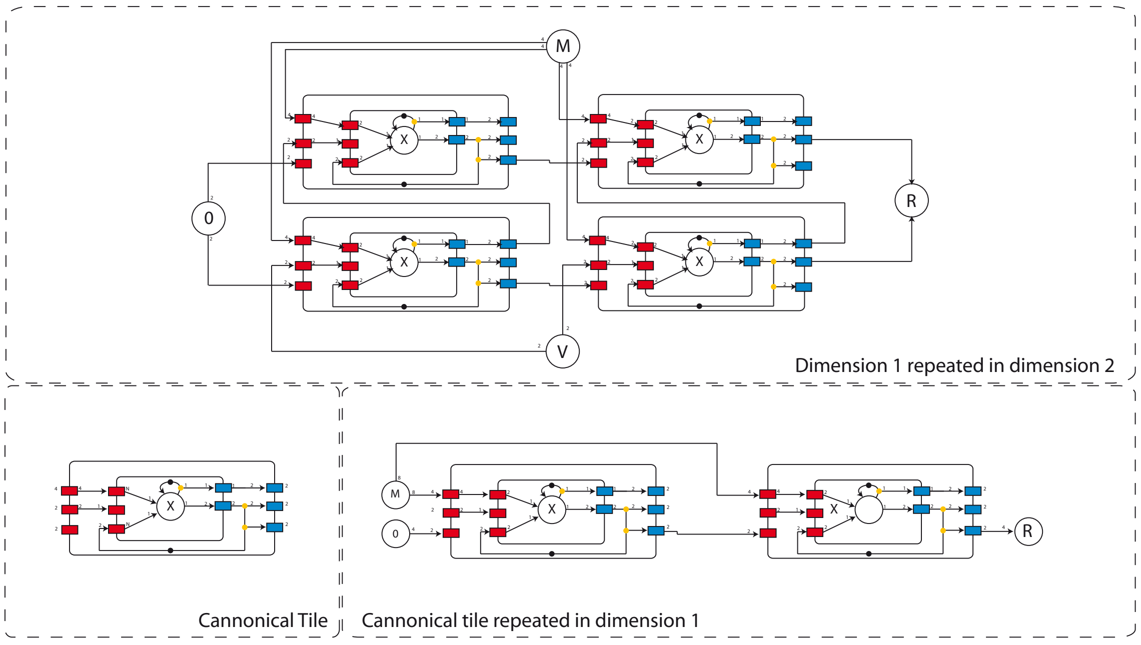

M
0
V
R
X
Dimension 1 repeated in dimension 2
Cannonical Tile
Cannonical tile repeated in dimension 1

FIGURE 3.5 – *Procédure de pavage du produit matrice vecteur*

Chapitre 4

Conclusion et propositions de travail

Le travail effectué lors de cette thèse contribue majoritairement au logiciel PREESM. Le modèle de hiérarchie présenté, permet la spécification de motifs factorisés dans une approche descendante. Cette approche autorise le concepteur à décrire le comportement de l'application à gros grain pour ensuite raffiner cette description. Des méthodes de transformation telles que celles présentées permettent d'extraire le parallélisme existant à grain fin. Ces techniques capitalisent sur les méthodes existantes pour les transformations de boucles imbriquées afin de générer une description exprimant du parallélisme de manière optimale. Afin de couvrir l'intégralité de la chaine de prototypage, j'ai aussi développé la génération de code tirant partie du modèle de description hiérarchique pour produire une description factorisée et tirant partie de la hiérarchie pour optimiser l'allocation mémoire.

Les résultats de cette thèse peuvent être appliqués dans d'autres domaines flux de données et notamment au langage CAL. En effet, il serait possible d'extraire des comportements SDF au sein de la description CAL et ensuite leur faire bénéficier des transformations proposées. Le modèle SDF étant purement statique, il pourrait aussi être intéressant d'utiliser le modèle PSDF dans le cadre du prototypage rapide. Ce modèle est en cours d'implémentation dans l'environnement PREESM et pourrait bénéficier des transformations proposées dans cette thèse et également d'une génération de code optimisée.

Une autre perspective serait de supporter des modèles de workflow itératifs dans PREESM et ainsi proposer au concepteur d'automatiser la procédure de prototypage rapide pour générer une implémentation de l'application sur l'architecture donnée qui réponde à des critères quantitatifs donnés.

En définitive le travail développé dans cette thèse s'inscrit dans une construction logique pour l'implémentation d'une chaîne de prototypage rapide. De plus, ce travail est supporté par une implémentation open-source. Ce logiciel est notamment utilisé par l'entreprise Texas Instrument pour le prototypage d'applications multi-coeurs pour plateformes multi-dsp. Nous espérons, dans l'avenir, attirer d'autres utilisateurs et contributeurs afin d'améliorer cet outil.

Part II

Background

Chapter 5

Introduction

5.1 Overview

Since applications such as video coding/decoding or digital communications with advanced features are becoming more complex, the need for computational power is rapidly increasing. Telecom standards such as Long Term Evolution (LTE) bring the computing requirement of the base station at a high level. Applications such as Reconfigurable Video Coding aims at creating a video codec with multi-standard capabilities, thus copping with multiple levels of complexity and time constraints.

In order to satisfy software requirements, the use of parallel architectures is a common answer. Depending on the target application, those parallel architectures can make use of a wide variety of processor cores and communication media. On one hand, in mobile applications, the power requirement being tight, System On Chip (SOC) designers prefer to make use of specialized cores that give the best performance for a reduced set of instructions. On the other hand one might want to use general purpose cores, that handle most of the applications with an average power/performance ratio. Thus some architectures make use of heterogeneous core types to best fit any application requirements. For example mobile applications usually make use of general purpose cores to handle most user interfaces and basic functions, and use DSP core or GPU for signal processing, video decoding or gaming. Communication media are of great importance on multi-core architectures, as most of the parallel performance rely on the ability cores have to exchange data at fast rates. Multiple kind of interconnection topologies exist, each fitting a given purpose at a given cost. The multiplicity of cores and interconnections greatly increases the complexity and time necessary to port an application on a given architecture. Moreover, once porting of the application has been realized, one cannot benefit of it to target a new architecture and has to start from scratch.

To reduce the software development efforts for such architectures, it is necessary to provide the programmer with efficient tools capable of automatically solving communications and software partitioning/scheduling concerns. The complexity introduced by the wide range of available architectures and the increasing number of cores make the optimal implantation of an application hard to reach by hand. Moreover the stability of the application has to be early proved in the development stage to ensure the reliability of the final product. Tools such as PeaCE [SOIH97], SynDEx [GS03] or PREESM [PRP+08] aim at providing solutions to problems described earlier, by assisting the designer through automated steps leading to a reliable final prototype in a short time. This kind of tools is called rapid prototyping frameworks.

Most of these tools use as an entry point a model of the application associated to a model of the architecture. Data flow model is indeed a natural representation for data-oriented applications since it represents data dependencies between the operations allowing to extract parallelism. In this model the application is described as a graph in which nodes represent computations and edges carry the stream of data-tokens between operations. A well known data flow model is the Synchronous Data Flow (SDF) that enables to specify the number of tokens produced/consumed on each outgoing/incoming edges for one firing of a node. Edges can also carry initialization tokens, called delay. That information allows to perform analysis on the graph to determine whether or not there exist a schedule of the graph, and if so to determine an execution order of the nodes and application's memory requirements at compile-time. In basic SDF representation, hierarchy is used either as a way to represent clusters of nodes in the SDF graph or as parameterized sub-systems. One contribution of this thesis is to describe a hierarchy type allowing the designer to describe sub-graph in a classical top down approach. Relevant information can also be extracted from this representation in order to ease the graph scheduling and to lead to a better implementation. The goal of tools such as PREESM is to provide the programmer with a multi-core implementation of the application in a language, a compiler or synthesizer can take. Another contribution of this thesis is to describe the generation of the C code that implements the computed multi-processor schedule in PREESM. The described C code generation method presents a high level of flexibility for the users and generates optimized C code in a comprehensible format.

When dealing with multi-dimensional problems such as image video encoding/decoding, the use of nested loops is a relevant answer to the application modeling. This kind of loop structure is used to hierarchically describe an algorithm, each nested level of hierarchy representing the application in a lower dimension space. For example, an image encoding application can be described at the picture level as a loop over the lines of the image. The line level is itself described as a loop over the blocks of the line, and the block level as a loop over the pixels of the blocks. This kind of representation can expose little parallelism from the hierarchy point of view, but a lot at the lowest hierarchy. In order to find a trade-off between too much and not enough parallelism extraction from the application description, PREESM integrates an SDF graphs transformation step that aims at extracting a given level of parallelism in concordance with available parallelism on the target architecture. This step also aims at decreasing the scheduling complexity by providing a factorized representation of the application. This step can take advantage of nested loops transformations algorithm inspired by methods such as iteration domain projection and iteration domain tiling to extract a given level of parallelism.

This thesis also led to other contributions in the domain of video coding. The image group inside the IETR lab, is highly involved into still image and video coding/decoding applications. Recently the team got involved into the MPEG-RVC standard, and as a consequence I was involved into related work. The first contribution concerns bit-stream validation for RVC. I was involved into the programming of the validation tool, and I contributed to two publications [LPM09, RPLM08] on the bit-stream validation procedure. I also helped at porting a reconfigurable video decoder on an ARM architecture. This reconfigurable video decoder [GWP+10] makes use of LLVM (Low Level Virtual Machine) to implement a decoder reconfigurable using description of a CAL network. Those other contributions will not be described in depth in this thesis but an overview of their context is given in the following.

5.2 Contributions of this Thesis

5.2.1 Rapid Prototyping Framework

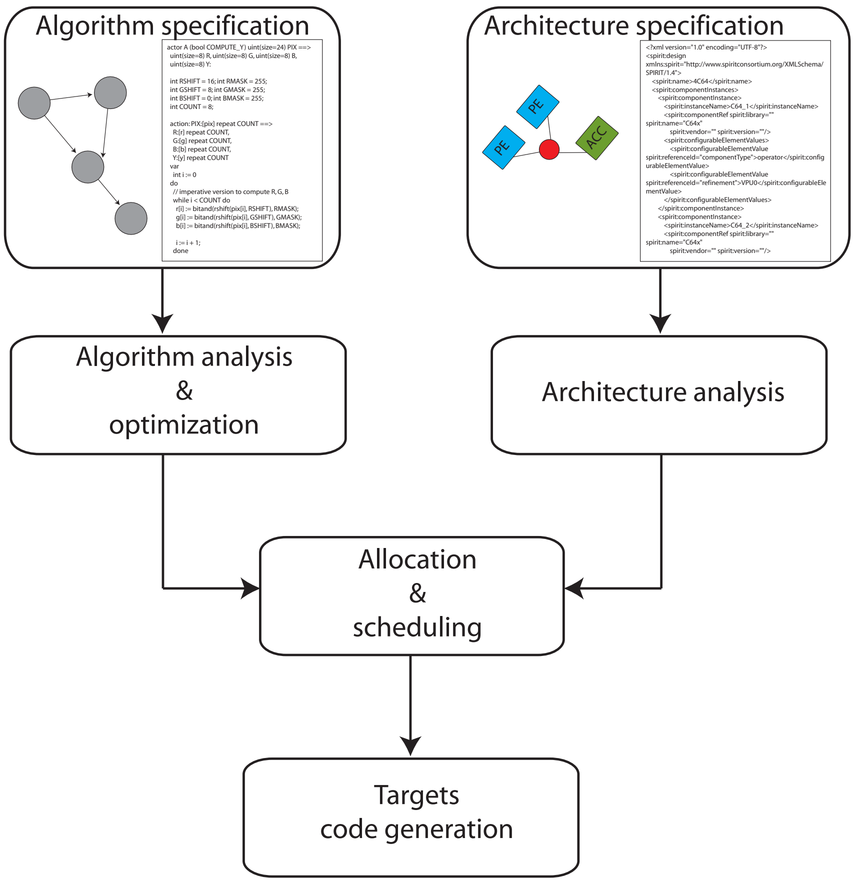
The recent evolution of digital communication systems (voice, data and video) has been become more and more complex. Over the last two decades, low data-rate systems (such as dial-up modems, first and second generation cellular systems, 802.11 Wireless local area networks) have been replaced or augmented by systems capable of data rates of several Mbps, supporting multimedia applications (such as DSL, cable modems, 802.11b/a/g/n wireless local area networks, 3G, WiMax and ultra-wideband personal area networks).

As communication systems have evolved, the resulting increase in data rates has necessitated a higher system algorithmic complexity. A more complex system requires greater flexibility in order to deal with different protocols in different environments. Additionally, there is an increased need for the system to support multiple interfaces and multi-component devices. Consequently, this requires the optimization of device parameters over varying constraints such as performance, area and power. Achieving this device optimization requires a good understanding of the application complexity and the choice of an appropriate architecture to support this application.

An embedded system commonly contains several processor cores in addition to hardware co-processors. The embedded system designer needs to distribute a set of signal processing functions onto a given hardware with predefined features. The functions are then executed as software code on target architectures ; this action will be called a deployment in this paper. A common approach to implement a parallel algorithm is the creation of a program containing several synchronized threads in which execution is driven by the scheduler of an operating system. Such implementation does not meet the hard timing constraints required by real-time applications and the memory consumption constraints required by embedded systems [Lee06]. One-time manual scheduling developed for single-processor applications is also not suitable for multiprocessor architectures : manual data transfers and synchronizations quickly become very complex, leading to a waste of time and potential deadlocks. Furthermore, the task of finding an optimal deployment of an algorithm mapped onto a multi-component architecture is not straightforward. When performed manually, the result is generally a sub-optimal solution. These issues raise the need for new methodologies, which allow the exploration of several solutions, to achieve a more optimal result.

Several features must be provided by a fast prototyping process : description of the system (hardware and software), automatic mapping/scheduling, simulation of the execution and automatic code generation. Based on [PAN08][PBRP09][PMAN09] a more complete rapid prototyping framework was created. This complete framework is composed of three complementary tools based on Eclipse [ecl] that provide a full environment for the rapid prototyping of real-time embedded systems : Parallel and Real-time Embedded Executives Scheduling Method (PREESM), Graphiti and Synchronous Data Flow for Java (SDF4J). This framework implements the methodology Algorithm-Architecture Matching (AAM) [GS03]. The focus of this rapid prototyping activity is currently static code mapping/scheduling but dynamic extensions are planned for future versions of the tool. Such tools rely on a design flow which takes an algorithm representation and an architecture representation to then go through algorithm transformations, architecture analysis, allocation and scheduling, to finally generate code adapted to the target as depicted in 5.1.

From the graph descriptions of an algorithm and of an architecture, PREESM can find the right deployment, can provide simulation information and can generate a framework code for the processor cores [PAN08]. These rapid prototyping tasks can be combined and parameterized in a *workflow*. In PREESM, a workflow is defined as an oriented graph representing the list of rapid prototyping tasks to



```
actor A (bool COMPUTE_Y) uint(size=24) PIX ==>
  uint(size=8) R, uint(size=8) G, uint(size=8) B,
  uint(size=8) Y:

  int RSHIFT = 16; int RMASK = 255;
  int GSHIFT = 8; int GMASK = 255;
  int BSHIFT = 0; int BMASK = 255;
  int COUNT = 8;

  action: PIX:[pix] repeat COUNT ==>
   R:[r] repeat COUNT,
   G:[g] repeat COUNT,
   B:[b] repeat COUNT,
   Y:[y] repeat COUNT
  var
   int i := 0
  do
   // imperative version to compute R, G, B
   while i < COUNT do
    r[i] := bitand(rshift(pix[i], RSHIFT), RMASK);
    g[i] := bitand(rshift(pix[i], GSHIFT), GMASK);
    b[i] := bitand(rshift(pix[i], BSHIFT), BMASK);

    i := i + 1;
   done
```

```
<?xml version="1.0" encoding="UTF-8"?>
<spirit:design
xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/
SPIRIT/1.4">
  <spirit:name>4C64</spirit:name>
  <spirit:componentInstances>
    <spirit:componentInstance>
      <spirit:instanceName>C64_1</spirit:instanceName>
      <spirit:componentRef spirit:library=""
spirit:name="C64x"
        spirit:vendor="" spirit:version=""/>
      <spirit:configurableElementValues>
        <spirit:configurableElementValue
spirit:referenceId="componentType">operator</spirit:config
urableElementValue>
        <spirit:configurableElementValue
spirit:referenceId="refinement">VPU0</spirit:configurableEle
mentValue>
      </spirit:configurableElementValues>
    </spirit:componentInstance>
    <spirit:componentInstance>
      <spirit:instanceName>C64_2</spirit:instanceName>
      <spirit:componentRef spirit:library=""
spirit:name="C64x"
        spirit:vendor="" spirit:version=""/>
```

Figure 5.1: *A Rapid Prototyping Frame (RPF) work design flow.*

execute on the input algorithm and architecture graphs in order to determine and simulate a given deployment. A rapid prototyping process in PREESM consists of a succession of transformations. These transformations are associated in a data-flow graph representing a workflow that can be edited in a Graphiti generic graph editor. The PREESM input graphs may also be edited using Graphiti. The PREESM algorithm models are handled by the SDF4J library. The framework can be extended by modifying the workflows or by connecting new plug-ins (for compilation, graph analyses, and so on).

There exist numerous solutions to partition algorithms onto multi-core architectures. If the target architecture is homogeneous (all cores of the architectures share the same properties), several solutions exist which generate multi-core code from

C with additional information (OpenMP [opeb], CILK [BJK$^+$95a]). In the case of heterogeneous architectures, languages such as OpenCL [opea] and the Multicore Association Application Programming Interface (MCAPI [mca]) define ways to express parallel properties of a code. However, they are not currently linked to efficient compilers and runtime environments. Moreover, compilers for such languages would have difficulty in extracting and solving the bottlenecks of the implementation that appear inherently in graph descriptions of the architecture and the algorithm.

The Poly-Mapper tool from PolyCore Software [pol] offers similar functionalities to PREESM but, in contrast to PREESM, its mapping/scheduling is manual. Ptolemy II [Lee01] is a simulation tool that supports many models of computation. However, it also has no automatic mapping and currently its code generation for embedded systems focuses on single-core targets. Another family of frameworks existing for data flow based programming is based on CAL [EJ03b] language like OpenDF [BBE$^+$08]. OpenDF employs a more dynamic model than PREESM but its related code generation does not currently support efficient automatic mapping on multi-core embedded systems.

Closer to PREESM are the Model Integrated Computing (MIC [KSLB03]), the Open Tool Integration Environment (OTIE [Bel06]), the Synchronous Distributed Executives (SynDEx [GLS99]), the Dataflow Interchange Format (DIF [HKK$^+$04]), and SDF for Free (SDF3 [SGB06]). Both MIC and OTIE can not be accessed online. According to the literature, MIC focuses on the transformation between algorithm domain-specific models and metamodels, while OTIE defines a single system description that can be used during the whole signal processing design cycle.

DIF is designed as an extensible repository of representation, analysis, transformation and scheduling of data-flow language. DIF is a Java library which allows the user to go from graph specification using the DIF language to C code generation. However, the hierarchical Synchronous Data Flow (SDF) model used in the SDF4J library and PREESM is not available in DIF.

SDF3 is an open source tool implementing some data-flow models and providing analysis, transformation, visualization, and manual scheduling as a C++ library. SDF3 implements the Scenario Aware Data Flow (SADF [TGS$^+$08]), and provides Multiprocessor System-on-Chip (MP-SoC) binding/scheduling algorithm to output MP-SoC configuration files.

SynDEx and PREESM are both based on the AAM methodology [GS03] but the tools do not provide the same features. SynDEx is not open source, has its own model of computation that does not support schedulability analysis and target code generation is possible but not provided with the tool. Moreover, the architecture model of SynDEx is at a too high level to take into account bus contentions and DMA used in modern chips (multi-core processors of MP-SoC) in the mapping/scheduling.

The features that differentiate PREESM from the related works and similar tools are :

- the tool is an open source and accessible online [1],
- the algorithm description is based on a single well-known and predictable model of computation,
- the mapping and scheduling are totally automatic,
- the functional code for heterogeneous multi-core embedded systems can be generated automatically,
- the algorithm model provides a helpful hierarchical encapsulation thus simplifying the mapping/scheduling [PBPR09].

5.2.2 Reconfigurable Video Coding

MPEG has produced several video coding standards over the time (MPEG-1, MPEG-2 , MPEG-4 ...). The specification of the standards usually relies on a monolithic implementation using C/C++ imperative language. This specification lacks flexibility, and does not allow to use the combination of coding algorithms from different standards enabling to achieve specific design or performance trade-offs and thus fill, case by case, the requirements of specific applications.

RVC is an MPEG standard that aims at specifying coder/decoder by combining together Functionnal Units (FUs) modeling video coding tools, from the MPEG standard library written in CAL. Those FUs are extracted from existing MPEG standards (MPEG-2, MPEG-4, MPEG-4 AVC, ...). The RVC framework includes :

- The standard Video Tool Library is a repository of video coding tools. Those tools are described using RVC-Cal, that specify the algorithmic behavior and the inputs/outputs of the unit.
- The Functional unit Network Language aims at describing the decoder configuration as a network of FUs from the VTL.
- The MPEG-21 Bitstream Description Language provides an xml syntax that can be used in a schema to describe the bitsream structure that a RVC decoder as to decode. A BSDL schema may be used to generate the parser to be used in the decoder network as a RVC-CAL actor.

The RVC standard relies on the CAL Actor Language for actor specification. This data flow language was developed as a sub-project of the ptolemy project at the university of California at Berkeley. A subset of the CAL language called RVC-CAL was adopted as a specification language for the FUs of the MPEG RVC library. This language allows low level actor behavior specification as CAL actor

1. http://preesm.sf.net

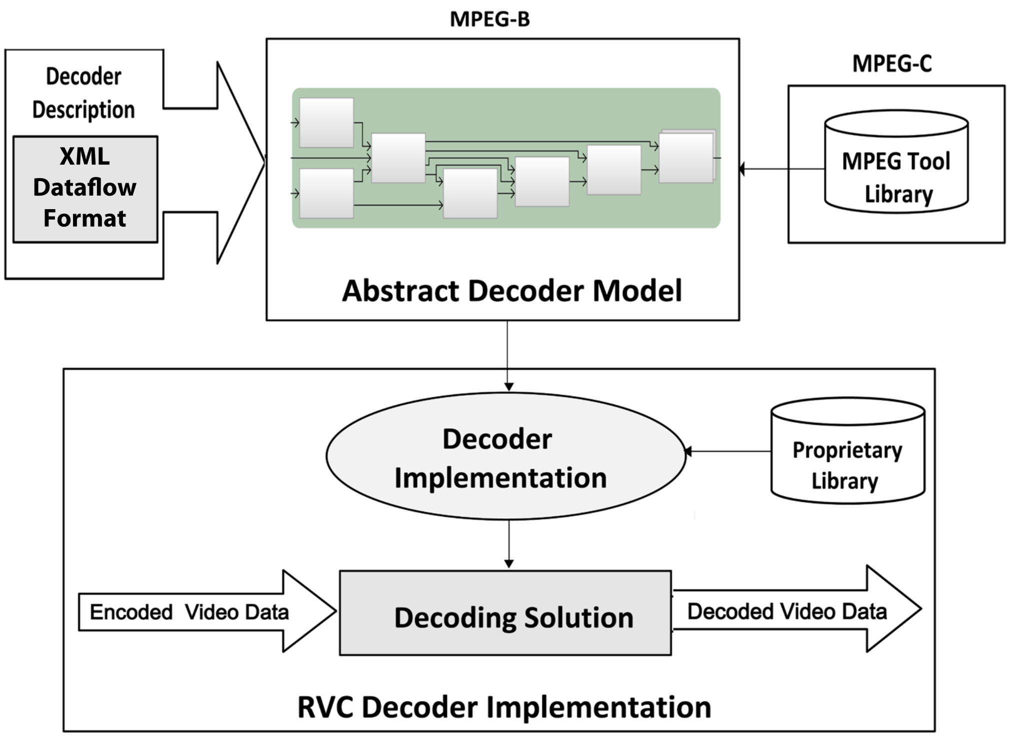


Figure 5.2: *The reconfigurable Video Coding Framework.*

that is a computational entity with input and output ports, states and parameters. An actor behavior is described through multiple actions that define the computation performed. An action consumes sequences of input tokens, and produces sequences of output tokens. Actor can communicate together by sending and receiving tokens on a CAL actors network.

5.2.3 Bit-Stream Description Language

MPEG-B part 5 is an ISO/IEC international standard that specifies BSDL [Int] (Bitstream Syntax Description Language), a language based on XML Schema aiming at describing the structure of a bitstream with an XML document named BS Description. For instance, in the case of a MPEG-4 AVC video codec [iso04], a BS Schema describes the structure common to all possible conformant MPEG-4 AVC video bitstreams, whereas a BS description describes a single MPEG-4 AVC encoded bitstream as an XML document. Figure 5.3 shows the BSDL Schema associated with the BSDL description in Figure 5.4. BSDL uses XML to describe the structure of video coded data. An encoded video bitstream can be described as a sequence of binary symbols of arbitrary length – some symbols contain a single bit, while others contain many bytes. For these binary symbols, the BSDL description indicates values in a human – and machine – readable format for example, using hexadecimal values (as for startCode in Figure 5.3), integers, or strings. It also organizes the symbols into a hierarchical structure that reflects the data semantic interpretation.

In other words, the BSDL description level of granularity can be fully customized to the application requirements [TKJM+07]. BSDL was originally conceived and designed to enable adaptation of scalable multimedia content in a format-independent manner [J. 07]. In the RVC framework, BSDL is used to fully describe the entire bitstream – each elementary bit has its corresponding value in a Variable Length Decoding (VLD) table. As a result, the corresponding BS schema must specify all components of the syntax at a finer granularity level than the ones developed and used for adaptation of scalable content. In this context BSDL does not replace the original data, but instead provides additional information (or metadata) to support an application for parsing and processing the binary content. Finally, BSDL does not mandate the names of the elements in the BSDL Description; the application assigns names that provide meaningful semantics for the description at hand. Figure 5.3 is an example of a BSDL description for video in MPEG-4 AVC format.

In the RVC framework, BSDL is preferred over Flavor [Ele97] because:

- it is stable and already defined by an international standard;
- the XML-based syntax well integrates the XML syntax used to describe the configuration of the RVC decoder; constituted by the instantiation of FUs from the toolbox and by their connettivity
- the RVC bitstream parser may be easily derived by transforming the BSDL schema using standard tools (e.g. XSLT).

The RVC framework aims at supporting the development of new MPEG standard and new decoding solutions. The flexibility offered by the standard video coding library to explore rapidly the design space is primordial. Defining coding tools and their interconnections becomes a relatively easy task if compared to the SW rewriting efforts need to modify (usually very large) monolithic specifications. However, testing new decoding solutions, new algorithms for new coding tools, or new tools configurations, the bitstream syntax may change from a solution to another. The consequence is that a new parser need to be rewritten for each new bitstream syntax. The parser FU is the most complex actor in the MPEG-4 SP decoder [iso04] described in [LMTKJ07] and its behavior need to be validated versus all possible conformant bistreams. This is equivalent to validate it using the BSDL schema for the syntax at hand. Moreover, it is certainly not a good idea to have to write it by hand when a systematic solution for deriving such parsing procedure from the BSDL schema itself could be developed.

```
<NALUnit>
  <startCode>00000001</startCode>
  <forbidden0bit>0</forbidden0bit>
  <nalReference>3</nalReference>
  <nalUnitType>20</nalUnitType>
  <payload>5 100</payload>
</NALUnit>
<NALUnit>
<startCode>00000001</startCode>
<!-- and so on... -->
</NALUnit>
```

Figure 5.3: *BS description fragment of an MPEG-4 AVC bitstream.*

```
<element name="NALUnit"
  bs2:ifNext="00000001">
<xsd:sequence>
  <xsd:element name="startCode" type="avc:hex4" fixed="00000001"/>
  <xsd:element name="nalUnit" type="avc:NALUnitType"/>
  <xsd:element ref="payload"/>
</xsd:sequence>
<!--  Type of NALUnitType -->
<xsd:complexType name="NALUnitType">
  <xsd:sequence>
    <xsd:element name="forbidden_zero_bit" type="bs1:b1" fixed="0"/>
    <xsd:element name="nal_ref_idc" type="bs1:b2"/>
    <xsd:element name="nal_unit_type" type="bs1:b5"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="payload" type="bs1:byteRange"/>

<!-- and so on... -->
```

Figure 5.4: *BS schema fragment of MPEG-4 AVC codec.*

5.3 Outline of this Thesis

The organization of this thesis is as follows:

– Part II groups all the useful information for the comprehension of Part III. In chapter 6 we give an overview of the parallel systems specifications, then we present the data flow model of computation and its child models. Finally we give important notion of multi-processor scheduling, and present an overview of the existing rapid prototyping tools. Chapter 7 is a state of the art on nested loops representation and partitioning techniques.

– Part III presents the contributions starting with Chapter 8 in which we present existing hierarchy model in the synchronous data flow model and a new hierarchy model. In chapter 9 this hierarchy model is used for clean and efficient code generation. The Chapter 10 shows how the nested loops partitioning techniques described in Chapter 7 can be efficiently used in the Synchronous data flow context. Chapter 11 concludes by giving current status of the research work and directions for future work.

Chapter 6

Background and related work

6.1 Introduction

As applications such as video coding/decoding or digital communications with advanced features (MIMO, Beamforming, Equalization, etc...) are becoming more complex, the need for computational power is rapidly increasing. In order to satisfy software requirements, the use of parallel architecture is a common answer. To reduce the software development effort for such architectures, it is necessary to provide the programmer with efficient tools capable of automatically solving communications and software partitioning/scheduling concerns. Such tools rely on an application model that allows compile-time optimizations and ensures the application to behave well at run-time.

In the following, we will give a brief overview of existing multi-processor architectures. Then an extensive study of the data flow model of computation will introduce the reader to the existing models and particularly the Synchronous Data Flow model. In a third section, we will explore the basics of multi-processor scheduling techniques, finally in a last section we will present existing tools for application modeling.

6.2 Parallel computing systems

A parallel computing system is a computing system whose computing resources are spatially and/or logically divided into subsystems. Such systems were designed to overcome the frequency limits imposed by the building technology and power consumption. A parallel architecture allows to increase the computing performance with a ratio depending on the degree of potential parallelism of a given application. Digital signal processing algorithms are usually good candidates for parallelization due to the multi-dimensional aspect of the applications. The multi-core architecture is today the most popular architecture but other architectures such as clusters and systolic arrays also exist. Multi-cores tend to become more and more complex using specialized communication networks, accelerators and heterogeneous cores. Such architectures can be classified based on different criteria :

- **Target application**

  Applications for multi-core architectures can be of two kinds : data processing dominated or control dominated. The data processing dominated applications are usually data-flow applications that expose some parallelism and require high throughput and performance to handle a large amount of data. Many of those applications have relatively flexible time constraints (not time critical) but hard power constraints when embedded into mobile target. The control processing applications are mostly event driven application with hard time constraints (time critical). The multi-core architecture in that case can provide low response time.

- **Power/Performance**

  Multi-core architecture provides the designer with scalability in both power and performance domain with tight relations between the two. Growth of cores inside a component usually allows to downscale the frequency of the processing elements, thus lowering the power consumption.

- **Processing Elements**

  Depending on the context of the application, the processing elements involved may vary. Processing elements (see Figure 6.1) are characterized by their Instruction Set Architecture (ISA) that defines the instructions supported, and their micro-architecture that defines the way the ISA is implemented. The ISA can usually be classified into two categories, the Complex Instruction Set Computer (CISC), and Reduced Instruction Set Computer (RISC). While the later results in larger code size, both behave the same after instruction decoding. For parallel architecture, the ISA is usually augmented with synchronization primitives. The efficiency of the PEs greatly resides in the micro-architecture. The micro-architecture can range from a simple in-order processing element to multiple pipelines with out-of-order execution to get the better trade-off between power consumption and computing performance. Simple Instruction Multiple Data (SIMD) or Very Long Instruction Word (VLIW) architectures can be used to increase performance but also lead to an increase of the programming complexity. In order to increase the performance the designer can either use to choose an homogeneous architecture made of a single type of PE, or use multiple heterogeneous PEs with an increase in the programming complexity.

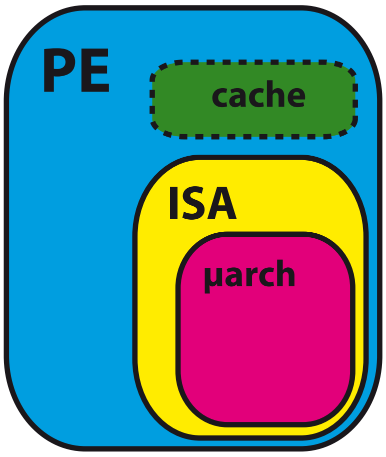


Figure 6.1: *The processing element structure.*

– **Memory System**

The throughput of a multi-core processor greatly depends on the memory architecture. The memory architecture can be characterized by the cache configuration, the intra-chip interconnect and the consistency model. The consistency model defines how memory consistency is ensured on the chip. A strong consistency model means that the write and read orders are strictly handheld by the communication system. A weak consistency model will let the programmer manage the memory consistency by using synchronization primitives. Cache memory has been introduced to tackle the issue of speed access to external memory. This kind of memory can be accessed at high clock speed and stores highly used chunk of memory. Processors usually have one to three cache levels, the closer level having the fastest speed and the smallest size, while the further has slowest speed with larger size. Cache coherence can be both handheld by on-chip hardware at silicon cost (takes area) or by the programmer. The architecture performances greatly depend on the intra-chip interconnects. It defines the way the PEs are connected together and can exchange information. The computation speed-up coefficient on a parallel architecture for a given application relies on the number of cores involved and the time spent communicating between the PEs. On a multi-core system, the intra-chip interconnect is the on-chip communication system that allows the processing elements to exchange data. Different intra-chip interconnect models exist:

– Bus interconnect topology: Processing Elements share a bus with an arbitration element (Fig.6.2).
– Crossbar topology: a crossbar is a model in which all components are connected through a switches (Fig. 6.2).
– Ring topology: all PEs are connected to a ring (Fig. 6.2).
– Network On Chip topology: on a network on chip, all resources are connected onto a communication network managing route selection to pass messages between PEs (Fig. 6.2).

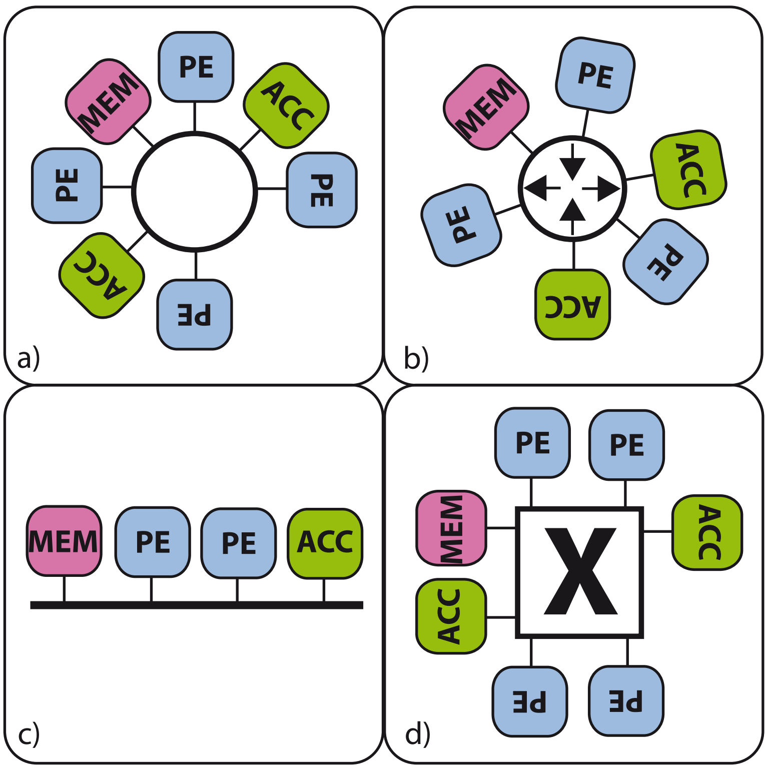


Figure 6.2: *Intra-chip interconnects: a) ring topology, b)NOC topology, c) Bus topology, d) Crossbar topology.*

– **Accelerators/Integrated peripherals**

In order to lower the power consumption and decrease the programming complexity, multi-core architectures can be provided with hardware accelerators. These accelerators can be targeted at communication or computation. Accelerators such as DMA allow to perform data-transfer with PEs involved only for transfer configuration. Accelerators that perform critical computation can also be used to decrease the processor load on specific algorithm. Those accelerators being specialized for a few computations, their power consumption is even more optimized.

A Multi-processor system can be described with the same criteria as a multi-core architecture with an additional element that is the inter-chip interconnects. On such systems, the processing elements are logically and physically divided as they do not run on the same die. Thus, they need communication medium to exchange data. The inter-chip interconnect is the on-board interconnection that exists between chips. Inter-chip interconnect working at lower speed than intra-chip interconnect, it usually results in a loss of throughput but a gain of potential scalability as the PE number can (in theory) infinitely increase. The programming model also becomes different, as the parallelism has to be extracted at a coarser grain to tackle the issue of low-speed communications.

6.3 Data Flow model of computation

6.3.1 Introduction

The data-flow model of computation aims at representing applications in regards to the data dependencies that exist between the different parts of the applications. The application is thus represented as a network of the several actions the application is composed of. This kind of representation allows to get knowledge of the application intrinsic parallelism and memory requirement. The original data flow paradigm was introduced by Kahn in [Kah74]. A Kahn process network is a network of actors connected by unbounded FIFO that carries data token. A data token is an atomic (can not be divided) chunk of data. Kahn also established a formal representation for Kahn process network. An actor can thus be described as a functional process that maps a set of token sequence into a set of token sequence. The Data flow process network inherits its formal underpinning from Kahn process networks, but associates to each actor a set of firing rules that gives the necessary tokens input for an actor to trigger. This allows further analysis of the network using a set or properties as described in [LP95]. To allow further compile-time analysis, the Synchronous Data Flow model (SDF) [LM87b] takes the Data-flow process network semantic and restricts its expressiveness by specifying the data production/consumption rate in and integer form for each actor interconnection. Such information allows to ensure the application to be deadlock free, to compute a statical schedule of the application and to compute memory requirements. In order to allow expressiveness of the quite restrictive SDF model, several models were issued. The Boolean Data Flow model (BDF) [Buc93] aims at introducing *switch* and *select* statements in the SDF model. This model proposes to include specific actor in the model, whose production/consumption rates can be controlled by a conditional input taking boolean values. The Cyclo-static Data Flow [BELP95] model (CSDF) allows to describe production/-consumption rates as a sequence of integers, thus allowing the actor to behave in a different manner in a sequence of activations. The Parametrized Synchronous Data Flow [BB01] (PSDF) populates the use of parameters that can affect productions/consumptions rates and functional behavior of an actor. Such models provide a better expressiveness than SDF but decreases the analyzability of the network. The Figure 6.3 shows a classification of the data-flow models. While it is easy to classify some models, it is difficult to establish a hierarchy between some of them. For example PSDF, CSDF and BDF really show different behavior and the classification established in Figure 6.3 is not relevant.

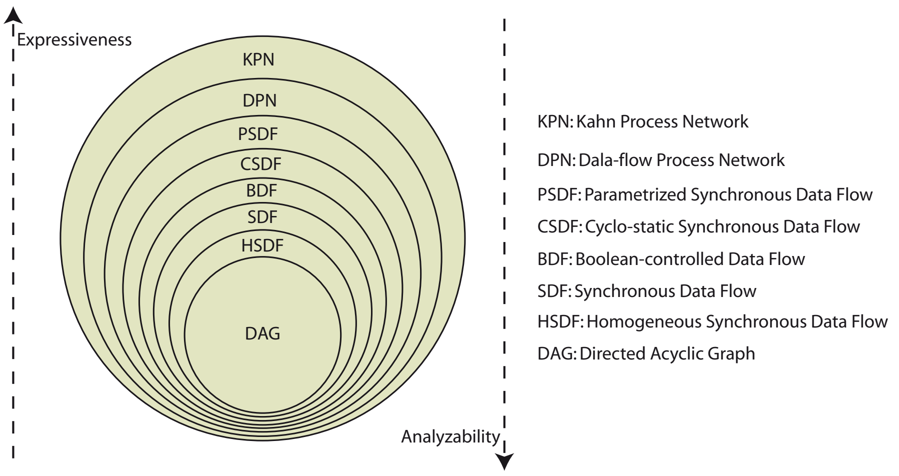

Expressiveness
KPN
DPN
PSDF
CSDF
BDF
SDF
HSDF
DAG
Analyzability
KPN: Kahn Process Network
DPN: Dala-flow Process Network
PSDF: Parametrized Synchronous Data Flow
CSDF: Cyclo-static Synchronous Data Flow
BDF: Boolean-controlled Data Flow
SDF: Synchronous Data Flow
HSDF: Homogeneous Synchronous Data Flow
DAG: Directed Acyclic Graph

Figure 6.3: *Data-flow Models of computation taxonomy.*

In the following we will give an introduction to the data-flow paradigm starting by the mother of all, the Kahn Process Network followed by the Data Flow Process Network and provide an in-depth exploration of the previously cited models.

6.3.2 Data Flow paradigm introduction

Kahn Process Network (KPN)

A process network is a set of concurrent processes that are connected through one-way unbounded FIFO channels. Each channel carries a possibly finite sequence (a stream) that we denote $X = [x_1, x_2, ...]$. Each x_i is an atomic data object called token belonging to a set. Token cannot be shared, thus tokens are written (produced) exactly once and read (consumed) exactly once. As FIFOs are considered unbounded, the operation that writes to a channel is non-blocking while a read operation blocks until at least one token is made available. A process in the Kahn model maps one or more input sequences to one or more output sequences. Consider a prefix ordering of sequences, where the sequence X precedes the sequence Y (written $X \subseteq Y$) if x is a prefix of (or is equal to) Y. For example $[x_1, x_2] \subseteq [x_1, x_2, x_3]$. It is then common to say that X approximates Y as it gives partial information about Y. An empty sequence is denoted $\perp$, and is a prefix to any other sequence. For an increasing chain of sequences $\chi = \{X_0, X_1, ...\}$ where $X_0 \subseteq X_1 \subseteq ....$ Such an increasing chain has one or more upper bounds Y where $X_i \subseteq Y, \forall X_i \in \chi$. The least upper bound $\top_\chi$ is an upper bound such that for any other upper bound Y, $\top_\chi \subseteq Y$. The least upper bound may be an infinite sequence. Let S denote the set of finite and infinite sequences. Let S^p denote the set of $p - tuples$ of sequences as in $X = \{X_1, X_2, ..., X_p\} \in S^p$. The set $\perp \in S^p$ is understood to be the set of empty sequences.

Such sets of sequences can be ordered as well; we write $X \subseteq X'$ if $X_i \subseteq X'_i$ for each $i, 1 \leq i \leq p$. A set of p-tuples of sequences $\chi = \{X_0, X_1, ...\}$ always has a greatest lower bound $\sqcap\chi$ (possibly $\perp$), but it may not have a least upper bound $\sqcup\chi$. If it is an increasing chain, $\chi = \{X_0, X_1, ...\}$, where $X_0 \subseteq X_1 \subseteq ...$, it has a least upper bound, so S^p is a complete partial order for any integer p.

A functional process $F : S^p \to S^q$ maps a set of input sequences S^p to a set of output sequences S^q. Thus, a network of processes is a set of simultaneous relations between sequences.

Let X_0 denote all the sequences in the network, including the output, and I the set of input sequences. Then the network of functional processes can be represented by a mapping F where $X_1 = F(X_0, I)$. Any X for which $X_1 = X$ is called a fixed point.

Data Flow Process Network

A data flow actor when firing processes input tokens and produces output tokens. Actor firing is conditioned by a set of rules specifying which tokens must be available at the input for an actor to fire. Such sequence is a special Kahn process called a data flow process. Thus a network of actors is a special kind of Kahn network called a data flow process network.

An actor with $p \geq 1$ input streams can have N firing rules, $R = \{R_1, R_2, ..., R_N\}$. The actor can fire if and only if at least one of the firing rules is satisfied. Each firing rule constitutes a set of patterns for each of the p inputs.

$R_i = \{R_{i,1}, R_{i,2}, ..., R_{i,p}\}$

Each $R_{i,j}$ is a finite sequence. To satisfy a firing rule i, each $R_{i,j}$ must form a prefix of the sequence of the unconsumed tokens on input j. An actor with no input stream is considered always enabled.

Some patterns in a firing rule may be empty $R_{i,j} = \perp$, this means that any available sequence at input j is acceptable, because as seen previously, $\perp$ is a prefix to any sequence. This does not mean that input j must be empty.

The following notation has been introduced in [LP95]. The symbol "*" will denote a token wildcard. Thus a sequence $[*]$ is a prefix to any sequence containing at least one token. $[*]$ is a prefix to any non empty sequence. $[*]$ can only be prefixed by $\perp$. The statement $[*] \subseteq X$ denotes that X is a sequence of at least one token , but not means that any-one token is a prefix to X.

The actor behavior is defined as a map function. This map function is a higher order function as it takes a function as an argument and returns a function. We define $F = map(f)$ where $f : S^p \Rightarrow S^q$ is a function, to return a function $F : S^p \Rightarrow S^q$ that applies f to each element of the stream when a set of firing rules is enabled. $F = map(f)$ where $F(R : X) = f(R) : F(X)$ and R is the firing rule of f. The colon ":" indicates a concatenation of sequences. This definition is recursive and terminates when the argument to F has no longer firing rule as a prefix.

6.3.3 Synchronous Data Flow (SDF)

The Synchronous Data-Flow (SDF) [LM87b] is used to simplify the application specification, by allowing the representation of the application behavior at a coarse grain. This data-flow model represents operations of the application and specifies data dependencies between the operations.

A synchronous data flow actor with p inputs is specified by one firing rule $R = \{R_1\}$ that is a set of patterns $R_1 = \{R_{1,1}, R_{1,2}, ..., R_{1,p}\}$ one for each p inputs. Each pattern is specified as a finite sequence of wildcards $R_{1,j} = [*]^n$ with $n > 0$.

A synchronous data flow actor is a functional process that maps a set of fixed size wildcard sequences into another set of fixed size wildcard sequences. The map function $F = map(f)$ is $f(R_{i,j}) = S^q$ with S^q being a set sequence of sequences $X \in [*]^n$. Thus a synchronous data flow actor consumes a fixed amount of tokens on all its input and produces a fixed amount of tokens on all its outputs.

This data flow model can be represented as a finite directed, weighted graph $G =< V, E, d, p, c >$ where :

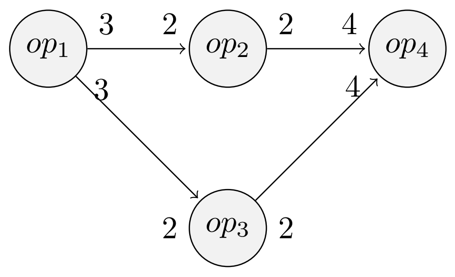
- V is the set of nodes; each node represents a computation that operates on one or more input data streams and outputs one or more output data streams.
- $E \subseteq V \times V$ is the edge set, representing channels carrying data streams.
- $d : E \rightarrow N \cup \{0\}$ $(N = 1, 2, \ldots)$ is a function with $d(e)$ the number of initial tokens on an edge e.

- $p : E \rightarrow N$ is a function with $p(e)$ representing the number of data tokens produced at e's source to be carried by e.
- $c : E \rightarrow N$ is a function with $c(e)$ representing the number of data tokens consumed from e by e's sink node.

This graph only specifies the topology of the network but does not give any information about the actor internal behavior. The only behavioral information is regarding the amount of tokens produced/consumed.

From this representation one can extract a valid schedule, as a finite sequence of actor invocation that fires at least once with no deadlock and does not change the state of the graph. Such sequence exists if and only if the equivalent Kahn process network admits a minimum fixed point X for which $X = F(X, I)$ where X denotes all the sequence in the network including the outputs and I the set of inputs.

Such networks can be characterized by a matrix similar to the incidence matrix in the graph theory. The topology matrix Γ is the matrix of size $|E| \times |V|$, in which each row corresponds to an edge e in the graph, and each column corresponds to a node v. Each coefficient (i, j) of the matrix is positive and equal to N if N tokens are produced by the j^{th} node on the i^{th} edge. (i, j) coefficients are negative and equal to N if N tokens are consumed by the j^{th} node on the i^{th} edge. It was proved in [LM87b] that a static schedule for graph G can be computed only if its topology matrix's rank is one less than the number of nodes in G. This necessary condition means that there is a Basic Repetition Vector (BRV) q of size $|V|$ in which each coefficient is the repetition factor for the j^{th} vertex of the graph. This basic repetition vector is the positive vector with the minimal modulus in the kernel of the topology matrix such as $q.\Gamma = \{0\}$. Such vector gives the repetition factor of each actor for a complete cycle of the network. In the Figure 6.4 the topology matrix is constructed as explained previously. The rank of this topology matrix being 3, it means that this network admits a schedule. The basic repetition vector of this network is $[4, 6, 6, 3]$ meaning that $q(op_1) = 4, q(op_2) = 6, q(op_3) = 6, q(op_4) = 3$.



$$\begin{pmatrix} 3 & -2 & 0 & 0 \\ 3 & 0 & -2 & 0 \\ 0 & 2 & 0 & -4 \\ 0 & 0 & 2 & -4 \end{pmatrix}$$

Figure 6.4: *An SDF graph and its corresponding topology matrix.*

In a SDF network some actor are purely computation less, as their role is only to organize their input(s) datas into their output(s). Those special vertices depicted in Figure 6.5 can flowingly be termed :

– Fork: the fork vertex takes N tokens on its single input, and outputs N/nb_o tokens on its nb_o output ports.
– Join: the join vertex takes N/nb_i tokens on its nb_i input and outputs N tokens on its single output.
– Broadcast or Diffuse: the broadcast or diffuse vertex takes N tokens on its single input, and output the same N tokens on each of its outputs.
– Ring : the ring vertex takes N tokens on each of its inputs, and outputs only the last N tokens on its single ouptut.

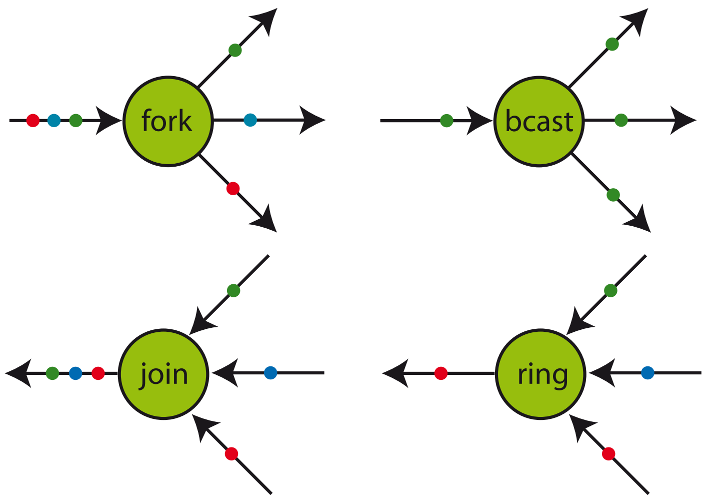


Figure 6.5: *Illustration of the computation less vertices.*

SDF to DAG translation

One common way to schedule SDF graphs onto multiple processors is to first convert the SDF graph into a precedence graph such that each vertex in the precedence graph corresponds to a single execution of an actor from the SDF graph. Thus each SDF graph actor A is “expanded into” q_A separate precedence graph vertices, where q_A is the component of the BRV that corresponds to A. In general, the SDF graph aims at exposing the potential parallelism of the algorithm; the precedence graph may reveal more functional parallelism and moreover, it exposes the available data-parallelism. A valid precedence graph contains no cycle and is called DAG (Directed Acyclic Graph). Unfortunately, the graph expansion due to the repetition count of each SDF node can lead to an exponential growth of nodes in the DAG. Thus, precedence-graph-based multiprocessor scheduling techniques, such as those developed in [Pri91] [SL90], in general have complexity that is not polynomially bounded in the size of the input SDF graph, and can result in prohibitively long scheduling times for certain kinds of graphs (e.g., see [PBL95]). The Figure 6.6 shows the precedence graph of an SDF network, thus highlighting the network expansion.

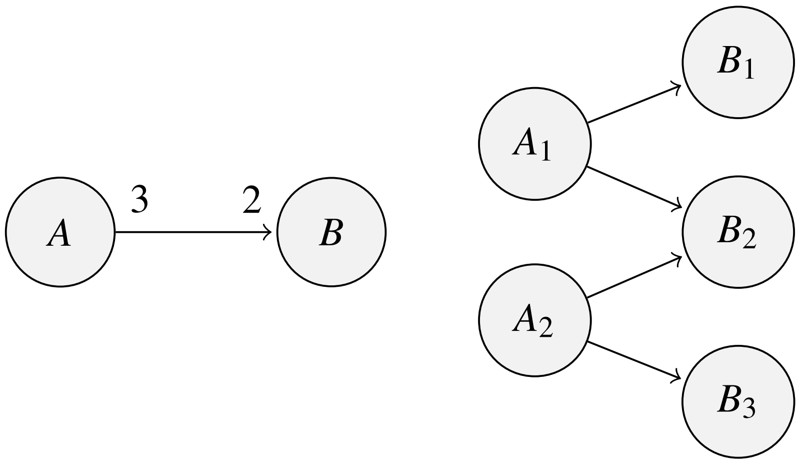


Figure 6.6: *An SDF graph and its precedence graph.*

6.3.4 Homogeneous Synchronous Data Flow (HSDF)

The homogeneous synchronous data flow model is a subset of the synchronous data flow. A homogeneous synchronous data flow actor with p inputs is specified by one firing rule $R = \{R_1\}$ that is a set of patterns $R_1 = \{R_{1,1}, R_{1,2}, ..., R_{1,p}\}$ one for each p inputs. Each pattern is specified as a sequence of one wildcard $R_{1,j} = [*]$. The mapping function associated with each actor maps a set of input sequence S_p into sets of unique wildcard sequences $f : S^p \to S^q$ with $S^q = \{X_1, X_2, ..., X_p\}$ with $X_i = [*]$. The Figure 6.7 shows an HSDF graph with only unary production and consumption rates. Production and consumption rates can higher than one, but must be equal on one arc.

6.3.5 Boolean-controlled Data Flow (BDF)

The boolean control data flow allows an SDF graph to have non-determinism actor. Those special actors are boolean controlled. This boolean control the firing rules and the producing rules. A boolean controlled actor with p inputs and a control port c is specified by two firing rules, $R = \{R_T, R_F\}$. The rule R_T specifies the valid input sequences for the p input ports for a true value on the port c $R_T = \{R_{T,1}, R_{T,2}, ..., R_{T,p}, [T]\}$ and R_F specifies the valid input sequences for the p input ports for a false value on the port c $R_F = \{R_{F,1}, R_{F,2}, ..., R_{F,p}, [F]\}$. A valid boolean controlled data flow actor must consume at least one token on any of its ports

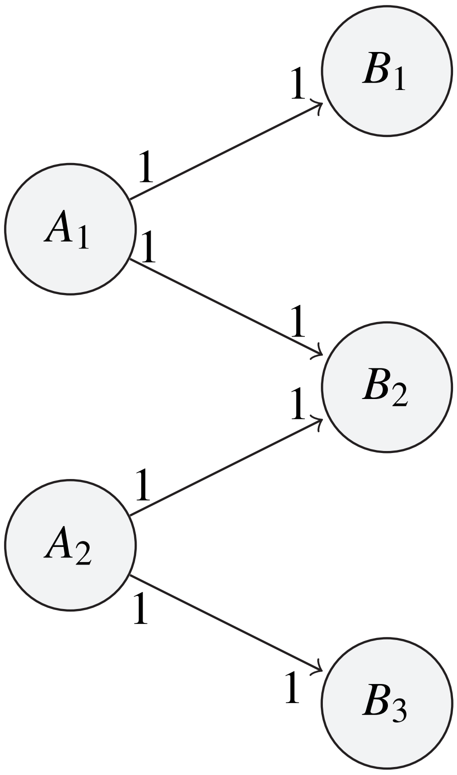


Figure 6.7: *An HSDF graph.*

(excluding the control port) for any control value. Typical BDF actors are the select and switch actors as depicted in Figure 6.8. The set of rules ruling the select actor are $R = \{[X, \perp, T], [\perp, X, F]\}$ with X a sequence of fixed number of wildcards. The switch actor only has one firing rule $R = \{[*, *]\}$ but its mapping function behavior is ruled by the control token $f([X, [T]]) = [X, \perp]$ and $f([X, [F]]) = [\perp, X]$ with X a sequence of fixed number of wildcards. The switch actor takes one input and copies it on one of its two outputs depending on the control token value. For a control token equal to one, the input is copied on the T port, when a control token zero leads to a copy on the output F. The select actor takes two inputs and copies the value of one of them on its outputs depending on the control token value. For a control token one, the input T is copied on the output, and a control token 0 leads to a copy of the F input on the output port.

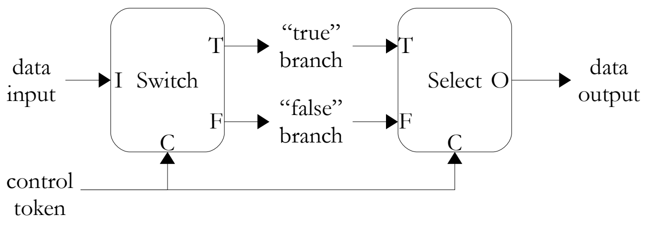


Figure 6.8: *BDF switch and select actors*

Consistency analysis of the BDF model relies on an extension of the topology matrix. Conditional ports are annotated with a symbolic expression p that represents the probability of token consumption/production on this port. If the port produces/consumes a token for a $TRUE$ token on its associated control port, the annotation p_i is the proportion of $TRUE$ tokens on the control port for n control tokens consumed. If the port produces/consumes a token for a $FALSE$ token on its associated control port, the annotation is the proportion of $FALSE$ tokens on the control port for n control tokens consumed, that is $p_i - 1$. We could consider p_1 as being the probability of appearance of a $TRUE$ token on the control port, but in that case the existence of a complete cycle would only be true for control token stream stationary in the mean. The above formulation denotes as n the length of a well-defined sequence of actor firing called a complete sequence.

BDF consistency can be computed by finding the null-space of a symbolic topology matrix Γ for which the elements are the production/consumption rate for the data flow actors and the p_i annotation for the BDF actors. Such vectors can have non-trivial solution and are consequently call*strongly consistent*. A system for which non-trivial solution exists only for specific values of p_i is called weakly consistent. Thus additional information must be provided by the user to ensure that the system can consistently run over time.

The solution vector of the system $Gamma(\vec{(p})r(\vec{p}) = \vec{0}$ is a function of $\vec{p}$ and k with $r(\vec{p} = kq(\vec{p})$. $q(\vec{p})$ is the minimal non-trivial solution of the system, and k is the minimal integer for which all components of the vector $kq(\vec{p})$ are integer. The definition of p_i being the proportion of $TRUE$ tokens on the control token port for n invocations of the graph, leads to $k = n$ as $p_i = \frac{t_i}{n}$ with t_i being the number of $TRUE$ tokens.

The Boolean controlled data flow can be extended to the Integer controlled dataflow where the control token can be of any integer value. Thus, each actor can be specified by a finite set of firing rules, one for each possible value of the control token. Such model can be analyzed as a composition of BDF actors forming an integer solution.

6.3.6 Cyclo-Static Synchronous Data Flow (CSDF)

The Cyclo-static synchronous data flow generalizes the SDF model by allowing the number of tokens consumed/produced to vary from one firing to the next in a cyclic pattern. A Cyclo-static actor with p inputs is specified by an ordered set of sequential firing rules $R = \{R_1, R_2, ..., R_N\}$ that gives for a finite number of firing the sequence of firing rules. Thus the actor is executed for the first time when the firing rule R_1 is met. Then for the following invocation the firing rule R_2 has to be met. Each following invocation require the next firing rule to be met. When the end of the ordered set is reached, the sequence start over at firing rule R_1. For a firing i of the actor, at least one interface must require a token. The Figure 6.9 shows a CSDF network with producing and consuming sequence a the source and sink of each arc.

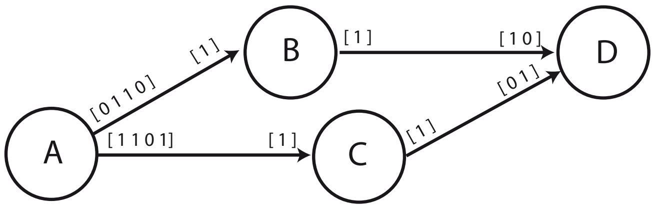


Figure 6.9: *A Cyclo-static Data Flow network.*

For such model, constructing a “consistent” schedule exists if the following condition is met. Given a connected cyclo-static data flow graph G. A vector $\vec{q_G} = [q_1, q_2, ..., q_{NG}]$ gives for actor q_i its number of invocation. $\vec{q_G} = P.\vec{r}$ with $\vec{r}$ being the solution of $\Gamma\vec{r} = 0$. Γ is denoted as the topology of the data flow graph, that can be described as

$$\Gamma_{i,j} = \begin{cases} (P_j/P_j^i.X_j^i(P_j^i)) \text{ if task j produces on edge i} \\ (P_j/Q_j^i.Y_j^i(Q_j^i)) \text{ if task j consumes from edge i} \\ 0 \text{otherwise} \end{cases}$$

where :

– P_v^u period of the production sequence of task v on edge u
– Q_v^u period of the consumption sequence of task v on edge u
– P_v is the least common multiple of $\{P_v^u\}$
– $X_v^u(P_v^u)$ number of tokens produced by actor v on edge u for a full production period of task v on edge u
– $Y_v^u(Q_v^u)$ number of tokens consumed by actor v on edge u for a full consumption period of task v on edge u

The solution vector q_G is said to be consistent as this necessary condition ensures buffers to be bounded, but does not ensure that deadlocks cannot occur. A cyclo-static data flow graph that will not deadlock is said to be “alive”. Checking a graph to be alive is performed by checking the singular loops. A singular loop is a direct path in a set of strongly connected element which starting and finishing element are the same for which each element appears only once.

Definition A singular loop L in a cyclo-static data flow graph is a direct path $v_1 \xrightarrow{u_{1,2}} v_2 \xrightarrow{u_{1,3}} ... \xrightarrow{u_{N_L-1,N_L}} v_{N_L} \xrightarrow{u_{N_L,1}} v_1$ where the starting node equal the ending node. A singular loop is a loop where all the tasks appear only once.

Definition The first producing invocation $FPI(v_j, n_j, L)$ is the s_j invocation of v_j with $s_j \geq n_j$ with non-zero production toward v_{j+1}.

Definition The first-dependent-successor-invocation $FDSI(v_j, n_j, L)$ of $v_j(n_j)$ in L is the first invocation of v_{j+1} that requires data from v_j to become executable.

Definition The first data sequence $FDS(v_j, n_j, L)$ of $v_j(n_j)$ in L is the invocation sequence $v_j(n_j)...v_j(s_j)v_{j+1}(n_{j+1})...v_{j+1}(s_{j+1})...v_{N_L}(s_{N_L})v_1(n_1)...v_{j-1}(s_{j-1})v_j(n_j)$ where $s_k = FPI(v_k, n_k, L))$ and $n_k = FDSI(v_{k-1}, s_{k-1}, L)$.

Finding a FDS sequence is performed by iteratively finding the iteration s_j of v_j that produces data toward v_{j+1}. Then this procedure is repeated among all actors of the loop until we reach invocation n_j of task v_j to construct a valid FDS sequence. The graph is then said to be alive if $FDS(v_j, s_j^k, L) = v_j(s_j^k)v_{j+1}(n_{j+1}^{k+1})...v_j(n_{j+1}^k)$ ends on an instance $n_j^{k+1} \geq s_j^k$ of v_j, for $0 \leq k \leq m$. In this formula $s_j^k = FPI(v_j, n_j^k, L)$, n_j^0 is the first invocation of v_j that is not initially executable and m corresponds to the first invocation of v_j for which $FDS(v_j, s_j^m, L)$ passes by a task invocation $n_i^{m+1} > q_i$ of a task v_i of L.

When one has determined a repetition vector and checked the graph aliveness, the graph can be scheduled. Scheduling a CSDF can be performed in two different ways: by transforming the CSDF specification into an HSDF representation, or by forcing each instance of a task to be executed on the same processing device. In the second way, the tasks are connected by FIFOs whose size can be statically computed.

6.3.7 Parameterized Synchronous Data Flow (PSDF)

Parameter-based SDF hierarchy has been introduced in [BB01] where the authors introduce a new SDF model called *Parameterized SDF*. This model aims at increasing SDF expressiveness while maintaining its compile time predictability properties.

In this model a sub-system (sub-graph) behavior can be controlled by a set of parameters that can be configured dynamically. These parameters can either configure sub-system interface behavior by modifying production/consumption rate on interfaces, or configure behavior by passing parameters (values) to the sub-system actors. In this model each sub-system is composed by three graphs: the init graph ϕ_i, the sub-init graph ϕ_s, the body graph ϕ_b.

The body graph ϕ_b is a synchronous data flow network. The actors of this data flow network are specified by one rule $R = \{R_1\}$ that is a set of patterns $R_1 = \{R_{1,1}, R_{1,2}, ..., R_{1,p}\}$ one for each input. Each $R_{1,N}$ is as sequence $R_{1,i} = [*]^t$

with t being parameter resolved at run-time. The mapping function of each actor can also be parameterized $f(R_1) = [*]^k$.

The init graph ϕ_i does not consume any data flow tokens but is only fired once before an invocation of ϕ_b. This graph affect the parameters used in the firing rules and mapping function. Thus ϕ_b mapping function and firing rules remain static between two invocations of ϕ_i. In Figure 6.10 the parameter s that affect the actor A consumption rate is set by the ϕ_i. This parameter s is the sub-sampling factor.

The sub-init graph consumes data flow tokens and is fired before each firing of the network ϕ_b. This network can affect parameters that can parameterized ϕ_b actors behavior in term of computation but can also set parameter used in production consumption rate as long as it does not affect the upper hierarchy network. In Figure 6.10 the parameter p that affect the actor A internal behavior is set by the ϕ_s. The parameter p is the phase parameter of the sub-sampler.

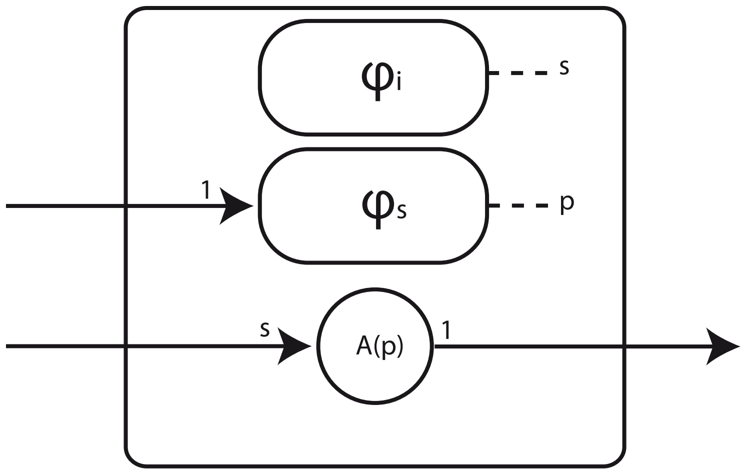


Figure 6.10: *PSDF specification of a sub-sampler actor.*

Each activation of the sub-system, is composed by an invocation of ϕ_s followed by an invocation of ϕ_b. The init graph is effectively decoupled from the data flow specification of the parent graph and invoked once, at the beginning of each (minimal periodic) invocation (see [BB01]). The sub-init graph performs reconfiguration that does not affect sub-system interface behavior and is activated more frequently than the init-graph that can modify sub-system interface behavior. In order to maintain predictability, actors of ϕ_b are assigned a configuration which specifies parameters values. This value can either be a domain which specifies the set of valid parameter value combinations for the actor, or left unspecified, meaning that this parameter value will be determined at run-time.

A PSDF graph is considered to have SDF behavior for each invocation. This ensures that its schedule can be constructed as a dynamically configurable SDF schedule. Such scheduling leads to a set of *local synchrony* constraints for PSDF graphs and PSDF sub-systems that need to be satisfied for consistent specification. Those constrains ensure the program to have a SDF behavior between two invocations. A configuration C is a complete set of parameters. Lets denote $DOMAIN(G)$ the set of valid complete configurations for a graph G. A configuration $C \in DOMAIN(G)$ must meet the following conditions to ensure the *local synchrony*:

1. The graph $instance_G(C)$ has a valid schedule, is sample rate consistent and deadlock free.
2. The consumption/production (k_v, φ_v) rate on any port Φ of any actor v for the configuration C must be less than the maximum consumption/production τ_v rate specified by the designer. $k_v(\Phi, config_v, C) \leq \tau_v$ and $\varphi_v(\Phi, config_v, C) \leq \tau_v$.
3. The maximum delay value bound μ_e is satisfied for every edge. $e \in E\delta_e(config_e, C) \leq \mu_e$.
4. Every child subsystem is synchronous.

If those conditions are satisfied for every $C \in DOMAIN(G)$, we say that g is inherently locally synchronous. If only some $C \in DOMAIN(G)$ satisfies the conditions then G is partially locally synchronous. A partially locally synchronous PSDF specification will require configuration checking at run-time to ensure a correct behavior.

6.3.8 Conclusion

The data flow model of computation is composed of several sub-models that each provide different information and/or limits the specification behavior to meet defined criteria. The synchronous data flow model, despite its limited expressiveness, has proved to be adapted for signal processing applications specifications. The main limitation of this model is the fixed production/consumption rates that are used to compute the statical scheduled. The CSDF model overcome this limitation by allowing to define production/consumption as a sequence of integers. Such representation can make the model painful to specify and the scheduling procedure can either bring a lot of complexity (HSDF transformation) or limit the mapping possibilities on multi-core architecture. The PSDF model seems to be a nice trade-off, but let the user with the responsibility to provide verified informations on the configuration to ensure local synchrony. Other models such as BDF tackles the issue of conditioning but does not ensure memory boundedness based only on the model informations. One must remind that these model are before all, model of execution. Thus a program can be thought as a composition of the different models. A Data Flow process network can be analyzed to extract those different behaviors for optimization or scheduling purpose. Unfortunately identifying those behaviors is hard to performed automatically, and a precise knowledge of the application behavior is necessary. In the following we will introduce the tools that implements these models as application specification models.

6.4 Application modeling tools

6.4.1 Introduction

As application complexity and available target architecture grow, application designer needs to use abstraction models that allow to specify an application behavior with minimum knowledge of the target architecture. Such model not only allows to target a wide range of architectures but can also be use to co-design the software and the hardware. Such model usually relies on a target independent language/representation and is associated to an execution model to allow automatic analysis and optimizations. Those specifications are adapted to the targeted application context. In the following, we will give an overview of the existing models and associated languages that target DSP applications.

6.4.2 Dataflow Interchange Format

DIF (Dataflow Interchange Format) [jHKyK+04] is a language for specifying dataflow models for DSP systems developed at the University of Maryland. DIF aims at providing an extensible repository of models, analysis, transformation, scheduling, code generation for dataflow. The models currently supported are:

- Synchronous Dataflow.
- Homogeneous Synchronous Dataflow.
- Single Rate Dataflow.
- Cyclo-static Synchronous Dataflow.
- Parameterized Synchronous Dataflow.
- Multidimensional Synchronous Dataflow.
- Boolean Controlled Dataflow.
- Enable Invoke Dataflow.
- Core Functionnal Dataflow.

DIF-to-C [HKB05] allows to generate C code from an SDF specification. To achieve so, the DIF package (Fig. 6.11) provides an extensive repository of schedul-

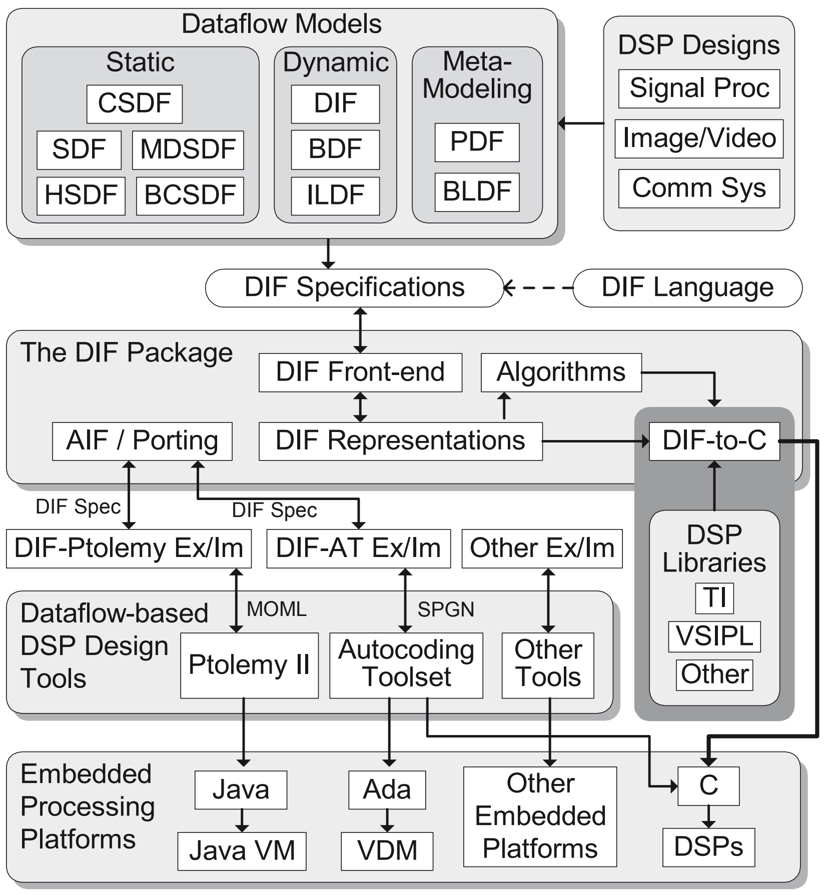


Figure 6.11: *DSP system design with DIF.*

ing (Flat, Flattening, Hierarchical, APGAN (Algorithm for Pairwise Grouping of Adjacent Node)) and buffering techniques (Non shared, Circular, Buffer Sharing, Static, In-place Buffer Merging). The generated C code targets mainly DSP processors.

6.4.3 SDF3

SDF3 [SGB06] is a tool for generating random Synchronous DataFlow Graphs (SDFGs), if desirable with certain guaranteed properties like strongly connectedness. It includes an extensive library of SDFG analysis and transformation algorithms as well as functionality to visualize them. The tool can create SDFG benchmarks that mimic DSP or multimedia applications. In addition, the tool support the Scenario Aware Dataflow model [The07] that allows parameterized production/consumption rates to be specified, and for which the actor execution time depends on the active scenario. In addition this model distinguish the dataflow and the control flow allowing further optimizations.

6.4.4 Ptolemy II

Ptolemy [LHJ+01] is a tool developed at Berkeley for application modeling and simulation. An application can be described using a composition of actors called networks directed by a given computation model called the director. It supports a wide collection of computation models, from event-driven to most of the data-flow models and process network. Ptolemy II is the successor of Ptolemy classic and has been under development since 1996. A specification can then be executed and debugged using underlying java code. The tool is supported by a graphical environment called VIRGIL (Fig 6.12), that allows graph specification and graphical application debugging. The models are stored in a concrete XML syntax called MoML.

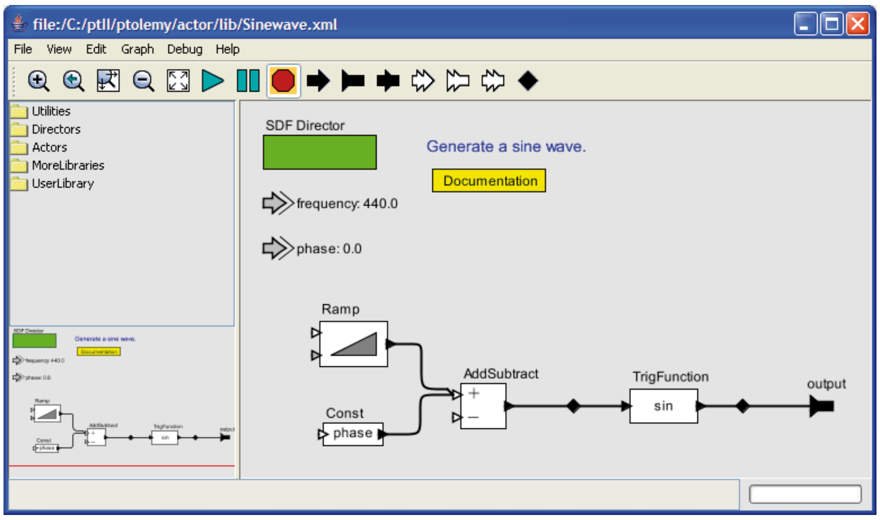


Figure 6.12: *Implementation of a Sine wave generator in ptolemy.*

6.4.5 StreamIt

StreamIt [TKG+01] is a programming language and a compilation infrastructure developed at the MIT, specifically engineered for modern streaming systems. It is designed to facilitate the programming of large streaming applications, as well as their efficient and effective mapping to a wide variety of target architectures, including commercial-off-the-shelf uniprocessors, multi-core architectures, and clusters of workstations. The specification is made using a textual representation. Each actor is declared as a filter. A Filter is an actor with exactly one input and one output port. It can declare an initialization (init), a pre-work and a work. The init is executed at the instantiation and does not consume/produce tokens. The pre-work is the very first execution of the filter. The work function defines the behavior of the filter. Work and pre-work are characterized by push and pop properties that define the production and consumption of the filter (see Fig. 6.13). Filters can support dynamic rates.

The network of actors is defined as a pipeline in which filters are added in a sequential order. One can also use a split-join section that is defined as a section of filters in which the input data is split and the output is the result of the join of several filters.

The model is supported by a command-line compiler that can be setup with different optimization options.

6.4.6 PREESM

PREESM (Parallel Real-time Embedded Executive Scheduling Model) is a rapid prototyping tool for easy application specification to automatic target specific code generation under development at the Institute of Electronic and Telecommunication of Rennes (IETR) since 2007 (see Figure 6.15). The proposed framework goes from graph specification using a graph Editor (Graphiti), to code generation using PREESM and a library of plug-in. The Graphiti graph editor allows the user to design application graph using a Synchronous Data Flow (SDF) semantic with specific interpretation of the hierarchy [PBR09]. The design can be hierarchical when a vertex in the graph has a graph representation for its behavior, or atomic when the vertex behavior is described using programming language as C. Graphiti can also edit architectures described using IP-XACT language, an IEEE standard from the SPIRIT consortium [SPI08].

Synchronous Data Flow graphs can then be processed using the PREESM Eclipse plug-in which computes different transformations on the graph and then can match the graph with a given hardware architecture using different mapping algorithms. At the end of the process, the so mapped application is translated into C code, which can be compiled and executed on the hardware target.

All the transformations (mapping, code generation) algorithms are designed as PREESM plug-ins, allowing users to extend the transformation, mapping, code generation algorithms library. The process, going from graph transformation to code generation, is driven by a Workflow graph (also edited in Graphiti) that specifies the order in which the PREESM plug-ins are to be executed and all the data exchanges between the plug-ins (Figure 6.14).

6.4.7 SynDEx data-flow model

SynDEx is a rapid prototyping tool developed at the INRIA Rocquencourt [GS03]. The tool allows graphical specification of the application and architecture and generates m4 code, that can be later transformed to c code, for each of the architecture processing element. The application specification relies on a dataflow model close to SDF that does not support multi-rythm. Production specification on one arc must be an integer product of the consumption, meaning that a consumer cannot consume more data than produced by one firing of the producer. At least one arc must satisfy these criteria for two connected actors, the other are considered as broadcast arcs. Figure 6.16 shows an MPEG-4 decoder specified with SynDEx tool. Blocks with blue headers are the actors when blocks with green headers represent the delays.

6.4.8 Canals

Canals [DEY+09] is a data flow modeling language supported by a compiler infrastructure. This language and its compiler are being developed at the university of Turku in Finland by the Center for Reliable Software Technology. The language is based on the concept of nodes (kernel and networks) and links which connect nodes together. A node or kernel is a computational element that takes a fixed amount of tokens on its input and output a fixed or arbitrary number of element on its output. A kernel can also *look* at values on its input port without consuming it. Special kernels *scater* and *gather* are used for data distribution and grouping with a specified behavior.

In the language, the scheduling is separated from the network, and specified by the user for the whole network. The scheduler is responsible for planning the execution of kernels, in such a way that data available as input to the network is routed correctly through the network and eventually becomes available as output from the network. The scheduler can accomplish correct routing by inspecting data, obviously at run-time, arriving to the network and make routing decisions based on the contents of the data.

The Canals compiler is then able to provide a mapping of the top network onto a given architecture and generate code for each programmable components. The generated C, C++ is platform independent and the hardware specific resources and communication links are handheld by an Hardware Abstraction Layer library for each component.

6.4.9 CAL Actor Language

RVC-CAL [BEJ+09, ISO09] is a subset of the CAL Actor Language [EJ03a] used for writing dataflow models, more precisely for defining the functionality of dataflow components called *actors*.

The CAL Actor Language was first created as part of the Ptolemy project [EJL+03]. Actors were originally written in Java with a complex API, but ultimately this solution proved unnecessarily difficult to use in practice, and the CAL Actor Language was designed as a result. An actor defines input ports and output ports, a definition of the variables containing its internal state, and a number of transition rules called actions. Each actor executes by making discrete, atomic steps or transitions. In each step, it picks exactly one action from its set of actions (according to the conditions associated with that action), and then executes it, which consists of a combination of the following things:

1. consumes input tokens,
2. produces output tokens,
3. modifies the state of the actor.

The profiled version of CAL Actor Language, called RVC-CAL, has been selected by the ISO/IEC standardization organization in the new MPEG Reconfigurable Video Coding (RVC) standard [ISO09]. RVC-CAL, compared with CAL, restricts the data types, the operators, and the features that can be used in actors. One reason for creating RVC-CAL is the difficulty of implementing a code generator that outputs correct and efficient code for CAL in its entirety. Another rationale for RVC-CAL is to make sure both software code and hardware code can be efficiently generated from CAL. However, the original expressiveness of the language and the strong encapsulation features offered by the actor programming model have been preserved and provide a solid foundation for the compact and modular specification of actors.

Two tools support the CAL language. The OpenDF framework allows to simulate network of CAL actors and generate HDL description, while ORCC (Open RVC CAL Compiler) supports only the RVC-CAL subset but generates C, LLVM, HDL code.

6.4.10 MCSE specification model

MCSE [CI94] (Methode de Conception de Système Electronique) is a french acronyms that stands for Electronic System Design Methodology. It was developped at the polytechnic school of Nantes (France) by Jean Paul Calvez. This methodology aims at providing the system engineer with a work-flow for systems design. This work-flow guides him/her from the customer requirement to a valid prototype. This process goes through several design steps associated to a relevant model. The process steps (Fig.) are:

1. Specification
2. Functionnal Design
3. Implementation specification
4. Implementation

The model associated with each step refines the design to lead to an execution model that can be simulated and synthesized. The functional model is an application model that mixes both data-flow and event driven models (Fig.). The use of the shared variables makes the model closer to the DDF model as a variable can be accessed without consuming it. No formal verification is made on the model behavior but statistic such as operator load and memory footprint can be extracted.

The conception flow allows to validate the designer choice at each step to lead to a valid solution. Documenting each step also eases the communication inside the team and with the customer.

6.4.11 Conclusion

This is an extensible presentation of the application modeling tools that can be used in a rapid prototyping design flow. All tools do at least model simulation or code generation but a few of them perform both. The Figure 6.19 tries to classify the tools with other well know multi-core programming tools. OpenMP (Open Multi-Processing) [DMI98] is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. Cilk is a language for multithreaded parallel programming based on ANSI C. Cilk [BJK+95b] is designed for general-purpose parallel programming, but it is especially effective for exploiting dynamic, highly asynchronous parallelism, which can be difficult to write in data-parallel or message-passing style. The PREESM software is classified as a simulation tool because it is used as a front-end for a Texas Instrument internal tool called NERIOS that performs cycle accurate DSP simulation for bus contention and processor load analysis. In [PRP+08] we also show that RVC-CAL (CAL subset standardized within MPEG) could be used as a front-end for PREESM. This figure's goal is not to rank the tools as there capabilities and target application are deeply different.

6.5 Multi-processor scheduling

On a multi-processor architecture, the scheduling aims at allocating a set of tasks onto the processor of the architecture, and then determines the start-time of each task on the given processor. Finding such schedule is a NP complete problem.

Scheduling of statical data flow model relies on the analysis of the DAG equivalent of the model. Two nodes of a DAG can only be connected by one arc associated to a weight that represents the cost of the data transfer. Most scheduling algorithm of DAGs are based on the list scheduling [ACD74]. In list scheduling, nodes are assigned a priority and organized in a list with a descending order of priorities. Then the nodes of the list are analyzed for scheduling following the list order. Assigning priority can be performed using different methods.

Two frequently used attributes commonly used to assign priorities are the $t-level$ and $b-level$. The $t-level$ (also referred as As Late As Possible (ALAP)) of a node n_i is the length of a longest path from an entry node to n_i. The length of a path is the sum of all node and all weights along the path. The $b-level$ of a node is the length of a longest path from the node n_i to an exit node. The time complexity for assigning $t-level$ is $O(e+v)$ and the same for $b-level$. The $b-level$ is bounded by the length of the *critical path*. The critical path is the longest path in the DAG.

The $t-level$ attribute of a node may vary during the scheduling of a node while the $b-level$ remains constant. When scheduling a node the weight of its incoming edge can be zeroed if the adjacent connected node is scheduled on the same processor. Thus the length of the path reaching that node varies with consequences on the $t-level$. The $b-level$ may vary when the node has been assigned onto a processor. Most algorithm schedules a node when all the parents have been scheduled, this explains why the $b-level$ remains constant during scheduling.

A List scheduling algorithm defines the priority assigning schemes for the nodes and choice criterion for the processor. In first place the algorithm constructs the ordered list of nodes based on the assigned priorities. In experiments, the relevance of the node order for the schedule length was demonstrated [SS05].Still, most algorithms apply orders based on node levels and the bottom level order exhibits good average performance. Then it works by constructing partial schedules of the nodes of the list. Each node of the list is successively allocated onto a processor of the architecture (mapping), thus each node n is successively added to the partial schedule S_{cur} that consists in the node schedule before n. A node is only scheduled once, and its scheduled can not be modified at later time. The usual criterion for processor assignment is the processor allowing the earliest start time ($t-level$) of the node. The complexity of list scheduling algorithm greatly depends on the number of vertices. A simple list scheduling algorithm like the one presented in [Sin07] has a complexity of $O(P(V+E))$. Other well-known list-scheduling algorithm such as the Highest Level First with Estimated Time [KA98] or the Modified Critical Path [WG90] show a complexity of $O(pv^2)$. Other scheduling algorithms using heuristic technique such as genetic algorithm or simulated annealing exist but usually use a list scheduling algorithm as a starting point.

6.6 Conclusion

A rapid prototyping framework takes advantage of informations contained into a model of the application to perform a matching of the given algorithm onto a given architecture. Depending on the architecture characteristics (PEs, memory system ...), this tool try to obtain the best performance. Considering the amount of architecture available, and the complexity of the algorithms, it is easy to understand that, in order to performs this operation, the tool must give some limitations. Most limitation are imposed by the model expressiveness, and the programmer have to deal with it to express its algorithm behavior. The more limited the model is, the more informations you can get from it at compile time. Some rapid prototyping tools choose to limit to one model (PREESM, SynDEx), and some other only defines a syntax from which multiple Data Flow behavior can be extracted. In the case of the rvc-CAL language, actor are DPN actors but some SDF zone can be determined [GJRB10], and one could take advantage of such zone to generate optimized multi-core code. Other tools such as Ptolemy II describe the application as a composition of models, but the model is only use for simulation, and there is no need to extract information at compile time. The SDF model is very limited in term of expressiveness, but you can get a lot scheduling information from it, which makes it a good choice for rapid prototyping of efficient multi-core code. Optimizing the application before scheduling is also of great importance. This optimizations aims at easing the scheduling step, but also extract more parallelism for better performance on parallel architectures. In the next chapter we give an overview of optimization of nested loops that could become of interest for data flow optimizations.

```
int->int loop matVectProd(int N){
        join roundrobin(1, N) ;
        add vectScalarProd();
        loop propagate();
        split duplicate ;
        enqueue 0.0;
}

int->int filter propagate(){
        prework push N{
                push(0.0);
        }
        work push 1 pop 1{
                push(pop());
        }
}

int->int loop vectScalarProd(){
        join roundrobin ;
        add mac();
        loop propagate();
        split duplicate ;
        enqueue 0.0;

}

int->int filter mac(void){
        work push 1 pop 3{
                a = pop();
                b = pop();
                c = pop();
                push(c * b + a);
        }
}
```

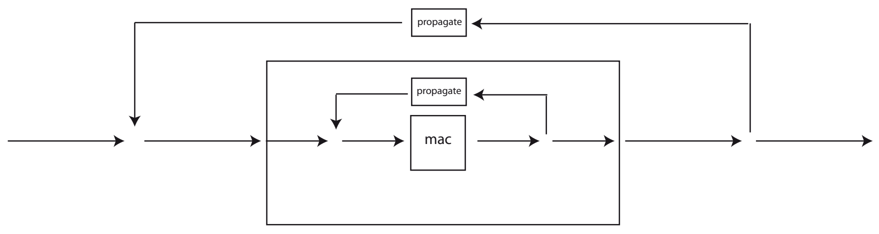


Figure 6.13: *StreamIt model of the matrix vector product example.*

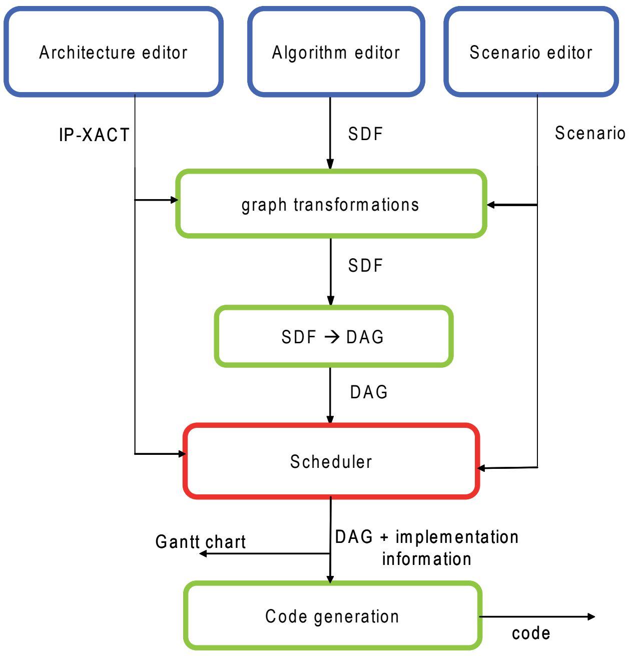
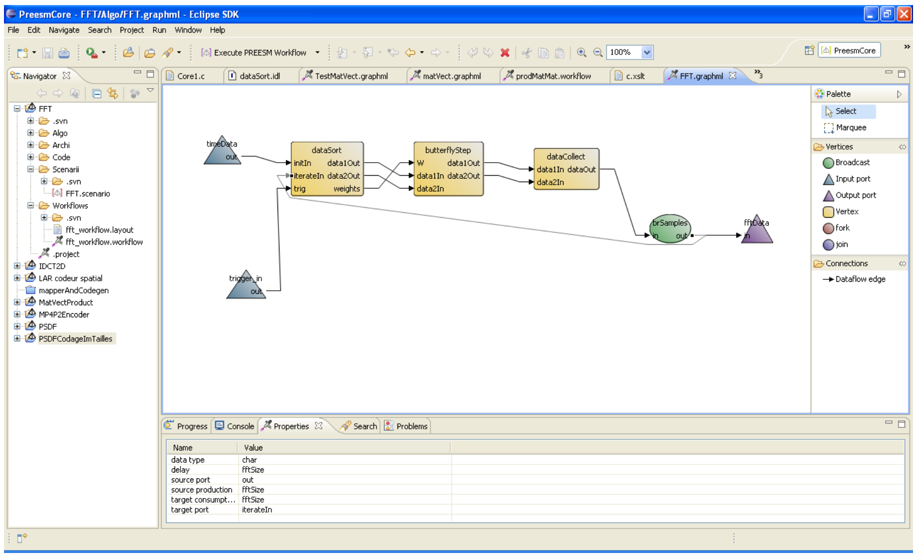


Figure 6.14: *From SDF and IP-XACT descriptions to code generation.*



Name	Value
data type	char
delay	fftSize
source port	out
source production	fftSize
target consumpt...	fftSize
target port	iterateIn

Figure 6.15: *Screen shot of the PREESM framework.*

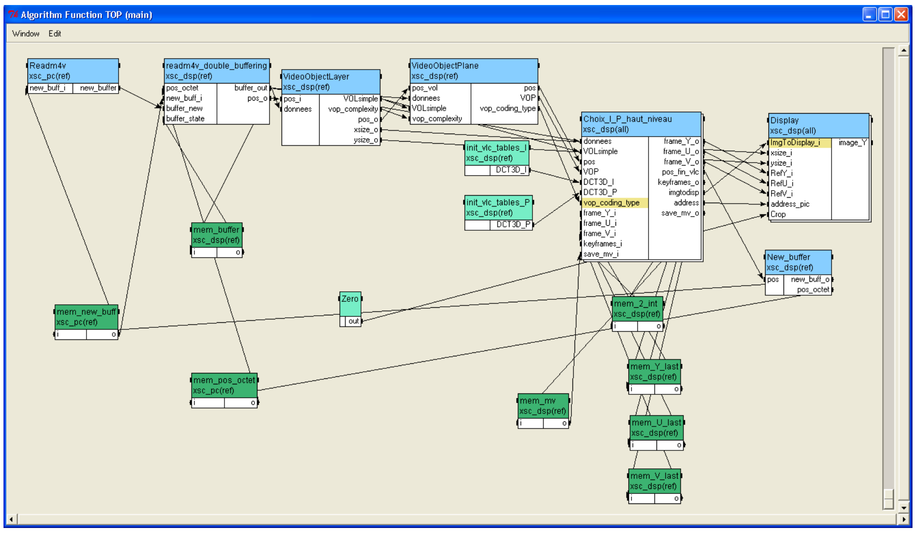


Figure 6.16: *MPEG-4 decoder model specified in SynDEx.*

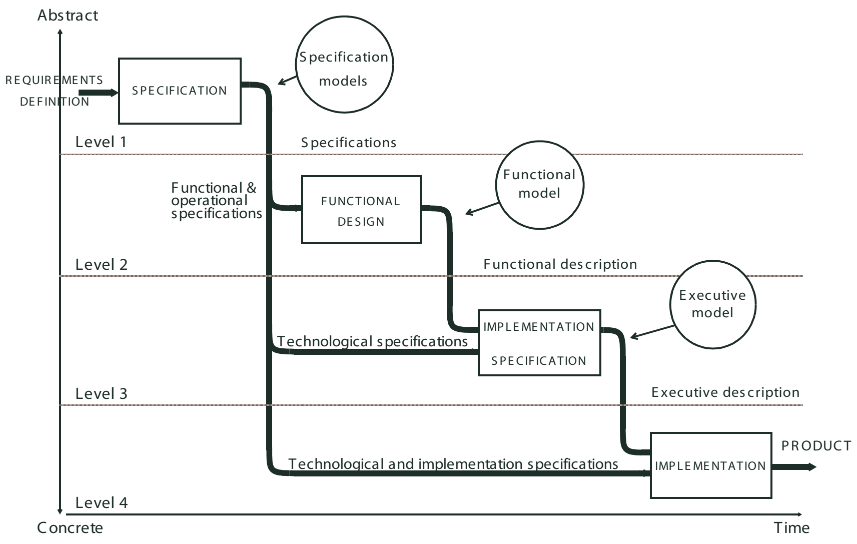


Figure 6.17: *The MCSE design cycle.*

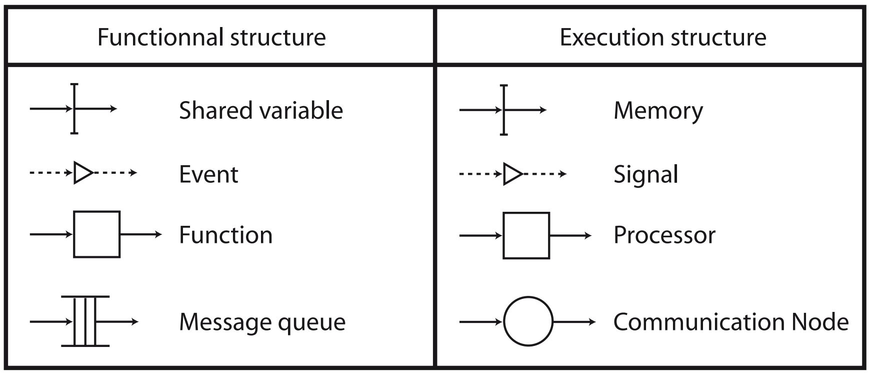


Figure 6.18: *The MCSE model components.*

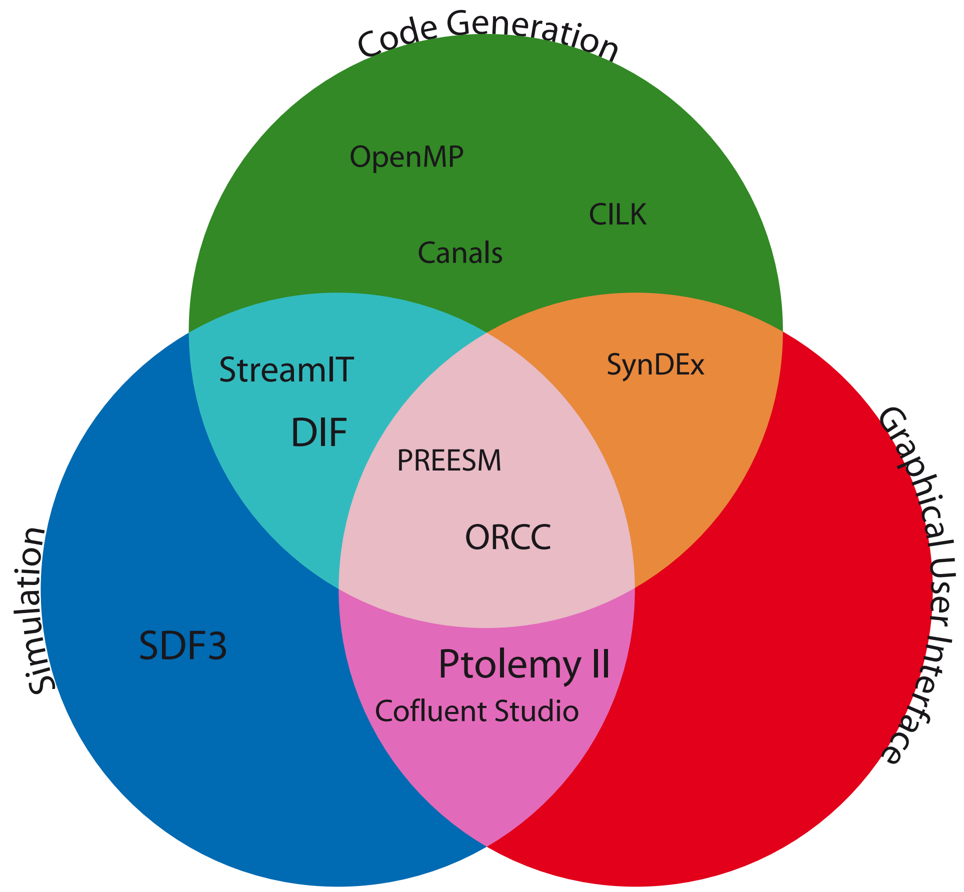


Figure 6.19: *Overview of the application modeling tools and their different abilities*

Chapter 7

Nested loops Partitioning

7.1 Introduction

Nested loops are widely used as logical structure in imperative programming language such as C. This structure aims at factorizing computation on hierarchical or multi-dimensional data. A nested loop is formed by nested repeating (loops) statements. This kind of structure operates in a highly sequential way, as one outer loop iteration triggers the iteration over its whole domain of the inner loops. Thus it is important for compilers to be able to optimize this type of structure to execute it in an efficient way. Moreover this is vital for parallel architecture compilers to be able to extract the parallelism nested into the hierarchy. This transformations are based on information extracted from the nested loop specification. Those information characterize the reads and writes order on memory locations. It helps at ensuring that even if the nested loop is not executed in a sequential way or in an out-of-order manner, the resources accessed in inner-most loop statements will remain the same. In the following we present the nested loops structure, and the dependency representations. Then we introduce the transformation techniques that aims at extracting parallelism from the nested loops representation.

7.2 Nested Loops representation

Definition A nested loop of depth n is a structure composed of n nested loops for which each loop, excluded the n^{th} one, contains only a loop (Fig. 7.1).

The iteration domain of the outer loop remains constant while the iteration domain of inner loops consists in maxima and minima of several affine functions.

Three types of dependencies exist:

```
for $i_1$ := $l_1$ to $u_1$ do
 for $i_2$ := $l_2(i_1)$ to $u_2(i_1)$ do
  ...
  for $i_n$ := $l_n(i_1, i_2, ..., i_{n-1})$ to $u_n(i_1, i_2, ..., i_{n-1})$ do
   $\{Instruction1\}$
   ...
   $\{Instructionk\}$
  end
 end
end
```

Figure 7.1: *Nested loop example.*

- Flow dependence: a statement S2 is flow dependent on S1 (written) if and only if S1 modifies a resource that S2 reads and S1 precedes S2 in execution.
- Anti-dependence: a statement S2 is anti-dependent on S1 (written) if and only if S2 modifies a resource that S1 reads and S1 precedes S2 in execution.
- Output dependence: a statement S2 is output dependent on S1 (written) if and only if S1 and S2 modify the same resource and S1 precedes S2 in execution.

Analyzing those dependencies can rely on a model that can be treated for optimizations.

In the following we will be using the "distance vector" as a dependency representation. A distance vector represents the flow dependency between two operations along the iteration domain. In nested loops of depth N, given two accesses to the same data in the flow order by two instructions S_1 and S_2, with $S_1 \Rightarrow S_2$ with respective index vector $\vec{p}$ and $\vec{q}$. The distance vector $\vec{\delta}$ is an N dimensional vector. For a flow dependency $S_1[\vec{p}] \Rightarrow S_2[\vec{q}]$ the distance vector is $\vec{\delta} = \vec{p} - \vec{q}$. This specific representation allows using linear algebra to perform analysis and optimizations.

```
for $i$ := 0 to $N$ do
 for $j$ := 0 to $N$ do
        S1 : a(i, j) = b(i, j - 6) + d(i - 1, j + 3)
        S2 : b(i + 1, j - 1) = c(i + 2, j + 5)
        S3 : c(i + 3, j - 1) = a(i, j - 2)
        S4 : d(i,j-1) = a(i,j-1)
  end
 end
end
```

Figure 7.2: *Nested loops example.*

In the example Fig. 7.2, the iteration vector of S_1 for a is $p_{S_{1a}} = (i, j)$, and the iteration vector of S_3 for a is $q_{S_{3a}} = (i, j - 2)$. Thus the resulting distance vector is:

$$S_1 \rightarrow S_3 \quad d_1 = \begin{pmatrix} 0 \\ 2 \end{pmatrix}$$

The other dependency vectors for this example are:

$$S_3 \rightarrow S_2 \quad d_2 = \begin{pmatrix} 1 \\ -6 \end{pmatrix}$$

$$S_2 \rightarrow S_1 \quad d_3 = \begin{pmatrix} 1 \\ 5 \end{pmatrix}$$

$$S_1 \rightarrow S_4 \quad d_4 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$S_4 \rightarrow S_1 \quad d_5 = \begin{pmatrix} 1 \\ -4 \end{pmatrix}$$

These vector can be organized into a dependency matrix D.

7.3 Nested Loops execution optimization

Optimizing nested loops aim at extracting parallelism [DV95], by transforming the loops structure. Those transformations can be any of the five types described below.

- Loop distribution: this transformation aims at distributing the nested loop to extract at least one forall loop [Lam74].
- Loop fusion: this transformation aims at fusionning several loops body into one unique loop [Man97].
- Loop unrolling: this transformation aims at unrolling a loop to extract the inter-iteration parallelism [KA01] [CDS96].
- Loop partitioning: this transformation aims at partitioning the loops to extract disjoint iteration domain [KKBP91].
- Unimodular Transformation: this transformation aims at modifying the iteration domain resulting in an out–of–order iteration [WL91].

Hyperplan Method

In 1974 Lamport [Lam74] developed a method consisting in transforming a nested loop into a nested loop for which some or all the intern loops can be computed in parallel. The optimal case being the one for which all the intern loop can be computed in parallel.

In Figure 7.3, the outer loop is one iteration of the nest, while $E(t)$ represents the iterations computed at time t and $P(p)$ all the iterations of vector p computed in parallel. The main issue of this transformation is to find a sequencing function that

```
for $t$ := 0 to $t_N$ do
 for $i_2$ := $l_2(i_1)$ to $u_2(i_1)$ do
   ...
   for $p \in E(t)$ do in parallel
     $P(p)$
   end
 end
end
```

Figure 7.3: *Transformed nested loops, with some inner loops being computed in parallel.*

satisfies the dependencies between iterations to preserve the computation integrity. In [Lam74], the author proposes to restrict the analysis to the case of a sequencing function linear to the points of the iteration domain. This is achieved by finding a set of affine parallel hyperplane, such as the points computed at time t be $E(t) = H(t) \cap D$, D being the iteration domain. The translation from an hyperplane to the next $H(t) \Rightarrow H(t+1)$ is performed by translating the hyperplane with a vector π called the time vector. Thus in the case of a uniform nested loop it is made easy to find a valid time vector. Such vector only needs to verify $\pi d \geq 1$ for any dependency vector d. The point computed at time t belongs to the hyperplane solution of $E(t) = \{p; Ap \leq b; [\pi p] = t\}$. The computing time of the transformed loop is then $T = \pi p_{max}$.

Lamport proposes the following technique to find the optimal time vector. Let us consider $D = (d_1, ..., d_m)$ the matrix of dependency vectors of size $n \times m$ where n is the size of the nested loop and m the number of dependency vectors. We suppose D organized in the lexicographic increasing order (from the shorter dependency to the largest). Be k_1 the first non null element of d_1 the first vector of D. As d_1 is positive $d_{1,k_1} \geq 0$. Let π_{k_1} and $\pi_k = 0$ for $k_1 \neq k \neq n$. Let k_2 be the first non-null element of d_2. As d_1 is lexicographically less than d_2 we can assume that $k_2 \leq k_1$. Let $\pi_k = 0$ for $k_2 \leq k \leq k_1$ and π_{k_2} be the the smallest integer such as $\pi.d_2 \geq 0$. If $k_2 = k_1$ we modify π_{k_1}. By applying the same procedure iteratively we find a valid time solution.

Other hyperplan method have been proposed such as selective shrinking [Pol88], true dependence shrinking [PC89] or unimodular transformations [Wol90]. Those methods aim at finding a special kind of time vector.

When a time vector has been computed transformation methods can be used to transform the nested loops by modifying the iteration domain, and the index in the inner-most loop instructions [LHS90].

7.4 Nested Loops partitioning by iteration domain projection

This nested loop partitioning technique was developed as a method for systolic array synthesis in [MF86]. A systolic array is massively parallel computing network. This network is composed of a set of cells locally connected to their spatial neighbors. All the cells are synchronous to a unique clock. For each clock cycle, a cell takes data from its incoming edges, performs a computation and outputs data to its outgoing cells. This partitioning technique aims at finding a projection vector by analyzing the distance vector of a nested loop of depth N. When this projection vector has been determined, the iteration domain is projected along this vector resulting in an $N-1$ dimension systolic array.

To determine the projection vector we must first determine a time vector τ that satisfies the data dependencies. Computing its time vector is performed as described in section 7.3. For a given τ vector, the parallel execution time of the nested loop, can be determined by $t_{parallel} = \tau * p_{max}$, p_{max} being the coordinate of the last point of the domain. In [MF86] Lamport proposes a solution to find one τ vector for which the parallel execution time is minimal.

Given a valid time vector, a valid projection vector s_n verify $\tau s_n \neq 0$. Let us now complete the s_n vectors into a unimodular matrix S_n, s_n being the first column of the matrix. The matrix S_n is the space base, so we project the computation domain along the first column of S_n onto the hyperplane generated by the $n-1$ other vectors of the matrix. Coordinates of a point p of the computation domain into the space base S_n are $S^{-1}p$. Thus, the allocation function is the lower $(n-1) \times n$ sub-matrix of S^{-1}, which corresponds to the $n-1$ last coordinates in the space base.

Using this allocation matrix we can now determine how the domain points are allocated onto the computing network. Further analysis using the time vector and distance vectors also allows figuring out the communication activity over the network and the computation activity of each cell in the network.

By completing the vector τ into a unimodular matrix T^{-1}, T is the time base of the computation domain. The last $n-1$ column vectors of T forms the base of the hyperplane of the point computed at time 0, while the first column, is the translation vector that enables to go from the hyperplane of the point computed at t to the hyperplane of the point computed at $t+1$. The activity translation vector corresponds to the $n-1$ elements of the first column vector of the product $S^{-1}.T$, and the $(n-1) \times (n-1)$ sub-matrix is the activity base.

Example

In this section we will apply the projection technique on the matrix vector product example (Fig.). Given a vector V and matrix M, the product $VxM = R$ can be described using a set of recurrent equations.

$$\begin{cases} R_{i,k} = 0 & if \quad k = 0 \\ R_{i,k} = R_{i,k-1} + v_i m_{i,k} & if \quad 1 \leq k \leq N \\ r_i = R_{i,N} & 0 \end{cases}$$

from this set we can extract the following system.

Initial state

$$\begin{cases} R_{i,k} = 0 & if \quad k = 0 \\ V_{i,k} = v_k & if \quad i = 0 \\ M_{i,k} = m_{i,k} \end{cases}$$

Calculus equations

$$\begin{cases} R_{i,k} = R_{i,k-1} + V_{i-1,k} M_{i,k} & i \quad 1 \leq k \leq N \\ V_{i,k} = V_{i-1,k} & i \quad 1 \leq k \leq N \\ M_{i,k} = m_{i,k} \end{cases}$$

Output equation

$Ri = R_{i,N}$

From this representation we can extract two distance vectors: $d_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ $d_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$

Using Lamport method we can determine the optimal time vector as being $\tau = (1 \quad 1)$. Knowing the time vector we can now find a set of projection vectors.

$$s_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \; s_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \; s_3 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Using those three vectors we can now determine the execution domain (Figure 7.5). The three projection vectors result in three different patterns of execution. Vectors s_1 and s_2 result in pipelined execution, while s_3 brings more parallelism, with an unbalanced computation load. Going through this analysis allows finding a trade off between parallelism and factorization by using nested loops intrinsic parallelism, without unrolling the loop.

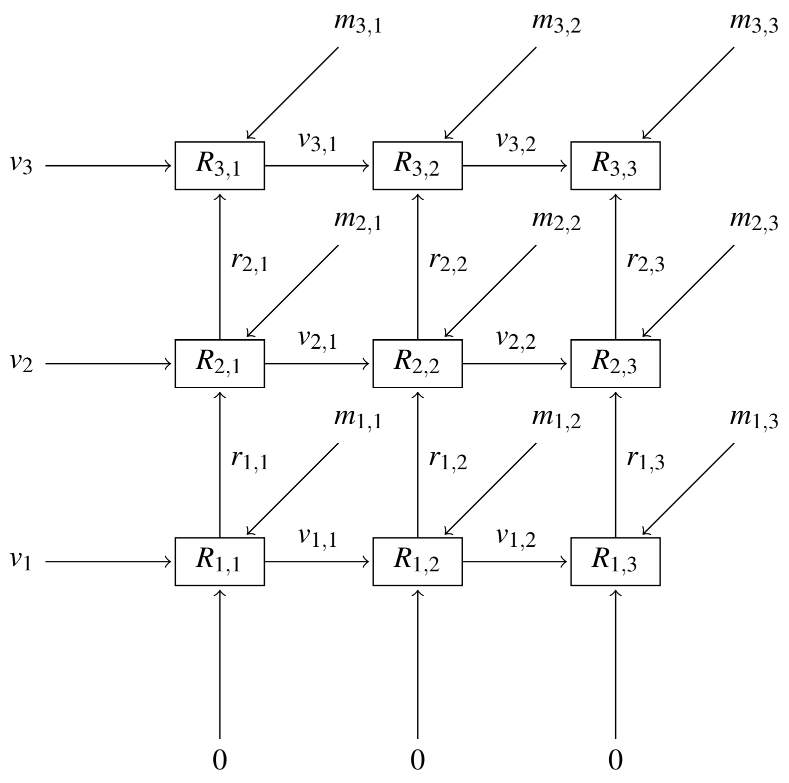


Figure 7.4: *The matrix vector iteration domain with dependencies for a* 3×3 *matrix.*

7.5 Nested Loops partitioning by iteration domain tiling

Loop partition partitions the loop iteration domain into small chunks or blocks that group several computation points. This technique is used to ensure that the computation performed by a block does not need synchronization thus minimizing communication overhead and increases granularity.

Constructing valid tiles can be performed by two equivalent ways.

- Cutting hyperplanes: Consider n linearly independent vector h_i. A tile is the translated copy of the canonical tile defined as follow: a node p belong to the tile if and only if :
  $\forall i \in [1, n]\ 0 \leq h_i.p \leq \beta_i$
  with β_i being the thickness of the tile. The thickness of the tile can either be chosen by setting β_i to 1 and scaling h_i or by choosing h_i with euclidean norm one and scaling β_i.
- Change of basis: Consider n linearly independent vector p_i to define the edges of the tile. The tile is then the set of nodes whose coordinates are positive and strictly less than one in the basis defined by the p_i's. If P denotes the matrix whose column vector are the p_i's, then this definition is equivalent to

the previous one with $P^{-1} = H$.

Tiling is simply paving the domain with translation of the canonical tile defined above. Classical constraints on tiles in the literature are the followings:

- Tiles are bounded: For scalability reason we want the number of point in a tile to be bounded by a constant independent of the domain size.
- Tiles are identical by translation: This means that P is an integral matrix.
- Tiles are atomic: Each tile is a unit of computation. Computation between tiles is made at the end of tiles. The order on tiles must correspond to the order on nodes. It means that one must avoid that two different tiles depend on each other. This condition can be mathematically expressed by $HD \geq 0$.

Tiling a domain results into two metrics that allow finding the optimal tiling. V_{calc} is the number of nodes in a tile and thus represents both the computation load, and the inverse of the parallelism loss. V_{com} is the communication load on the network for tile synchronization. An optimal tiling method will try to conjointly optimize these two criteria for a given architecture.

- V_{calc} the volume of computation in a tile is defined as follows: $V_{calc}(T) = |detP| = \frac{1}{|detH|}$
- V_{com} the volume of communication in a tile is approximated as follows: $V_{com}(T) = \frac{1}{|detH|}\sum_{i=1}^{n}\sum_{j=1}^{m}(h_i.d_j) = \dfrac{1}{|detH|}\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{n}(h_{i,k}d_{j,k})$

The matrix vector product example:

Taking the equations of the matrix vector product described above. we can extract the matrix $D = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. Then every tile for which $HD \geq 0$ is a valid tile.

In Figure 7.6 the matrix describing the tiles are : $P_1 = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$

$P_2 = \begin{pmatrix} 2 & 0 \\ -2 & 2 \end{pmatrix}$

$P_3 = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$

We must first ensure that the three matrix verify $HD \geq 0$ with $H = P^{-1}$:

$H_1D = \frac{1}{2}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$

$H_2D = \frac{1}{2}\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$

$H_3D = \frac{1}{2}\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$

Those three tiles being valid, we can now compute the computation volume and communication volume of each tile.

$V_{calc}(T_1) = 4$
$V_{calc}(T_2) = 4$
$V_{calc}(T_3) = 2$

$V_{com}(T_1) = 4$
$V_{com}(T_2) = 6$
$V_{com}(T_2) = 3$

This three different tile shapes show the different computation load with different communication volume. Finding an optimal tiling is a trade-off between granularity (volume of computation) and communication load (volume of communication). Several optimization methods exist in the literature all adapted to specific target machine.

In [BDRR94] the author proposes a new criterion to define optimality. This criterion being scalable it is efficient to use with a wide class of machines. In this paper the authors describe the "ideal" tiling procedure to be following:

- For any fixed value of V_{calc}, find H which minimizes the value of V_{com}.
- Set the value of V_{calc} in accordance to local memory requirements.

  Consider now the following expression:
  $V(H) = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{n} h_{i,k}d_{k,j}}{|det(H)|^{\frac{1}{n}}}$
  Scaling H by a factor λ ($\lambda \geq 0$) gives:
  $V(\lambda H) = \lambda \left|\frac{det(H)}{det(\lambda H)}\right|^{\frac{1}{n}} = V(H)$

The authors prove that minimizing $V(H)$ and scaling the solution to obtain a fixed volume of computations minimizes also V_{com} for the given volume of computation. Thus the tiling problem only relies on minimizing $V(H)$ and scale H to fit the available volume of computation. Finding H is described as a combinatorial problem. In the paper authors prove that any matrix whose rows are n linearly independent b_i's is a minimum solution of $V(H)$. The b_i's are vectors that generate the polyhedral cone $C = \{x|b_1^T x \geq 0, ..., b_t^T x \geq 0\} = \{H|HD \geq 0\}$. If D is a square matrix the matrix H_0 for which $V(H_0) = V_{min}$ is $H_0 = D^{-1}$.

7.6 Conclusion

The nested loop transformation techniques presented show good parallelism extraction performance with minor complexity. The domain projection technique is the simplest of the two. It is flexible considering that you can choose any projection vector that respects the $\pi D \geq 0$ condition, but the resulting network size still depends on the iteration domain size. Finding an optimal solution for the tiling method is more complicated as the tile is a N dimension shape whereas the projection vector is a $N - 1$ dimension shape in the projection technique. Still the tiling technique has more flexibility, as one can adapt the tile size to fit its available computation volume, and communication volume. Moreover when using the tiling technique, the resulting network size does not depend on the iteration domain size, but only on the tile shape and size. Combining the two techniques might also be a good solution. One could first tile the network, and then use the projection technique to schedule the tile network. The tile scheduling using the domain projection technique could lead to execution pattern such as vectorized, or pipelined execution that have proved to be efficient on parallel architectures.

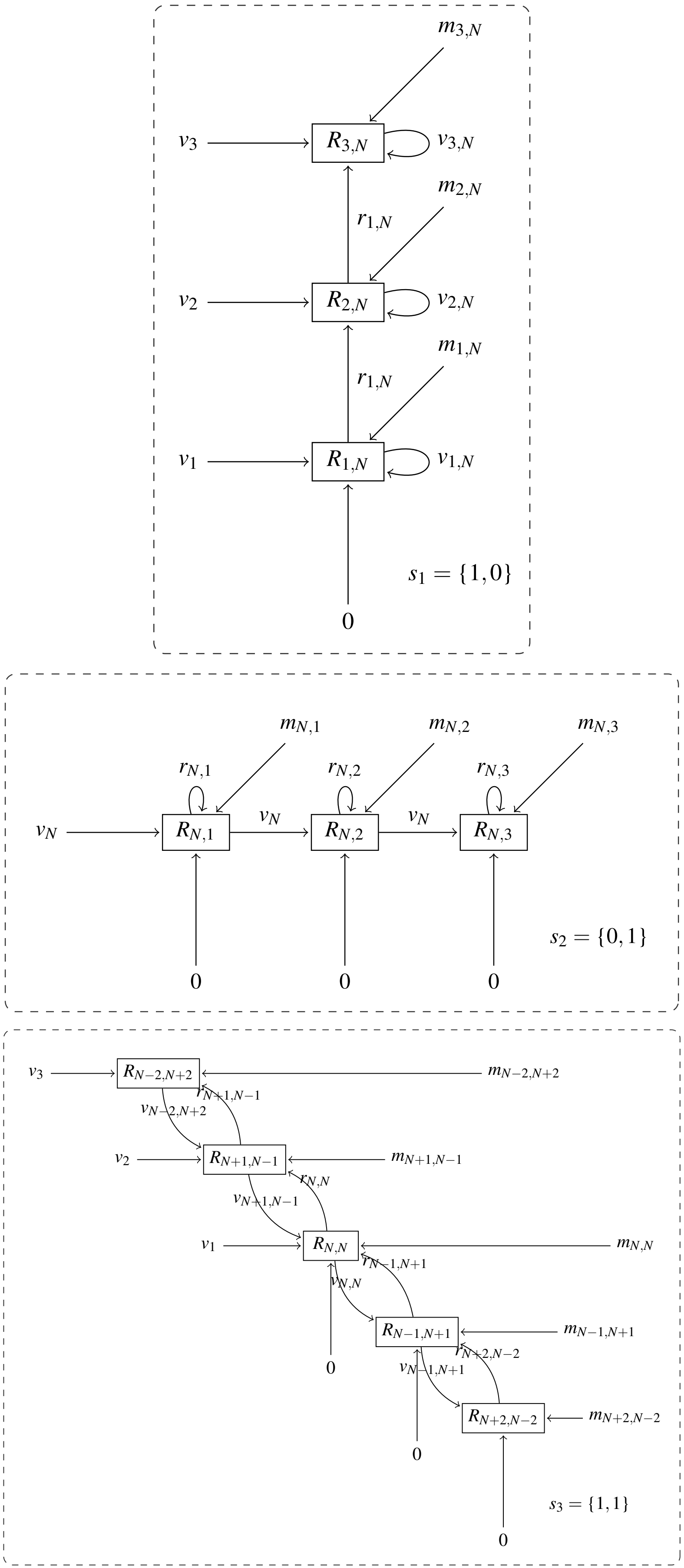


Figure 7.5: *Projections of the iteration domains along the three projection vectors.*

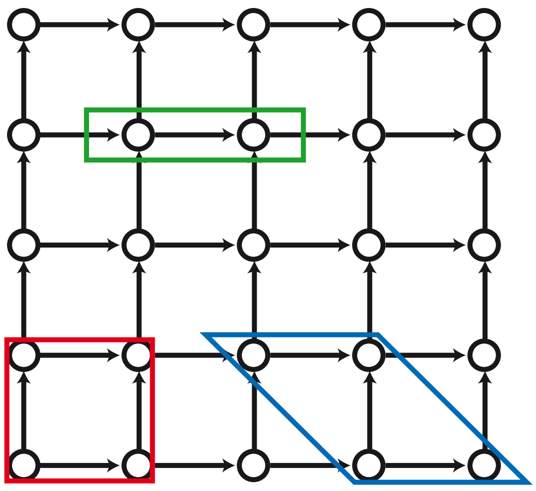


Figure 7.6: *The matrix vector product iteration domain with different tile .*

Part III

Research Work

Chapter 8

Hierarchy Representation in Synchronous Data Flow Graphs

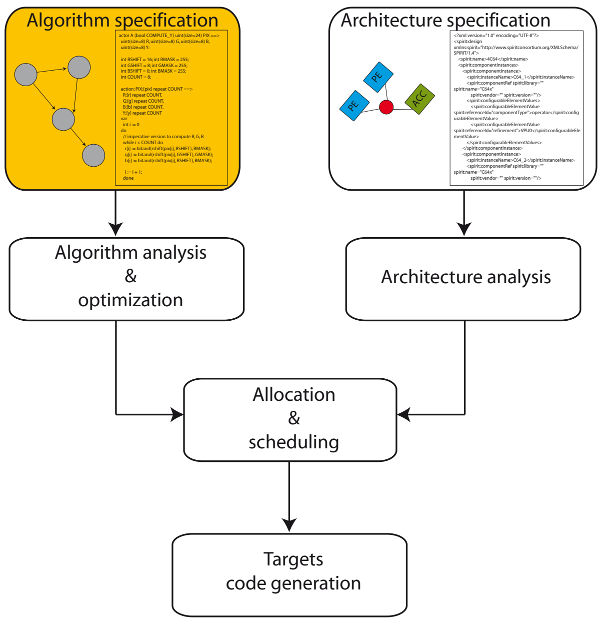
8.1 Introduction

Application modeling requires to describe the application in respect to the model rules at a given granularity. By granularity we refer to the level of abstraction of the application structure/behavior. For example any signal processing algorithm could be modeled at the finest grain using only elementary operations such as additions and multiplications. A coarser grain would model an algorithm using signal processing operation such as FFT, DFT, Filters. In order to provide the designer with a way to model an application as a composition of subsystems modeling the application at different grain levels, it is usual to make use of a hierarchical model. The top most level represents the application at its coarser grain while the bottom most level represents the finest level of modeling. The finest level models the application with a higher complexity level and contains more information on the application behavior that can be useful for later analysis. The top-most hierarchy level has less complexity but does not give as much information. A hierarchical model offers more flexibility for better analysis as a trade-off between complexity and available information is possible. This definition of the hierarchy matches the definition of the *nested hierarchy* that can be illustrated by Russian matryoshka dolls. Each hierarchy level contains and consists of sub-sets that can either contain themselves a hierarchy level or be atomic. From the data flow point of view, the hierarchy can be defined as follow: a data flow actor behavior can either consists in a Data Flow process network or be atomic. Thus the mapping function of the actor f can itself be described as a network of data flow processes which input sequence is a firing rule of the actor. It means that $map(f) = F(R : X) = f(R) : F(X) = F_sub(X_sub, I) : F(X)$, where F_sub is the mapping function of the sub network and X_sub all the sequences in the sub network.

In the following we will consider a data flow hierarchy representation has the three following properties:

1. Data locality: The flow of data in a hierarchy level is local and thus cannot be modified nor accessed by the upper hierarchy levels.
2. Schedule composition: The schedule of a network is the composition of the schedule of the top most network and its inner networks.
3. Atomicity: A hierarchical actor remains atomic from its parent network point of view. Thus the schedule of the inner network of the actor does not need to be interrupted by the parent schedule.

Rapid prototyping framework (Fig. 8.1) can also take advantage of such model to analyze the model at a relevant hierarchy level regarding to the kind of parallelism available on the target architecture. This hierarchical representation also has a significant impact on the scheduling complexity that greatly depends on the number of vertices to schedule at the top level ($O(P(V+E))$ or $O(pv^3)$).



```
actor A (bool COMPUTE_Y) uint(size=24) PIX ==>
 uint(size=8) R, uint(size=8) G, uint(size=8) B,
 uint(size=8) Y:

 int RSHIFT = 16; int RMASK = 255;
 int GSHIFT = 8; int GMASK = 255;
 int BSHIFT = 0; int BMASK = 255;
 int COUNT = 8;

 action: PIX:[pix] repeat COUNT ==>
  R:[r] repeat COUNT,
  G:[g] repeat COUNT,
  B:[b] repeat COUNT,
  Y:[y] repeat COUNT
 var
  int i := 0
 do
  // imperative version to compute R, G, B
  while i < COUNT do
   r[i] := bitand(rshift(pix[i], RSHIFT), RMASK);
   g[i] := bitand(rshift(pix[i], GSHIFT), GMASK);
   b[i] := bitand(rshift(pix[i], BSHIFT), BMASK);

   i := i + 1;
  done
```

```
<?xml version="1.0" encoding="UTF-8"?>
<spirit:design
xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/
SPIRIT/1.4">
  <spirit:name>4C64</spirit:name>
  <spirit:componentInstances>
    <spirit:componentInstance>
      <spirit:instanceName>C64_1</spirit:instanceName>
      <spirit:componentRef spirit:library=""
spirit:name="C64x"
        spirit:vendor="" spirit:version=""/>
      <spirit:configurableElementValues>
        <spirit:configurableElementValue
spirit:referenceId="componentType">operator</spirit:config
urableElementValue>
        <spirit:configurableElementValue
spirit:referenceId="refinement">VPU0</spirit:configurableEle
mentValue>
      </spirit:configurableElementValues>
    </spirit:componentInstance>
    <spirit:componentInstance>
      <spirit:instanceName>C64_2</spirit:instanceName>
      <spirit:componentRef spirit:library=""
spirit:name="C64x"
        spirit:vendor="" spirit:version=""/>
```

Figure 8.1: *Application modeling in the Rapid Prototyping workflow.*

The Synchronous Data Flow Model defines the hierarchy as a group of actors in a graph, that can be seen as a single actor from a higher point of view. This hierarchy does not define a specific behavior, nor specific scheduling rules. In contrary, the

Parameterized Synchronous Data Flow (PSDF) clearly defines the hierarchy behavior, with the concept of local synchrony, but does increase a lot the complexity of the specification, and as a consequence, decreases the analyzability of the model. In order to increase the expressiveness of the SDF model we have chosen to define a new kind of hierarchy that implies a specific behavior and specific scheduling rules. This model stands as the Interface-based SDF and relies on the specification of interfaces for each level of hierarchy with a precisely defined behavior (Fig. 8.2).

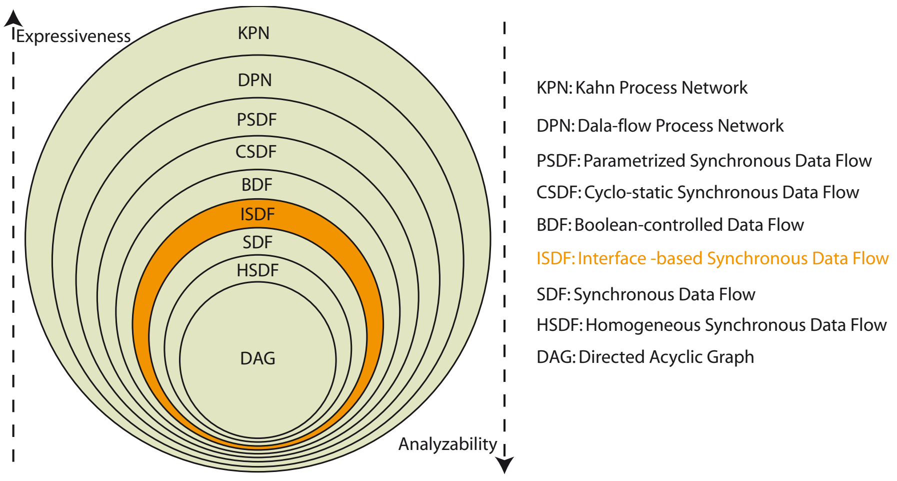

Expressiveness
KPN
DPN
PSDF
CSDF
BDF
ISDF
SDF
HSDF
DAG
Analyzability
KPN: Kahn Process Network
DPN: Dala-flow Process Network
PSDF: Parametrized Synchronous Data Flow
CSDF: Cyclo-static Synchronous Data Flow
BDF: Boolean-controlled Data Flow
ISDF: Interface -based Synchronous Data Flow
SDF: Synchronous Data Flow
HSDF: Homogeneous Synchronous Data Flow
DAG: Directed Acyclic Graph

Figure 8.2: *The integration of the Interface-based SDF in the data flow taxonomy.*

8.2 Existing hierarchy representation in Synchronous Data Flow Graphs

8.2.1 Repetition based hierarchy

Hierarchy has been described in [PBL95], as a mean of representing cluster of actors in a SDF graph. In [PBL95] clustering is used as a pre-pass for the scheduling described in [HPB08] that reduces the number of vertices in the DAG, minimizing synchronization overhead for multi-threaded implementations and maximizing the throughput by grouping buffers [HPB08]. Given a consistent SDF graph, this approach first clusters strongly connected components to generate an acyclic graph. A set of clustering techniques is then applied based on topological and data flow properties, to maximize throughput and minimize synchronizations between clusters. This approach is a bottom-up approach, meaning that the starting point is a SDF graph with no hierarchy and it automatically outputs a hierarchical (clustered) graph. In order to ensure that clustering an actor may not cause the application to be deadlocked, the authors (in [PBL95]) describe five composition rules based on the data flow properties (Figure 8.3).

Considering a consistent connected SDF Graph G, and (x, y) is an ordered pair of distinct adjacent nodes in G. Then $cluster(\{x, y\}, G)$, the graph resulting from clustering $\{x, y\}$ into a single node Ω is consistent if the following four conditions hold true. (Figure 8.3).

- Cycle introduction condition: [PBL95] there is no simple path from x to y that contains more that one arc. A simple path is the one which does not visit any node along the path more than once.
- Hidden delay condition: [PBL95] if x and y are in the same strongly connected component then both **a** and **b** must hold true.
  - **a** at least one arc from x to y has zero delay.
  - **b** for some positive integer k, $q(x) = kq(y)$ or $q(y) = kq(x)$.
- First precedence shift condition: [PBL95] if x is in a non trivial strongly connected component C, then either a or b must hold true.
  - **a** : for each arc $\alpha \in \left\{ \alpha' \| \begin{pmatrix} (snk(\alpha') = x) \\ and \\ (src(\alpha') \in C) \\ and \\ (src(\alpha') \notin \{x, y\}) \end{pmatrix} \right\}$ with $snk(\alpha')$ and $src(\alpha')$ respectively being the sink and the source of the arc α', there exist integers $k_1 > 0$ and $k_2 \geq 0$ such that $\rho_\alpha = k_1 Q(x, y)\kappa\alpha$ and $\delta_\alpha = k_2 Q(x, y)\kappa\alpha$
  - **b** : for each arc $\alpha \in \left\{ \alpha' \| \begin{pmatrix} (src(\alpha') = x) \\ and \\ (snk(\alpha') \in C) \\ and \\ (snk(\alpha') \notin \{x, y\}) \end{pmatrix} \right\}$ there exist integers $k_1 > 0$ and $k_2 \geq 0$ such that $\kappa_\alpha = k_1 Q(x, y)\rho\alpha$ and $\delta_\alpha = k_2 Q(x, y)\rho\alpha$
- Second precedence shift condition: [PBL95]
  - **a** : for each arc $\alpha \in \left\{ \alpha' \| \begin{pmatrix} (snk(\alpha') = y) \\ and \\ (src(\alpha') \in C) \\ and \\ (src(\alpha') \notin \{x, y\}) \end{pmatrix} \right\}$ there exist integers $k_1 > 0$ and $k_2 \geq 0$ such that $\rho_\alpha = k_1 Q(x, y)\kappa\alpha$ and $\delta_\alpha = k_2 Q(x, y)\kappa\alpha$
  - **b** : for each $\alpha \in \left\{ \alpha' \| \begin{pmatrix} (src(\alpha') = y) \\ and \\ (snk(\alpha') \in C) \\ and \\ (snk(\alpha') \notin \{x, y\}) \end{pmatrix} \right\}$ there exist integers $k_1 > 0$

and $k_2 \geq 0$ such that $\kappa_\alpha = k_1 Q(x,y)\rho\alpha$ and $\delta_\alpha = k_2 Q(x,y)\rho\alpha$

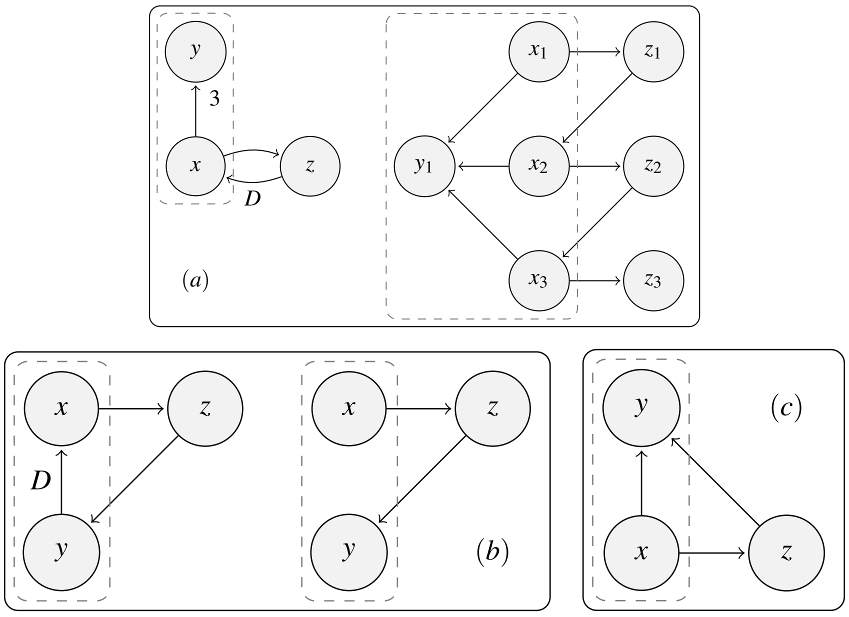


Figure 8.3: *Pino's clustering rules:* ***(a)*** *illustrates the violation of the first precedence shift condition,* ***(b)*** *illustrates the violation of the hidden delay condition, and* ***(c)*** *illustrates the violation of the cycle introduction condition.*

All those conditions ensure that clustering nodes does not introduce delay in the precedence graph. A cluster verifying all the given conditions can be declared valid for an ordered pair of vertex $\{x, y\}$.

8.2.2 Parameter based hierarchy

Parameter-based SDF hierarchy has been introduced in [BB01] where the authors introduce a new SDF model called *Parameterized SDF*. This model aims at increasing SDF expressiveness while maintaining its compile time predictability properties. The model behavior and rules are fully depicted in section 6.3.7. This model can be considered as hierarchy model within the SDF model, as it allows to describe an actor as a sub-network that is locally SDF. This is explained by the fact that for an invocation of the actor it has an SDF behavior, but its behavior can vary between invocations. From the upper hierarchy level point of view the network containing this sub-network remains SDF for a full iteration of the schedule as the sub-network interfaces can only be affected by the init graph at its instantiation. This means that one iteration of the schedule can execute without considering the sub-network behavior (Schedule composition). A PSDF specification of an actor is in essence atomic as its schedule is independent from its parent network schedule. The data inside a PSDF specification can be considered local as they are only shared between the actor of the network. Thus the PSDF model has the three properties defined in the introduction and can be considered as a hierarchy representation in SDF.

8.3 Interface based hierarchy

While designing an application, user might want to use hierarchy in a way to design independent graphs that can be instantiated in any design. From a programmer view it behaves as closures since it defines limits for a portion of an application. This kind of hierarchy must ensure that while a graph is instantiated, its behavior might not be modified by its parent graph, and that its behavior might not introduce deadlock in its parent graph. The rules defined in the composition rules ensure the graph to be deadlock free when verified, but are used to analyze a graph with no hierarchy to create hierarchy levels in a bottom-up approach. Such graph specification is depicted in Figure 8.4. In order to allow the user to hierarchically design a graph in a top-down approach, this hierarchy semantic must ensure that the composed graph will have no deadlock if every level of hierarchy is independently deadlock free. To ensure this rule we propose to integrate special nodes in the model that restrict the hierarchy semantic. In the following a hierarchical vertex will refer to a vertex which embeds a hierarchy level, and a sub-graph will refer to the graph representing this hierarchy level.

8.3.1 Special nodes

Source node: a Source node is a bounded source of tokens which represents the tokens available for an iteration of the sub-graph. This node behaves as an interface to the outside world. A source port is defined by the four following rules:

A-1 Source production homogeneity: a source node *Source* produces the same amount of tokens on all its outgoing connections $p(e) = n \quad \forall e \in \{Source(e) = Source\}$.

A-2 Interface Scope: the source node remains write-locked during an iteration of the sub-graph. This means that the interface can not be filled by the outside world during the sub-graph execution.

A-3 Interface boundedness: a source node cannot be repeated, thus any node consuming more tokens than made available by the node will consume the same token multiple times (ring buffer). $c(e)\%p(e) = 0 \quad \forall e \in \{source(e) = source\}$.

A-4 SDF consistency: all the tokens made available by a source node must be consumed during an iteration of the sub-graph.

Sink node: a sink node is a bounded sink of tokens that represents the tokens to be produced by an iteration of the graph. This node behaves as an interface to

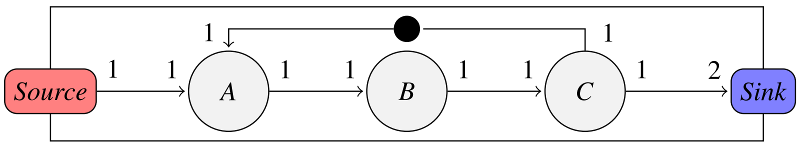


Figure 8.4: *Design of a sub-graph.*

the outside world. A sink node is defined by the four following rules:

B-1 Sink producer uniqueness: a sink node *Sink* has only one incoming connection.

B-2 Interface Scope: the sink node remains read-locked during an iteration of the sub-graph. This means that the interface cannot be read by the outside world during the sub-graph execution.

B-3 Interface boundedness: A sink node cannot be repeated, thus any node producing more tokens than needed by the node will write the same token multiple times (ring buffer). $p(e)\%c(e) = 0 \quad \forall e \in \{target(e) = Sink\}$.

B-4 SDF consistency: all the token consumed by a sink node must be produced during an iteration of the sub-graph.

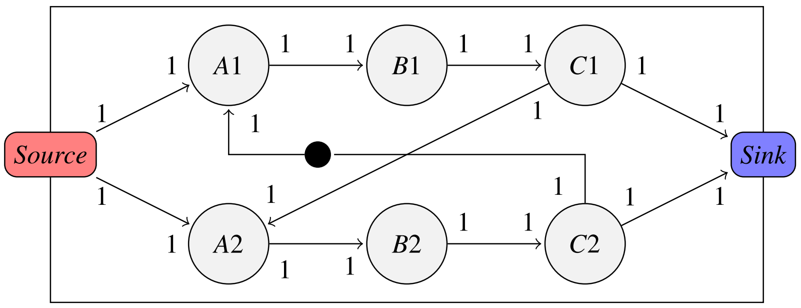


Figure 8.5: *A sub-graph after HSDF transformation.*

8.3.2 Hierarchy deadlock-freeness

Considering a consistent connected SDF graph $G = \{g, z\}$, $g = \{Source, x, y, Sink\}$ with *Source* being a source node and *Sink* being a sink node, and z being an actor. In the following we show how the hierarchy rules described above ensure the hierarchical vertex g to not introduce deadlocks in the graph G.

- If it exists a simple path going from x to y containing more than one arc, this path cannot introduce cycle since this path contains at least one interface, meaning that the cycle gets broken. User must take this into account to add delay to the top graph as this kind of path will create a cycle onto the actor g.
- Rules **A2-B2** ensure that all the data needed for an iteration of the sub-graph are available as soon as its execution starts, and that no external

vertex can consume on the sink interface while the sub-graph is being executed. As a consequence no external vertex strongly connected with the hierarchical vertex can be executed concurrently. The interface ensures the sub-graph content to be independent to the outside world, as there is no edge $\alpha \in \left\{ \alpha' \| \begin{pmatrix} (src(\alpha') = x) \\ and \\ (snk(\alpha') \in C) \\ and \\ (snk(\alpha') \notin \{x,y\}) \end{pmatrix} \right\}$ considering that $snk(\alpha') \notin \{x,y\}$) cannot happen.

– The designing approach of the hierarchy cannot lead to an hidden delay since even if a delay is in the sub-graph, an iteration of the sub-graph cannot start if its input interfaces are not full.

Those rules also guarantee that the edges of the sub-graph have a local scope, since the interfaces make the inner graph independent from the outside world. This means that when an edge in the sub-graph creates a cycle (and contains a delay), if the sub-graph needs to be repeated, this iterating edge will not link multiple instances of the sub-graph.

The given rules are sufficient to ensure a sub-graph to not create deadlocks when instantiated in a larger graph.

Transformations

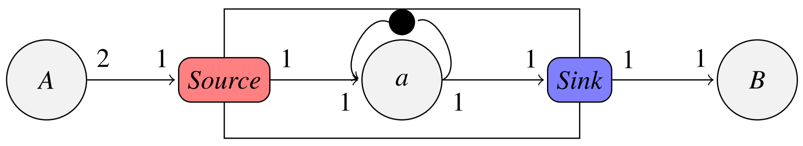


Figure 8.6: *Local edge in sub-graph.*

The main problem of this representation is its behavior during hierarchy flattening. Rules define earlier restrict the edges of a sub-graph to have a local scope. This means that when removing the hierarchy we must ensure that no edges will propagate data among multiple instances of the sub-graph. By removing the hierarchy with the basic SDF semantic, local delays of the sub-graph may have totally different meaning while performing HSDF transformation on the graph (Figure 8.7).

With our hierarchy semantic, before flattening a level of hierarchy, an HSDF transformation must be applied to preserve the edges scope of the sub-graphs.

The other concern is to preserve the interface behavior while removing the hierarchy. Those interfaces can either behave as fork for Source port, when tokens

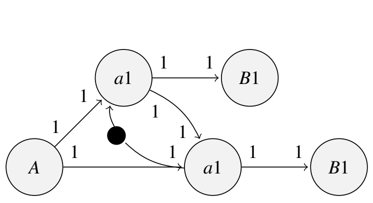
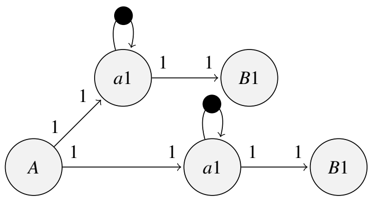


Figure 8.7: *Edge scope preservation in Interface based hierarchy.*



need to be broadcasted to multiple nodes, or join when multiple tokens must be grouped. In order to not introduce deadlocks, during the hierarchy flattening process, we must replace the interfaces by a special vertex in the flatten graph. Those special interfaces can then be treated to maximize the potential parallelism.

8.3.3 Hierarchy scheduling

As explained in [LM87a] interfaces to the outside world must not be taken into account to compute the schedule-ability of the graph. As in our hierarchy interpretation, interfaces have a meaning for the sub-graph, they must be taken into account to compute the schedule-ability, since we must ensure that all the tokens on the interfaces will be consumed/produced in an iteration of the sub-graph (see rules **A4-B4**).

Due to the interface nature, every connection coming/going from/to an interface must be considered like a connection to an independent interface. Adding an edge e to a graph G increases the rank of its topology matrix Γ if the row added to Γ is linearly independent from the other row. Adding an interface to a graph G composed of N vertices, and one edge e connecting this interface to G adds a linearly independent row to the topology matrix. This increases the rank of the topology matrix of one, but adding the interface's vertex will yield in a $N+1$ graph: $rank(\Gamma(G_N)) = N - 1 \Rightarrow rank(\Gamma(G_{N+1})) = rank(\Gamma(G_N)) + 1 = (N+1) - 1$. The rank of the topology matrix remains equal to the number of vertices less one meaning that this graph remains schedule-able. Since adding an edge between a connected interface and any vertex of the graph results in (in meaning) adding an edge between a newly created interface and the graph, it does not affect the schedule-ability considering the proof above. This means that a sub-graph can be considered schedule-able if its actor graph (excluding interfaces) is schedule-able.

Before scheduling a hierarchical graph, we must verify that every level of hierarchy is deadlock free. Applying the balance equation to every level is sufficient to prove the deadlock freeness of a level.

8.3.4 Hierarchy behavior

Interfaces behavior can vary due to the graph schedule. This behavior flexibility can ease the development process, but needs to be understood to avoid meaningless applications.

- Source behavior

As said in the *Source* interface rules, a *Source* interface can have multiple outgoing (independent) connection and reading more tokens than made available results in reading modulo the number of tokens available (circular buffer). This means that the interface can behave like a broadcast. In Figure 8.8, vertices A and B have to execute respectively 4 and 6 times to consume all the data made available by the port. In this example, the *Source* interface will broadcast twice to vertex A and three times to vertex B. This means that the designer must keep in mind that the interfaces can have effect on the inner graph schedule.

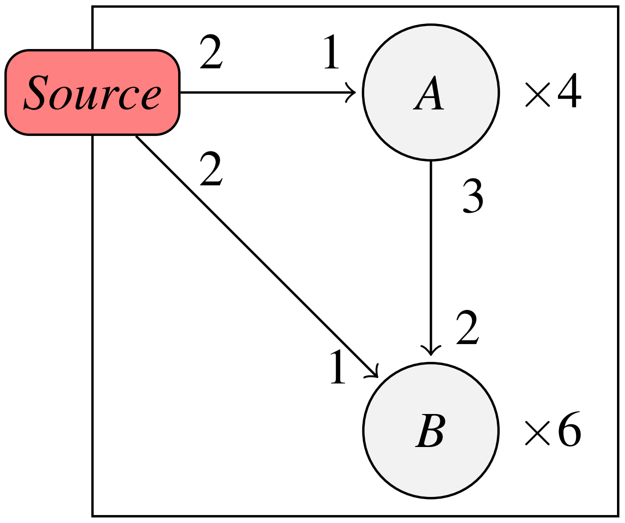


Figure 8.8: *Source example and its execution pattern.*

- Sink behavior

As said in the *Sink* interface rules, a *Sink* interface can only have one incoming connection, and writing more tokens than needed on the interface results in writing modulo the number of tokens needed (circular buffer). In Figure 8.9, the vertex B writes 3 tokens in a *Sink* that consumes only one token, due to the circular buffer behavior, only the last token written will be made available to the outside world. This behavior allows to easily design iterative pattern without increasing the number of edges. This behavior can also lead to mistakes (from the designer view) as if there is no precedence between multiple occurrences of a vertex that writes to an output port, a parallel execution of these occurrences leads to a concurrent access to the interface and as a consequence to indeterminate data in the *Sink* node. This also leads to dead codes from the node occurrences that do not write to the *Sink* node.

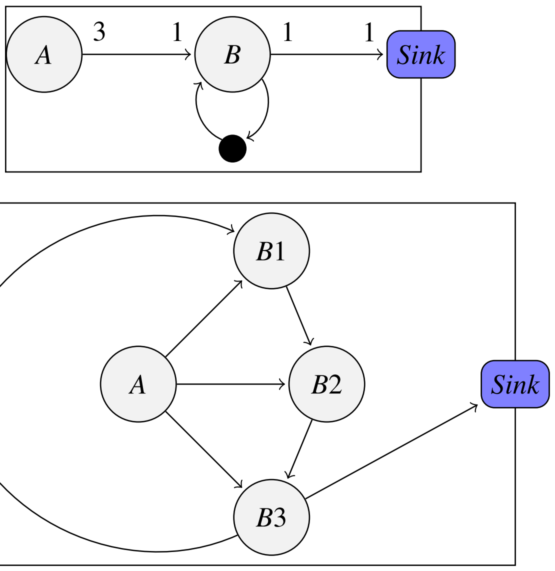


Figure 8.9: *Sink example and its precedence graph.*

8.3.5 Hierarchy improvements

As said earlier, this hierarchy type eases the designer work, since he/she can design subsystems independently and may instantiate them in any application. Not only easing the designer work, this kind of hierarchy also improves the application with the same criteria than the clustering techniques (scheduling complexity reduction, buffer requirements ...). Those improvements are based on the designer's choice but it can be completed by automatic transformation allowing more performance to be extracted from the graph.

8.4 Application case study

In this section we will show how the new hierarchy type (interface based hierarchy) can be used to design a IDCT2D_CLIP examples. The IDCT2D is a common application in image decoding which operates over a 8×8 matrix of coefficient to decode 8×8 pixel block. In the video decoding context the IDCT2D is followed by a clip operation which adds or not a sign bit on samples depending on the kind of prediction being used for the block (INTER or INTRA).

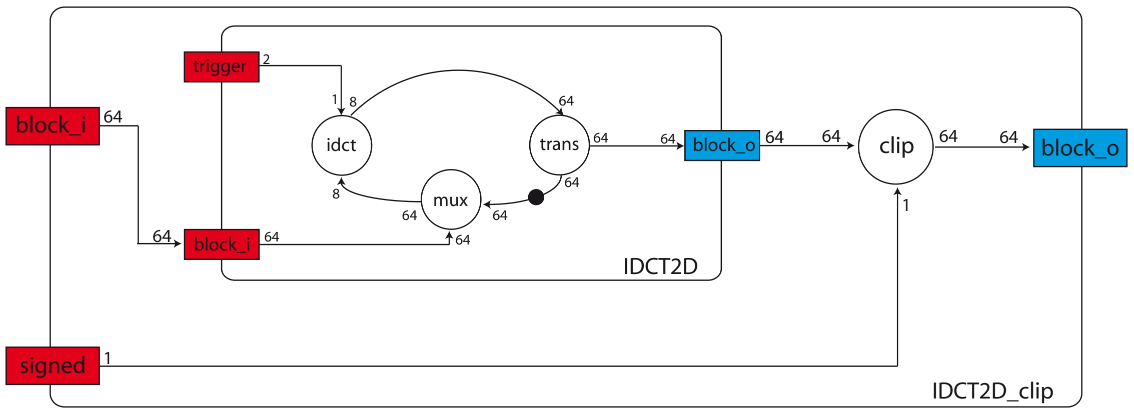


Figure 8.10: *IDCT2D_CLIP SDF graph*

8.4.1 IDCT2D description

The IDCT2D_CLIP used in this example (Figure 8.10) is a recursive pattern using only 2 operations.

- *mux*: The mux actor takes data from the $block_i$ port and the delay and output the data from $block_i$ ont its first invocation, and output the data from *trans* on other invocations.
- *idct*: Performs an IDCT on a vector of 8 elements.
- *trans*: Transposes an 8×8 matrix.
- *clip*: Apply an additional signed bit depending on the kind of the prediction type.

In this representation the *trigger* port is used to force the loop to iterate twice. A port that is connected on one of its side is taken into account for the repetition vector computation, but does not carry data at execution time. The IDCT2D_CLIP is defined using two levels of hierarchy. The first level performs a classic IDCT2D by using IDCT1D and transposition of an 8×8 matrix. The additional level adds the *clip* operation which is specific to the video decoding algorithm. This operation computes on each sample a signed 9-bit integer for INTER prediction, while it does an unsigned 8-bit integer for INTRA prediction.

8.4.2 Structural analysis

The IDCT2D graph takes into account some of the specific behavior of the new hierarchy type. This graph is described as an entity consuming 64 tokens on its input port and producing 64 tokens on its output port. The *trigger* port forces the recursive pattern composed of *mux, idct, trans* to iterate twice in order to perform the *idct* onto lines and columns of the 8×8 block that is present in the delay. The mux operation being computed twice, it consumes twice the same 64 tokens on the input port *block*$_i$ of the sub-network, making it behaves in a manner analogous to a dynamic block of parameter values that is held fixed during an execution of the subsystem. The *trans* operation being computed twice writes two times 64 tokens on the output port of the graph. In this case the output port behaves as a ring buffer since only the last 64 tokens written will be made available to the upper hierarchy level.

Describing the same application using repetition based SDF hierarchy would require the programmer to describe the application at fine grain and then use a clustering algorithm to construct the hierarchical representation. Moreover, the programmer would have to insert *broadcast* and *ring* actors explicitly into the graph to emulate the exact same behavior.

8.5 Conclusion

The Interface-based SDF fills a need of true hierarchy composition in a top down approach in the SDF model. Moreover, this hierarchy semantic provides the programmer with a more natural approach. Indeed a hierarchy level can be interpreted as a block of code in the C language semantic. Thus the programmer does not have to take care of a hierarchical actor internal behavior, as it cannot affect the upper hierarchy level. Furthermore, the specific behavior of the interface improves the SDF expressiveness, and allows to specify more algorithm in a SDF manner. The drawback of a factorized representation being the parallelism extraction, parallelism extraction algorithm must be provided to allow performance to be extracted from a hierarchical representation. Clustering does not have to deal with this problem, as it starts with a flat representation it tries to factorized. Thus its main problem is to minimize the parallelism starting from a parallel full prepresentation. In Chapter 10 we give hints and methods on how to extract performance from the interface-based representation.

Chapter 9

Multi-core code generation of Interface based Synchronous Dataflow

9.1 Introduction

As a rapid prototyping framework aims at providing the designer with methods/software that ease the application development process, the code generation is the output of the whole procedure. This step takes the result of the scheduling and must generate code executing the functionality specified by the user. Thus it must be able to reflect the behavior defined by the underlying model of computation, to interpret the tool specific attributes and to handle the complexity associated with the target multi-component system. The generated code must also be human readable to ensure proper comprehension from the designer and to allow later human driven optimizations.

The PREESM rapid prototyping framework aims at providing the designer with automated methods to go from the application description to an efficient multiprocessor C implementation. PREESM takes advantage of the Eclipse tool environment[1] that provides a clean free multi-platform graphical environment that can be extended through plugins. The PREESM plugin (Fig. 9.1) is a core of data flow oriented functionalities that make the use of PREESM plugins to transform, schedule graphs, and generate code. The transformations – kind of scheduling and code generation to perform on a graph – are configured using a PREESM workflow. A PREESM workflow is itself a graph of tasks to perform transformation on either a data flow graph or an architecture graph. It specifies the data dependencies be-

1. http://www.eclipse.org/

tween those tasks, and the configuration of these tasks. A workflow starts with three sources: the data flow graph, the architecture graph and the scenario file. The scenario file groups all target specific informations such as graph actor execution time, data-type size and can force actor allocation on the architecture. Then the workflow decribes a chain of actions, typically in the order: data flow graph transformation, allocation/scheduling, code generation. Those tasks are implemented in plugins that can provide more than one kind of algorithm. For example the graph transformation plugin provides HSDF transformation, hierarchy flattening, loop transformation and the sheduling plugin implements more than one scheduling heuristic. The allocation and scheduling plugins implement the work developped in [M.P10]. Those transformations can be extended by other plugins that will then be able to be instantiated in the workflow. Architecture graph, data flow graph and workflow graph are edited using the Eclipse Graphiti plugin [2] that is a generic configurable graph editor.

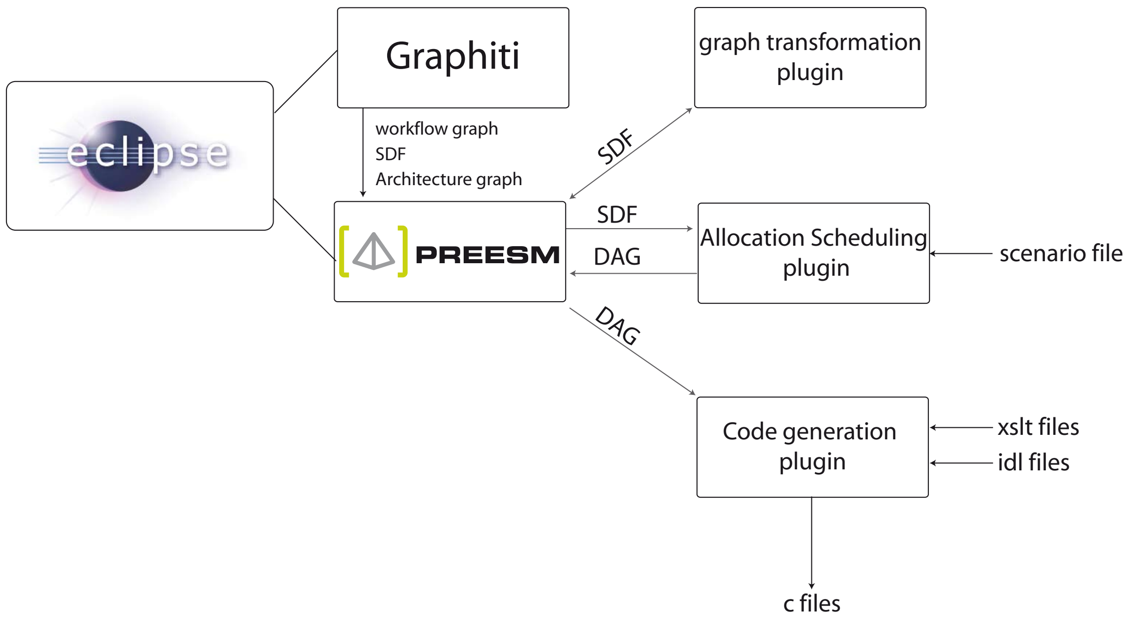
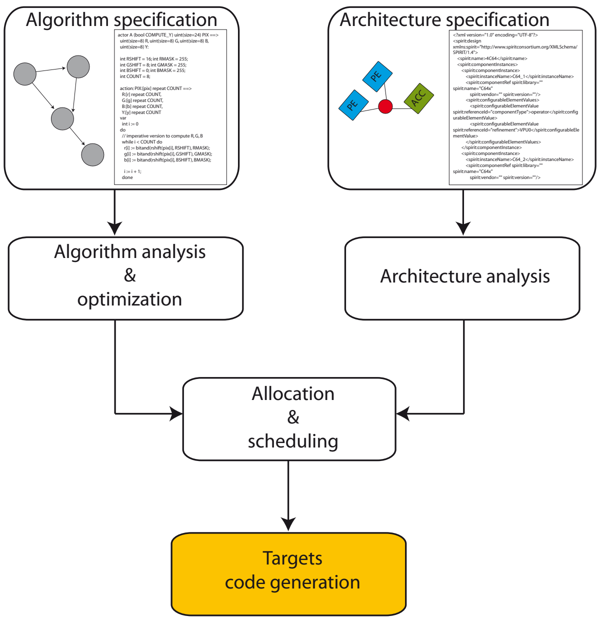


Figure 9.1: *PREESM framework structure.*

In the following I will describe how the code generation is performed in the PRESSM C code generation plugin (Fig. 9.2). In [PBPR09] we show how the Interface-based SDF is managed by the code generation in the case of multi-processor code-generation.

In the PREESM framework, the scheduler produces a Single Appearance Schedule (SAS).The top graph is thus an SDF graph in which vertices have additional properties providing information on the core the vertex is mapped to and the vertex's scheduling order. This graph also has additional vertices representing Inter

2. http://sourceforge.net/projects/graphiti-editor/



```
actor A (bool COMPUTE_Y) uint(size=24) PIX ==>
  uint(size=8) R, uint(size=8) G, uint(size=8) B,
  uint(size=8) Y:

  int RSHIFT = 16; int RMASK = 255;
  int GSHIFT = 8; int GMASK = 255;
  int BSHIFT = 0; int BMASK = 255;
  int COUNT = 8;

  action: PIX:[pix] repeat COUNT ==>
   R:[r] repeat COUNT,
   G:[g] repeat COUNT,
   B:[b] repeat COUNT,
   Y:[y] repeat COUNT
  var
   int i := 0
  do
   // imperative version to compute R, G, B
   while i < COUNT do
    r[i] := bitand(rshift(pix[i], RSHIFT), RMASK);
    g[i] := bitand(rshift(pix[i], GSHIFT), GMASK);
    b[i] := bitand(rshift(pix[i], BSHIFT), BMASK);

    i := i + 1;
   done
```

```
<?xml version="1.0" encoding="UTF-8"?>
<spirit:design
xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/
SPIRIT/1.4">
   <spirit:name>4C64</spirit:name>
   <spirit:componentInstances>
      <spirit:componentInstance>
         <spirit:instanceName>C64_1</spirit:instanceName>
         <spirit:componentRef spirit:library=""
spirit:name="C64x"
            spirit:vendor="" spirit:version=""/>
         <spirit:configurableElementValues>
            <spirit:configurableElementValue
spirit:referenceId="componentType">operator</spirit:config
urableElementValue>
            <spirit:configurableElementValue
spirit:referenceId="refinement">VPU0</spirit:configurableEle
mentValue>
         </spirit:configurableElementValues>
      </spirit:componentInstance>
      <spirit:componentInstance>
         <spirit:instanceName>C64_2</spirit:instanceName>
         <spirit:componentRef spirit:library=""
spirit:name="C64x"
            spirit:vendor="" spirit:version=""/>
```

Figure 9.2: *Code generation phase in the rapid prototyping workflow.*

Processor Communications (IPCs) as a pair of a send vertex and a receive vertex linked by a precedence arc and mapped onto the network. The goal of the code generation is to synthesis code that executes the behavior of the given SDF graphs using the resources of the architecture in respect to the information provided by the scheduler.

9.2 Multi-threaded execution model

The code generation relies on an execution model in which the parallel computation is synchronized by the data. The generated program consists of multiple threads for each target processor. A first thread handles the computation while the other threads handle the inter processor communication on each used medium. Such execution model allows parallelism between computation and communication and can take advantage of hardware accelerator (DMA). The threads are synchronized using mutual exclusion semaphores as depicted in Figure 9.3. This approach also allows to have bounded memories on heterogeneous architectures as communication can block the computation when waiting for useful data. This also means that the scheduling step must be aware of the heterogeneity of the operators and the performance of the communication media.

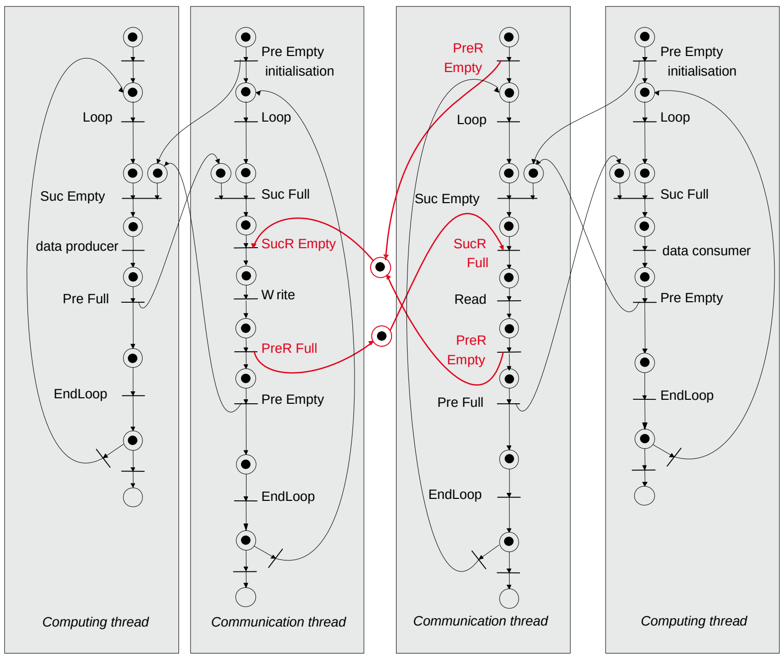


Figure 9.3: *Petri network of the thread synchronizations*

Such model requires an underlying operating system that handles multi-threading and mutex primitives. Purely sequential code with embedded communication primitive code could be generated with a significant loss in performance, as communications would become blocking.

Each thread is defined as a sequence of three states. At first the init state performs dynamic allocations, like initializations of semaphores and resources. The loop state is a infinite loop in which the SDF specification or the communication flow is executed. The end state frees the resources allocated during the init state in case the loop state ends.

9.3 C code generation procedure

The C code generation aims at producing code that execute the SDF graph behavior in a human readable format compilable for a given architecture. An SDF graph only describes the data dependencies and actor behavior regarding data consumption/production. In the PREESM framework the graph is thus annotated with information like data-types and corresponding function calls. The generated C code must then be completed with files containing actor function calls implementations.

Moreover the code generation is performed using a scheduled SDF graph plus a set of Interface Description Language files (IDL) that gives for each actor the associated function prototype, and a scenario file that groups all architecture dependent informations related to the application and the target architecture. Those informations are mainly data-type sizes, actor execution time and data-rates on inter processor communications. In addition, each edge of the network is annotated with a data-type information. Those informations are used to generate a generic code in an xml syntax that can then be transformed, using xslt style-sheet, to C code. This structure allows to use different xslt style-sheet to target different processors and/or different Operating Systems by generating target dependent synchronization and communication primitives. One file is generated per processor/core, and all the allocations are performed in respect to the processor the element is allocated to. Figure 9.4 shows the steps of the code generation.

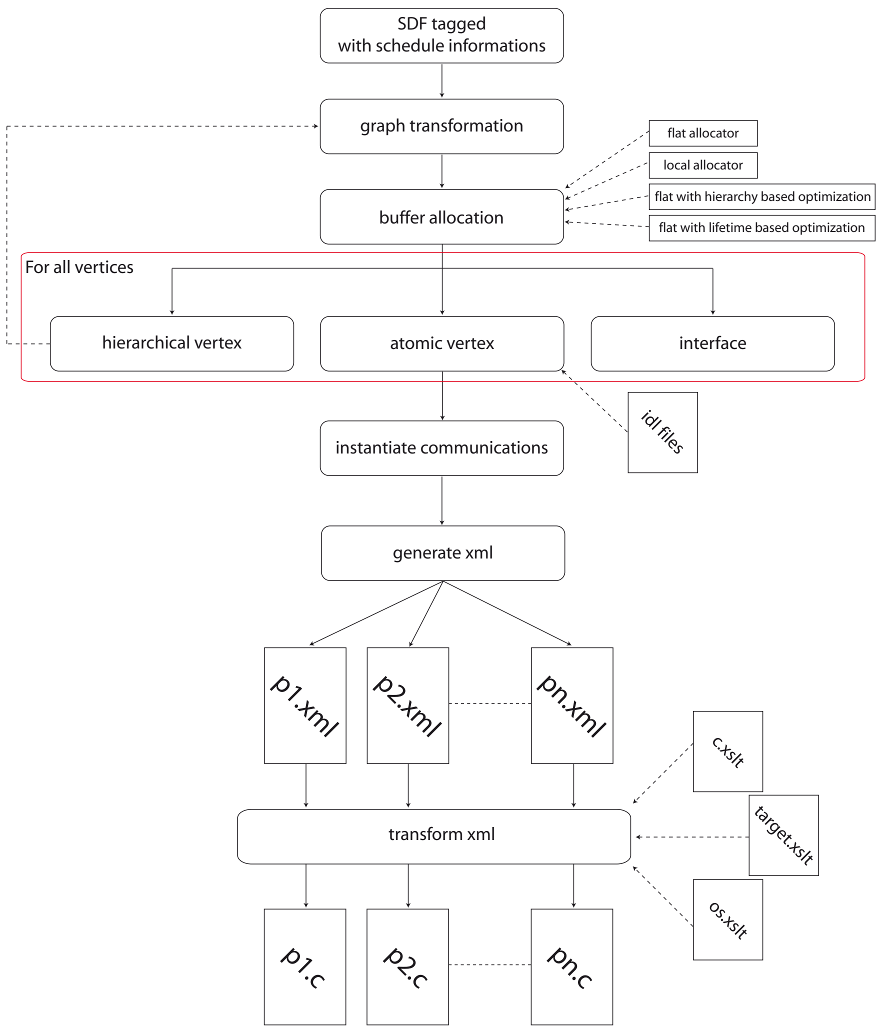


Figure 9.4: *Code generation steps.*

Thus the code generation is performed through several steps (see Figure 9.5). A first step transforms the graph for better code generation. A second step allocates all the buffer involved in the computation. Then all the actors are synthesized into function calls using the information provided in the IDL files. In a fourth step the communication thread is created. Finally the xml generic code is generated and transformed into a c file for each processor.

```
public void generateCode(Graph g, Scenario s){
        transformGraph(g);
        for(Edge e : g.getAllEdges()){
                allocateBuffer(e)
        }
        for(Vertex v : g.getAllVertex()){
                if(v.isSendVertex()){
                        allocateIPCSend(v, v.getIncomingEdge());
                }else if(v.isReceiveVertex()){
                        allocateIPCReceive(v, v.getOutgoingEdge());
                }if(v.isInputPort()){
                        getSubBufferFromInterface(v);
                }else if(v.isOutputPort()){
                        getSubBufferFromInterface(v);
                }else if(v.isHierarchicalNode()){
                        codeGeneration(v.getSubGraph())
                }else{
                        instantiatePrototype(v, s);
                }
        }
        if(!g.getParentGraph() == null){
                createCommunicationThread(g, s);
        }
}
```

Figure 9.5: *Code generation procedure.*

9.4 Graph optimization for code generation

The tagged SDF graph that is provided by the scheduler needs some modification for efficient code generation (Fig. 9.6).

```
public void transformGraph(Graph g){
        for(Set<Vertex> stronglyConnected : getStronglyConnectedComponents()){
                if(greatestCommonDividerOfRepetionFactor(stronglyConnected) > 1){
                        clusterize(stronglyConnected);
                }
        }
        for(Vertex v : g.getAllVertex()){
                if(v.isInputPort()){
                        treatInputInterface(v);
                }else if(v.isOutputPort()){
                        treatOutputInterface(v);
                }
        }
}
```

Figure 9.6: *Graph Transformation procedure.*

The first optimization concerns the strongly connected components or cycles. In order to generate human readable code, a cycle that is repeated over time must be embedded into a loop. To do so, we detect strongly connected components with more than one vertex in the graph, and clusterize them. The edges of the cycle that carry a delay are wrapped around the created cluster (see Figure 9.7). From strongly connected components of size greater than one in the graph, we create a hierarchical actor that embeds all its elements. All the arcs that connect actors not belonging to the strongly connected set are connected through the created cluster interface. The edges of the strongly connected set that carries a delay are connected around the cluster.

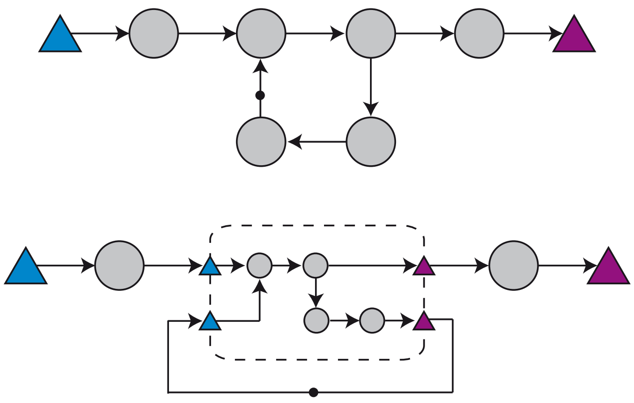


Figure 9.7: *A graph containing a cycle and its resulting transformation.*

The second important transformation aims at ensuring a correct interface behavior in the hierarchy (Fig. 9.10). As described in [PBR09], an input interface has an intrinsic behavior of broadcast. Thus if multiple edges are connected to one input interface, we must ensure for each edge that the broadcast behavior of the interface will be respected. The broadcast adding procedure stands as in Figure 9.8:

```
public void treatInputInterface(InputVertex v, Graph g){
        for(Edge e : g.getOutgoingEdgesOf(v)){
                if(e.getCons()\%e.getProd() != 0){
                        addBroadcastVertex(e, lcm(e.getProd(), e.getCons()) ;
                }
        }
}
```

Figure 9.8: *Procedure for adding broadcast on input interfaces.*

As described in [PBR09], an output interface has an intrinsic behavior of ring buffer. Thus we must ensure that the edge connected to the input interface can support this specific behavior. The ring buffer adding procedure stands as follows (Fig. 9.9):

```
public void treatOutputInterface(OutputVertex v, Graph g){
        for(Edge e : g.getIncomingEdgesOf(v)){
                if(e.getProd()\%e.getCons() != 0){
                        addRingBufferVertex(e, lcm(e.getProd(), e.getCons()) ;
                }
        }
}
```

Figure 9.9: *Procedure for adding ring buffer on output interfaces.*

In Figure 9.10, one can see that the actor connected to the input port has a repetition factor of three and a compsumption rate of four. Thus it consumes twelve tokens from a port that only produces three tokens. For proper operation we must insert a *broadcast* vertex that will output four times the three tokens produced by the port. In Figure 9.10 we can also see that the actor connected to the output port has a repetition factor of four and produces three tokens. It means that it produces twelve tokens in a port that requires three tokens to be produced. By adding a *ring* vertex that consumes twelve tokens and outputs only the last three we ensure proper operation.

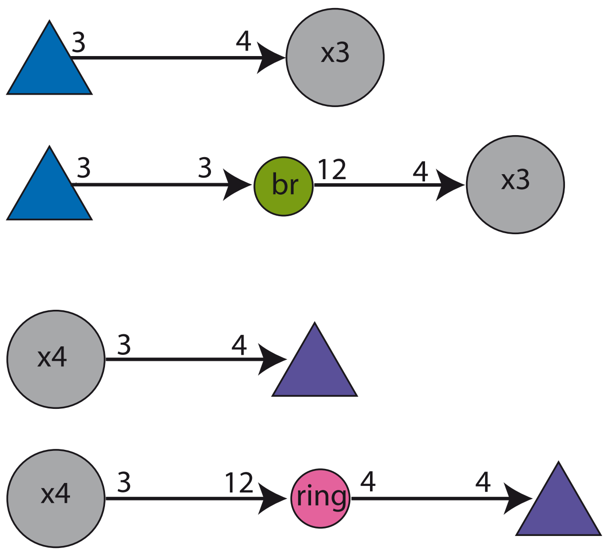


Figure 9.10: *Broadcast and ring buffer adding rules.*

9.5 Buffer optimization and allocation

The buffer allocation step aims at allocating the memory resources used by the application. In the SDF graphs, data dependencies between actors are symbolized using arcs with an integer production on the producer side and an integer consumption on the consumer side. Thus each edge represents a memory buffer in which the producer writes, and the consumer reads. The size of such buffer can easily be computed by $p(e) * q(src(e)) * sizeof(datatype)$. Allocating the buffer of the application can be performed in different ways, with different levels of optimization. In order to have flexibility on the buffer allocation, the code generation delegates the buffer allocation task to a given buffer allocator defined in the scenario that just returns the allocated buffer. Thus changing the buffer allocation strategy is just a question of defining a new buffer allocator.

```
public void allocateBuffer(Edge e, Scenario s){
        getBufferAllocator().allocateBuffer(e);
}
```

Figure 9.11: *Buffer allocation procedure.*

- **Flat buffer allocation**:
  This allocation technique (Fig.) is based on the *non-shared memory model*. Each buffer is allocated individually and lives throughout the schedule. In the flat buffer allocation, all the buffers of the top network, and all the buffers of the sub-networks (hierarchy) are globally allocated. Thus each edge can be compiled as fixed size array that has only one producer and one consumer, which data life-time is a full iteration of the schedule (see Figure 9.12). The static scheduling step ensures that there will not be any buffer overflow. This means that memory requirement of the application corresponds to the sum of the sizes of the buffer of the network. The buffer size for an edge of the network is equal to $p(e) * q(src(e)) * sizeof(datatype)$. Thus the total memory requirement of the network is equal to $\sum_{i=0}^{n} p(e_i) * q(src(e_i)) * sizeof(datatype(e_i))$ with n being the number of edges in the network. This memory allocation takes advantage of any memory optimization done at the scheduling step. For such allocation strategy, the code generation is made pretty simple. Each edge of the graph is allocated as a global array of the given data-type which size equals $p(e) * q(src(e))$. This allocation strategy provides an easy to read memory allocation as each edge can be associated to its corresponding buffer with ease. The main drawback of this allocation step, is that it does not take advantage of buffer life-time to optimize application memory requirement.
- **Hierarchical buffer allocation**:
  The Interface base hierarchy as described in [PBR09] allows further optimization based on its properties. This kind of hierarchy sets a sub-network to behave like a block of code. This means that all the arcs of the sub-network can be allocated at the beginning of the execution of the sub-network and freed at its end. The hierarchical buffer allocation takes advantage of this specific

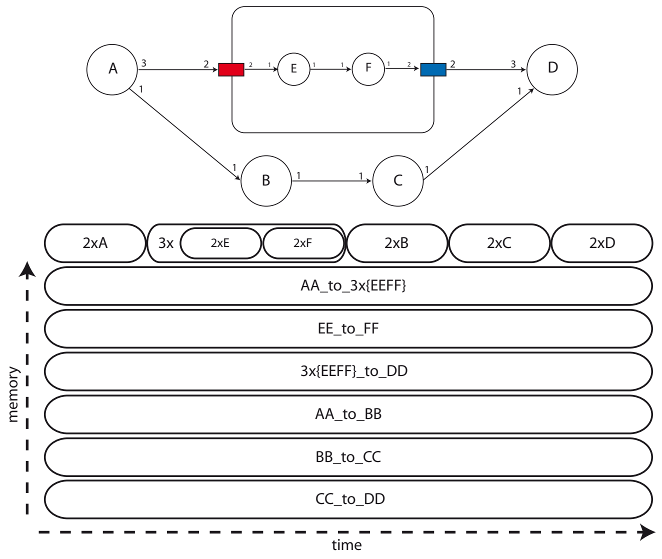


Figure 9.12: *A graph and its buffer life-time using a flat allocation.*

behavior to limit buffer life-time (see Figure 9.13). As the generated code takes advantage of sub-network by describing as block of code, buffers are allocated at the beginning of the block of code. Only buffers of the top graph remain as global variables as they are shared between the computation and communication threads. The c compiler can then take advantage of this structured code to optimize the memory footprint of the code. The main drawback is that the sub-network buffers are allocated in the stack at run-time. It means that the stack must be dimensioned accordingly.

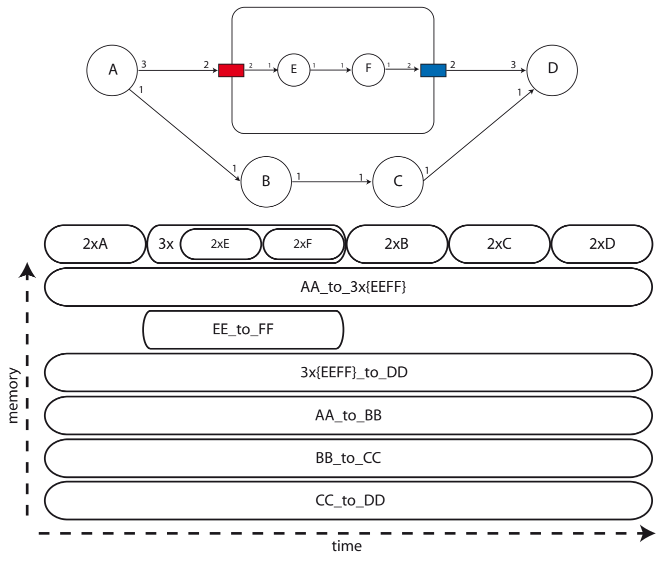


Figure 9.13: *A graph and its buffer life-time using a hierarchical allocation.*

- **Flat buffer allocation with hierarchy based optimization**: In the previous buffer allocation strategy, it appears that the memory allocated for a sub-network, has its life-time limited regarding to the schedule length. Thus it is possible to take advantage of this information to globally allocate the buffers embedded in sub-networks at the top level, with multiple sub-network sharing the same memory space. This is performed by allocating all the edges of the top graph at disjoint index ranges in a global array. Then when a hierarchical vertex starts its buffers are allocated at disjoint index range in the same global array (see Figure 9.14). All the memory allocated for the hierarchical vertex is considered freed when its end and the next hierarchical level can use the same index space to allocate its own buffers.
  In order to have a more fragmented allocation, it is also possible to define the maximum size of this global array, to then allocate in another memory space when the allocation overflows the buffer size.

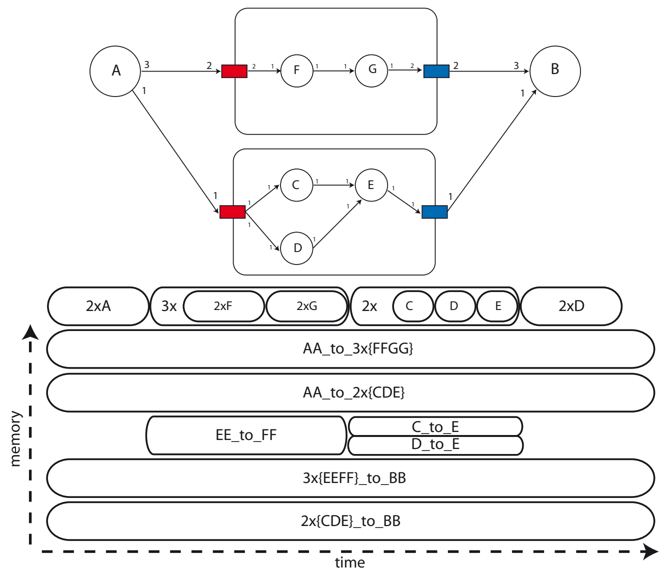


Figure 9.14: *A graph and its buffer life-time using a flat buffer allocation with hierarchical optimization.*

- **Flat buffer allocation with buffer life-time based optimization**:
  As to optimize memory footprint of an application it is possible to allocate the buffer based on a *shared memory model*. It is based on the fact multiple buffers can be allocated in the same memory space as long as their life-time does not overlap (see Figure 9.16). A buffer life-time starts when its producer actor is triggered and ends when its consumer actor ends (Fig. 9.15).
  Different techniques such as graph coloring [GR96] or in the case of data and code optimization [MBL97] exist. The last technique shows good improvements and is coupled with a graph transformation that aims at factorizing

```
public computeLifetime(Edge e){
        e.setStartTime(e.getSource().getStartTime());
        e.setStopTime(e.getTarget().getStopTime().e.getTarget().getDuration());
}
```

Figure 9.15: *Procedure to compute a buffer life-time.*

the code by clustering vertices which can be embedded into a loop. In addition it provides a technique for memory allocation of the different buffers based on their respective life-time. This problem is called *Dynamic Storage Allocation* and is proved [GJ79] to be an NP complete problem. The *First Fit* algorithm [Kie91] tackles this problem by assigning the smallest feasible location to each interval in the order they appear in the enumerated instance. In [MB01] the authors say that ordering the buffers based on their life-times give the best result, compared to a start-time based organization. Another technique described in [Rau06] was especially developed for buffer optimization in the context of the rapid prototyping tool SynDEx. In the case of multi-processor with Inter Processor Communications, one must also take into account that the buffer life-time for buffer involved in a communication cannot be computed at compile time as the computation thread and the commnication threads are asynchronous. Thus any buffer involved in an IPC must not be allocated in the shared memory space.

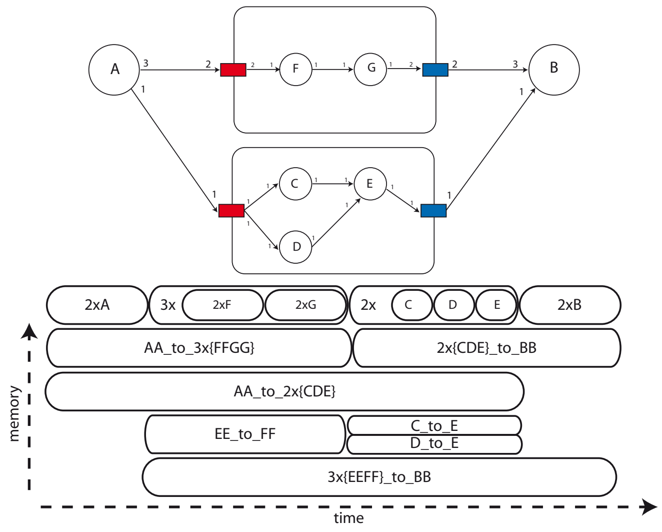


Figure 9.16: *A graph and its buffer life-time using a flat buffer allocation with life-time based optimization.*

Above all the allocation techniques described, the buffer allocation with life-time based optimization gives the best results in term of memory footprint of the application. The flat allocation gives the worst results, but can be useful for debugging purpose. The hierarchical allocation can show good result, but in the case of large graph, the stack must be large enough to allocate all the buffer of a hierarchical level. The flat allocation with hierarchical optimization provides a solution to the stack dimensioning problem and is good solution for debugging purpose compared to the allocation with life-time optimization as buffers of the top graph do not reside in a shared memory space. The choice of the buffer allocation strategy is guided by the user in the PREESM software to satisfy his/her needs at a given development step.

9.6 Hierarchical actor port management

A hierarchical actor is connected to its parent network through ports. Thus data tokens produced by the input port vertices are related to data tokens consumed on the corresponding actor interface in its parent network. Moreover data tokens consumed by an output port are equivalent of the data tokens produced on the corresponding actor interface in its parent network. Thus there is no need to allocate the buffers corresponding to the arcs connected to an interface as they will carry the same data tokens. All edges connected to an input port are to be synthesized as pointers into the buffer allocated for the equivalent edge in the upper hierarchy level (see Figure 9.17). For the single edge connected to an output port (*Sink* producer uniqueness rule), its equivalent buffer is allocated as a pointer into the buffer allocated for the arc connected to this output port in the upper hierarchy level.

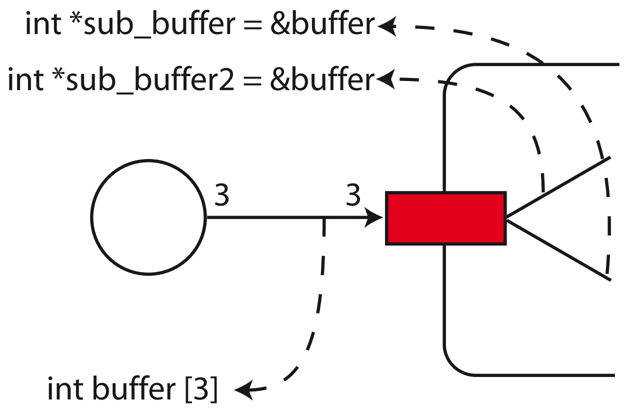


Figure 9.17: *An input port and its equivalent buffer allocation.*

9.7 Actor prototypes instantiation

The actor prototype instantiation aims at instantiating the prototype of each actor and passing it the right buffers as arguments. Our code generation takes IDL (Interface Description Language) for each actor as the prototype definition (Fig. 9.18). The IDL file provides the description of three interfaces. The first interface is the prototype to be instantiated into the init phase, the second prototype is to be instantiated in the loop phase, and the third prototype in the end phase. The type *parameter* is used as a keyword to identify SDF specification parameters, that can be resolved at compile-time. The init and end prototypes do not expose any argument, but parameters as they do not belong to the dataflow specification. The loop interface's prototype exposes arguments corresponding to the dataflow actor ports and static parameters. The IDL keywords *in* and *out* provide the direction of the port, and the data-type gives the type of the data tokens streaming in/out that port.

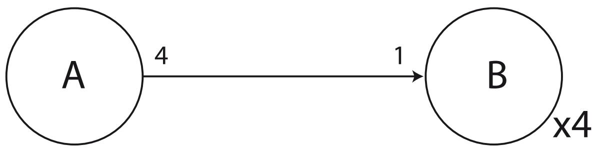
```
module mux {
        typedef long parameter;
        interface init{
                void init_mux();
        };
        interface loop{
                void mux( in char initIn , in char iterateIn ,
                in char trig ,out char data1Out ,out char data2Out ,
                out char weights ,in parameter size );
        };
        interface end{
                void end_mux();
        };
};
```

Figure 9.18: *IDL definition of an actor*

The code generation is thus in charge of instantiating the given prototype, with the arguments corresponding to the buffer allocated for the edges connected to the input and output ports.

9.8 In loop actors instantiation

In an SAS schedule, some actors A are associated to a repetition factor $q(A)$ greater than one. Those specific actors can either be flattened, meaning that they will be instantiated $q(A)$ times, or be unique and be repeated $q(A)$ times. For the last statement, it has been chosen to embed those actors into *for* loop, with an iteration vector of size $q(A)$. As a consequence, the inputs and outputs of this actor depend on the iteration step, and are subsets of the buffer corresponding to the connected edges (Fig. 9.19). Thus the sub-buffer sb for the iteration i of an actor B that consumes n token at each iteration from a buffer e of total size s, is allocated as $sb = e[(i \times n)\%s]$.



```
int buffer [4] ;
int i ;
A(buffer);
for(i = 0 ; i < 4 ; i++){
      int * sub_buffer = buffer[(i*1)%4] ;
      B(sub_buffer);
}
```

Figure 9.19: *A two actor SDF graph and its corresponding code generation.*

9.9 Special vertices instantiation

As mentioned earlier, some vertices with special behavior can be added to the graph. Those vertices are broadcast, fork, join and ring buffer.

Depending on the case those vertices can either be synthesized as memory copies or pointer into input/output buffer.

- The Fork vertex: the fork vertex cuts a finite sequence of input tokens into multiple sub-sets of this input buffer $Fork = \{N\} \Rightarrow N \times \{1\}$. Thus, the fork vertex can be synthesized as pointers into the input buffer. If this fork vertex belongs to the top graph, it is synthesized as memory copies of the input buffer into the output buffers if one the output edge is connect to an IPC vertex (Fig. 9.20).
- The Join vertex: the join vertex takes multiple input tokens and groups them on its single output $Join = N \times \{1\} \Rightarrow \{N\}$. The join vertex can either be synthesized as a memory copy, or as pointers into the output buffer. If it belongs to the top graph it is synthesized as memory copies of the input buffers into the output buffer if its outgoing edge is connected to an IPC vertex (Fig. 9.21).

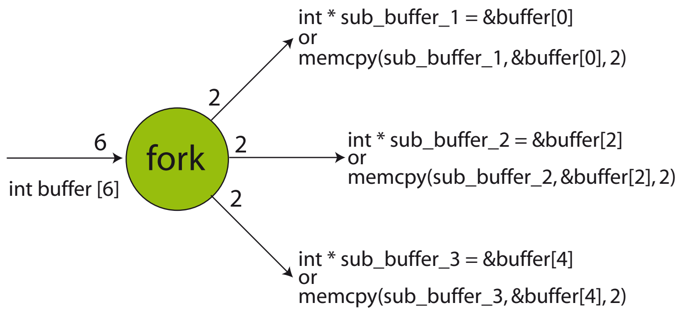

int * sub_buffer_1 = &buffer[0]
or
memcpy(sub_buffer_1, &buffer[0], 2)
2
6
fork
2
int buffer [6]
2
int * sub_buffer_2 = &buffer[2]
or
memcpy(sub_buffer_2, &buffer[2], 2)
int * sub_buffer_3 = &buffer[4]
or
memcpy(sub_buffer_3, &buffer[4], 2)

Figure 9.20: *A fork vertex and its two possible code generations.*

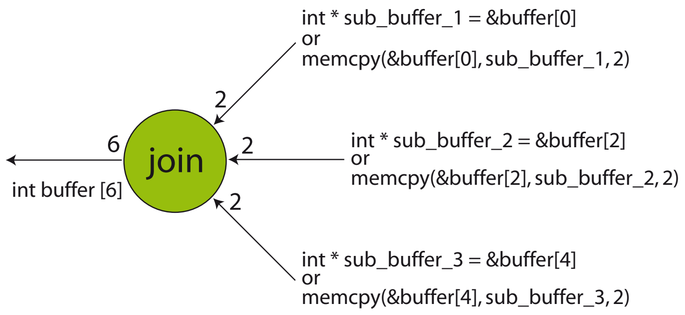

int * sub_buffer_1 = &buffer[0]
or
memcpy(&buffer[0], sub_buffer_1, 2)
2
6
join
2
int buffer [6]
2
int * sub_buffer_2 = &buffer[2]
or
memcpy(&buffer[2], sub_buffer_2, 2)
int * sub_buffer_3 = &buffer[4]
or
memcpy(&buffer[4], sub_buffer_3, 2)

Figure 9.21: *A join vertex and its two possible code generations*

– The Broadcast vertex: the broadcast vertex copies its input tokens as many times as needed on its output edges $Broadcast = \{N\} \Rightarrow b \times \{N\}$ (Fig. 9.22).

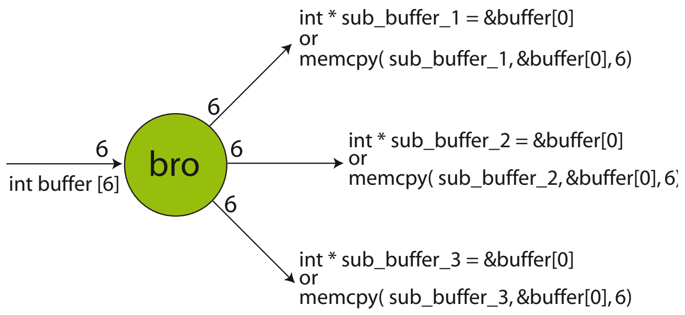

int * sub_buffer_1 = &buffer[0]
or
memcpy(sub_buffer_1, &buffer[0], 6)
6
6
bro
6
int buffer [6]
6
int * sub_buffer_2 = &buffer[0]
or
memcpy(sub_buffer_2, &buffer[0], 6)
int * sub_buffer_3 = &buffer[0]
or
memcpy(sub_buffer_3, &buffer[0], 6)

Figure 9.22: *A broadcast vertex and its two possible code generations*

– The Ring vertex: the ring vertex copies the last N tokens consumed into its output $Ring = r \times \{N\} \Rightarrow \{N\}$ (Fig. 9.23).

9.10 Inter Processor Communication instantiation

The graph computed by the scheduling steps contains *send* and *receive* vertices that are allocated onto processors. The *send* vertex is allocated onto the data producer processor, and the receive vertex is allocated onto the receiving processor. This pattern is handheld both in the computation thread and in the communication one.

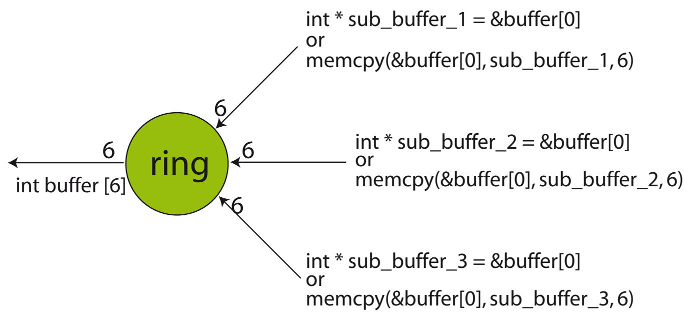

int * sub_buffer_1 = &buffer[0]
or
memcpy(&buffer[0], sub_buffer_1, 6)
6
6
ring
6
int buffer [6]
int * sub_buffer_2 = &buffer[0]
or
memcpy(&buffer[0], sub_buffer_2, 6)
6
int * sub_buffer_3 = &buffer[0]
or
memcpy(&buffer[0], sub_buffer_3, 6)

Figure 9.23: *A ring vertex and its two possible code generations*

As the buffer containing the data to be transferred is shared between the computation and communication threads, it has to be protected using concurrent processing data protection primitives. This primitives being OS dependent, their prototype instantiation is done by the xslt transformation. Two concurrent access cases can occur for both send and receive. For a data to be transferred, we must ensure that the buffer in which the producer vertex writes is not currently being transferred, and that the communication thread will not try to send a buffer in which no data has been written. For a data to be received we must ensure that the consuming vertex does not read into a buffer in which no data has been received, and that the communication thread will not receive data into a buffer that is currently accessed by the consumer vertex. Thus we need two buffers to ensure data coherence. The data protection is managed using mutual exclusion semaphores. The first semaphore is managed by the communication threads and indicates "data transferred" (either sent or received), and its state is initialized to 0 (lock) in the receive case and 1 (unlock) in the send case. The second semaphore is managed by the computation thread and indicates "data produced/consumed" and its state is initialized to 0 (lock) in the send case and 1 (unlock) in the receive case. A communication sequence operates as follows on the computation side:

1. A semaphore wait is reached into the computation thread and blocks until the semaphore that indicates " data transferred" is freed.
2. The data producing/consuming function is triggered.
3. The semaphore that indicates " data produced/consumed" is freed.
4. The code keep running until the same sequence is activated.

And as follow on the communication side:

1. A semaphore wait is reached into the communication thread and blocks until the semaphore that indicates " data produced/consumed" is freed
2. The transfer function is called and blocks until the transfer completes.
3. The semaphore that indicates " data transferred" is freed.

4. The communication thread keeps running until the same sequence is activated.

In the code generation procedure when a send vertex is encountered, a new Semaphore s_1 is allocated. Then a $sem_{wait}(sem_1)$ call is instantiated before the connected vertex in the computation thread. A second semaphore s_2 is then allocated and a $sem_{post}(s_2)$ is instantiated after the producing vertex. In the communication thread the sequence $sem_{wait}(s_2), send(data), sem_{post}(s_1)$ is instantiated (see Figure 9.24). The same procedure is executed for the receive vertices (see Figure 9.25), the main difference is that the semaphore sem_2 is initialized at 0 (V) and the sem_1 at 1 (P) whereas for a send sequence the semaphores are initialized at 1 for sem_1 and 0 for sem_2. Figure 9.26 shows a network with an IPC, and the resulting schedule.

```
public allocateIPCSend(SendVertex v, Edge e){
        Semaphore sem1 = allocateSemaphore();
        Semaphore sem2 = allocateSemaphore();
        computationThread.addInitSemaphore(sem2, 0);
        computationThread.addSemWaitAt(sem1, e.getSource()-1);
        computationThread.addSemPostAt(sem2, e.getSource()+1);
        communicationThread.addSemWait(sem2);
        communicationThread.addSendFunction(e);
        communicationThread.addSemPost(sem1);
        communicationThread.addInitSemaphore(sem1, 1);
        }
```

Figure 9.24: *Procedure to allocate an IPC send.*

```
public allocateIPCReceive(ReceiveVertex v, Edge e){
        Semaphore sem1 = allocateSemaphore();
        Semaphore sem2 = allocateSemaphore();
        computationThread.addInitSemaphore(sem2, 1);
        computationThread.addSemWaitAt(sem1, e.getTarget()-1);
        computationThread.addSemPostAt(sem2, e.getTarget()+1);
        communicationThread.addSemWait(sem2);
        communicationThread.addReceiveFunction(e);
        communicationThread.addSemPost(sem1);
        communicationThread.addInitSemaphore(sem1, 0);
}
```

Figure 9.25: *Procedure to allocate an IPC receive.*

9.11 xml to C transformation

The steps described above generate an xml file for each processor. This xml file contains generic code that can be transformed to C code using an xslt style-sheet. Xslt that stands for Extensible Stylesheet Language Transformations, is a W3C standard that aims at providing a syntax to define transformations for an XML

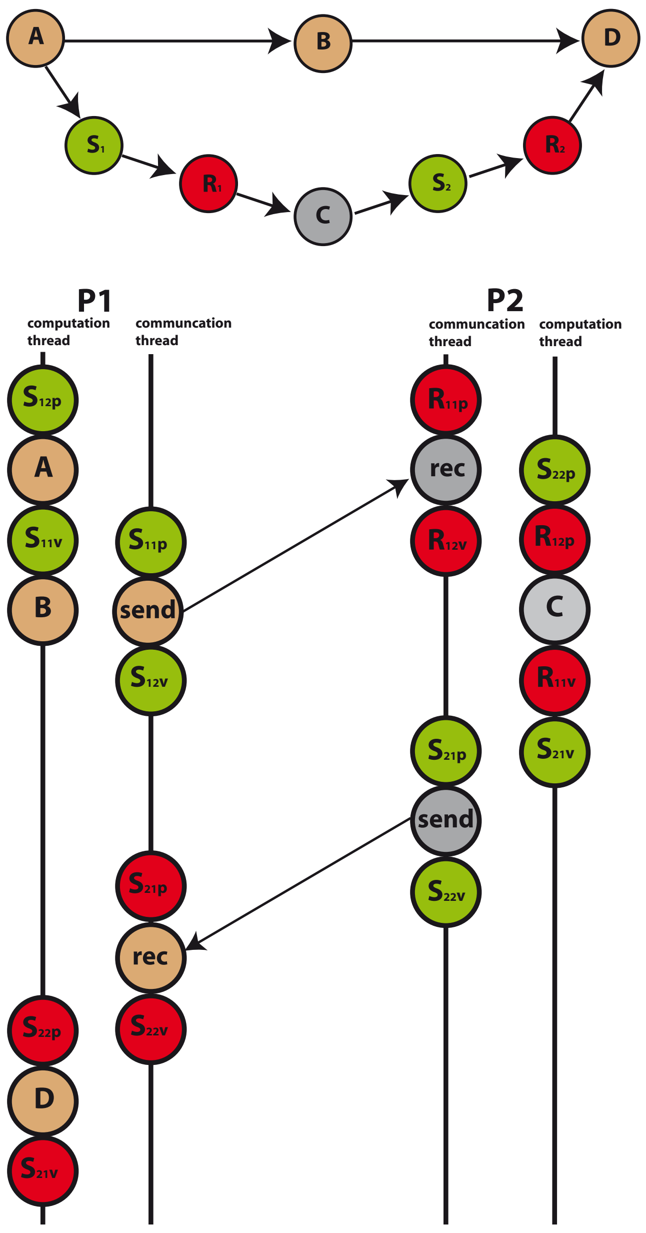


Figure 9.26: *A scheduled graph and the resulting execution.*

dialect to another or into any text based dialect. In PREESM this transformation is managed through three classes of xslt style-sheets.

- C code style-sheet: defines the xml to C translation for ansi C components (types, calls ...)
- Target style-sheet: defines the transformation for target specific calls (transfer)
- OS style-sheet: defines the transformation for OS specific calls (semaphores, threads)

Thus for a new hardware architecture, the user has to define (or use) a style-sheet that specifies the transformations for target specific calls. The user can also have to define a new OS style-sheet if required. The C style-sheet remain the same for all C code generations. The targeted architecture could also be extended by defining VHDL transformations style-sheets to generated code for programmable hardware, like FPGA.

This code production flow was inspired by the SynDEx tool that produces m4 code that can then be transformed using transformations defined in m4x files. As m4 is an out-dated tool it was chosen to rely on a widely adopted standard that is xml. Moreover xml is well supported in JAVA which is PREESM main programming language. Tool such as DIFtoC in DIF also adopt a code generation through libraries, but those libraries are C libraries which limit the tool to C code generation. The xslt approach used in PREESM offers good flexibility and could allow the PREESM software to not be limited to C code generation.

9.12 Code generation example

As to demonstrate the power of the code generation in PREESM, we will use an FFT example. The FFT is inserted in a testbed that generates 2048 time sample on each run of the schedule and gather the 2048 frequency samples (Fig. 9.27). The FFT hierarchical actor operates on 1024 samples, and is repeated twice in each schedule run. This actor is composed of two atomic actors and a hierarchical one. The first atomic actor *mux* acts as a multiplexer, it takes the first 1024 time samples on its first call and passes it to the butterfly step. On the other call, it passes the data from the looping edge. It also provides the relevant twiddle factor for each call of the butterfly. This actor is forced to be repeated $log_2(1024) = 10$ times by the *trig* port. A port with no connection to the outside is simply ignored by the code generation step. The *sort* actor takes the output of the butterflies calls and sorts it for the next run. Its output *data* is broadcasted to both the output interface and the looping edge. The $ring - buffer$ nature of the port allows such graph to be specified, and means that only the last 1024 output samples produced will be carried out to the hierarchy. The butterfly hierarchical actor takes two samples, and a twiddle factor and provides the output with $s_1 + (s_2 \times w)$ and $s_1 - (s_2 \times w)$.

This graph is processed trough an HSDF transformation of the top level, that splits the FFT in two distinct calls, and is then scheduled onto a two cores architecture connected by a TCP link. The scheduler computes the multi-processor schedule (Fig. 9.28). The first core manages the time sample source, performs a fork on the data, and sends an half to second core. It then computes on of the FFTs, and sends its result to the second core. The second core receives the data and processes the second half of the FFT, then receives the frequency samples from the first core and puts it into the sink.

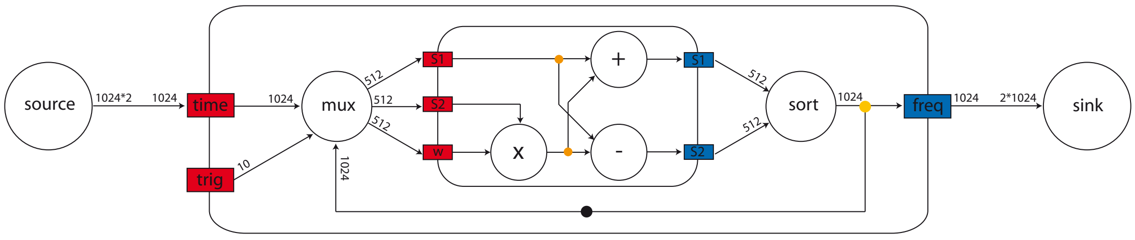


Figure 9.27: *The hierarchical representation of the FFT.*

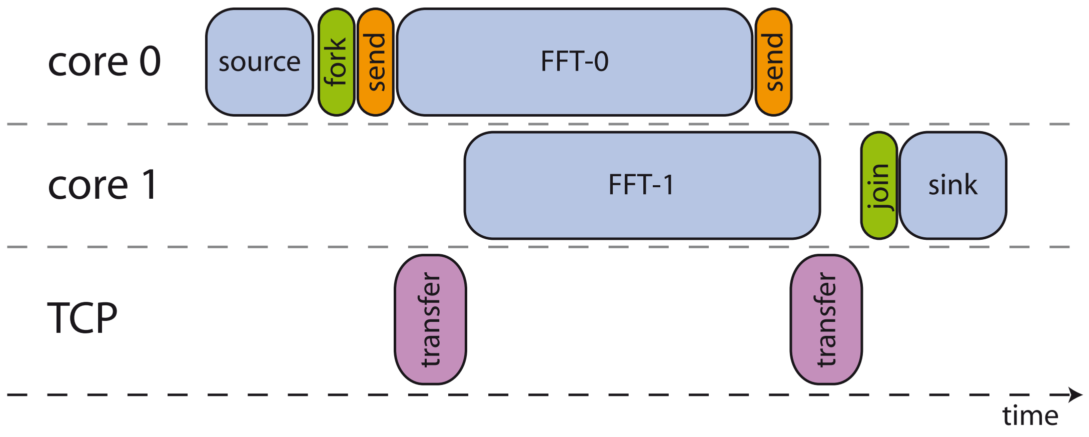


Figure 9.28: *The two-processor schedule of the FFT.*

For the reader convenience the code generation is split into multiple chunks. The true code generation outputs a source file for each processor with only two thread declarations, and no function prototype. In the following the code generation of the FFT hierarchical actor is declared outside the thread when it is normally an inlined code. The code generation showed here uses the local buffer allocation. The code generation of the FFT hierarchical actor shows that the strongly connected elements have been clusterized and the looping edge wrapped around (Fig. 9.29). The broadcast at the end of the FFT actor is generated as a memory copy of the sort actor output in both the looping edge and the output. The code generation for core 0 (Fig. 9.30) shows how the data generated by the source is exploded using both a memory copy for the data to transfer and a pointer for the local data. The code generation for core 1 (Fig. 9.31) shows how the data from the two FFTs are joined into the sink input.

```
void fft(char * time_samples, char * freq_samples){
    char loop_edge [1024];
    for(i = 0; i<10 ; i ++)
    {// cluster of strongly connected components
        char samples_in_1[512];
        char samples_in_2[512];
        char samples_out_1[512];
        char samples_out_2[512];
        char weights[512];
        char data_ouput[1024];
        char * init_datas = &time_samples[((i*(1024))%0)];
        mux(init_datas, loop_edge, samples_in_1, samples_in_2, weights, 1024);
        for(j = 0; j<512 ; j ++)
        {//butterflyStep
            char broadcast[1];
            char *sub_samples_in_1 = &samples_in_1 [((j*(1))%512)];
            char *sub_samples_in_2 = &samples_in_2 [((j*(1))%512)];
            char *sub_samples_out_1 = &samples_out_1 [((j*(1))%512)];
            char *sub_samples_out_1 = &samples_out_2 [((j*(1))%512)];
            char *sub_weights = &sub_weights [((j*(1))%512)];
            char *op1 = &broadcast [((0*1)%1)];
            char *op2 = &broadcast [((0*1)%1)];
            mult(sub_samples_in_2, sub_weights, res_in);
            add(sub_samples_in_1, op1, sub_samples_out_1);
            sub(sub_samples_in_1, op2, sub_samples_out_2);
        }
        sort(samples_out_1, samples_out_2, data_ouput, 1024);
        {//broadcast
           memcpy(freq_samples, data_ouput, 1024*sizeof(char));
           memcpy(loop_edge, data_ouput, 1024*sizeof(char));
        }
    }
}
```

Figure 9.29: *Generated code from the hierarchical FFT actor.*

9.13 Conclusion

The PREESM project was started in 2007 and the code generation is one of the latest contribution of this thesis in cooperation with the work developed in [M.P10]. This plugin was designed based on the work developed for the tool SynDEx and exploits some of its principles. The main difference is that PREESM deals with a truly SDF model with a special interpretation of the hierarchy. Another PREESM difference is the structural choice that allows the user to customize the code generation in particular the available algorithm for memory optimization, and the xslt style-sheets. The execution model is a direct inheritance of SynDEx and is well suited for the generation of multi-core, multi-processor C code. The PREESM code generation provides the user with a human-readable code that can (with a user choice) be memory optimized. The code generation step is also highly customizable thanks to its structure. Indeed the choice of a generic xml code generation with xslt transformation for proper c code, allows to target a wide range of architectures and OSes. Moreover this code generation plugin is theoretically not limited to C code generation and could allow to target hardware programmable components such as FPGA. In addition one could also decide to provide its own buffer allocator implementation for different optimizations of the memory footprint. The open source status of the PREESM project makes it attractive for any professionals that have to deal with multi-core code generation. One can also want to contribute to the source code to implement its own code generation needs.

```
#include "../../Lib_com/include/x86.h"

// Buffer declarations
char fft_freq_samples_1[1024];
char fft_time_samples[2048];
char *fft_time_samples_1;
char fft_time_samples_0[1024];
semaphore sem_0[4];

DWORD WINAPI computationThread_Core0( LPVOID lpParam );
DWORD WINAPI communicationThread_Core0( LPVOID lpParam );

DWORD WINAPI computationThread_Core0( LPVOID lpParam ){
    // Buffer declarations
    long i ;
    long j ;

    { // init
        semaphoreInit(sem_0, 4/*semNumber*/, Core1);
        CreateThread(NULL,8000,communicationThread_Core1,NULL,0,NULL);
        ReleaseSemaphore(sem_0[0],1,NULL); //empty
        ReleaseSemaphore(sem_0[2],1,NULL); //empty
    }

    for(;;){ // loop
        source(fft_time_samples);
        WaitForSingleObject(sem_0[0],INFINITE); //sem wait
        { //fork
            fft_time_samples_1 = &fft_time_samples[1024];
            memcpy(fft_time_sample_0, &fft_time_samples[0], 1024*sizeof(char));
        }
        ReleaseSemaphore(sem_0[1],1,NULL); //sem post
        WaitForSingleObject(sem_0[2],INFINITE); //sem wait
        fft(fft_time_samples_1, fft_freq_samples_1);
        ReleaseSemaphore(sem_0[3],1,NULL); //sem post
    }

    return 0;
}//computationThread

DWORD WINAPI communicationThread_Core0( LPVOID lpParam ){
    // Buffer declarations

    { // init
        Com_Init(MEDIUM_SEND,TCP_1,Core1,Core0);
    }

    for(;;){ // loop
        WaitForSingleObject(sem_0[1],INFINITE); //sem wait
        sendData(TCP_1,Core1,Core0,fft_time_sample_1,1024*sizeof(char));
        ReleaseSemaphore(sem_0[0],1,NULL); //sem post
        WaitForSingleObject(sem_0[3],INFINITE); //sem wait
        sendData(TCP_1,Core1,Core0,fft_freq_samples_1,1024*sizeof(char));
        ReleaseSemaphore(sem_0[2],1,NULL); //sem post
    }

    return 0;
}//communicationThread
```

Figure 9.30: *Code generation for core 0.*

```
#include "../../Lib_com/include/x86.h"

// Buffer declarations
char sink_data[2048];
char time_samples_0[1024];
char fft_freq_samples_1[1024];
char fft_freq_samples_0[1024];
semaphore sem[4];

DWORD WINAPI computationThread_Core1( LPVOID lpParam );
DWORD WINAPI communicationThread_Core1( LPVOID lpParam );

DWORD WINAPI computationThread_Core0( LPVOID lpParam ){
    // Buffer declarations
    long i ;
    long j ;

    { // init
        semaphoreInit(sem, 4/*semNumber*/, Core0);
        CreateThread(NULL,8000,communicationThread_Core0,NULL,0,NULL);
    }

    for(;;){ // loop
        WaitForSingleObject(sem[0],INFINITE); //full
        fft(time_samples_0, fft_freq_samples_0);
        ReleaseSemaphore(sem[1],1,NULL); //empty
        WaitForSingleObject(sem[2],INFINITE); //full
        {//join
            memcpy(&sink_data[0], fft_freq_samples_0, 1024*sizeof(char));
            memcpy(&sink_data[1024], fft_freq_samples_1, 1024*sizeof(char));
        }
        ReleaseSemaphore(sem[3],1,NULL); //empty
        sink(sink_data);
    }

    return 0;
}//computationThread

DWORD WINAPI communicationThread_Core1( LPVOID lpParam ){
    // Buffer declarations

    { // init
        Com_Init(MEDIUM_RCV,TCP_1,Core1,Core0);
        ReleaseSemaphore(sem[1],1,NULL); //empty
        ReleaseSemaphore(sem[3],1,NULL); //empty
    }

    for(;;){ // loop
        WaitForSingleObject(sem[1],INFINITE); //empty
        receiveData(TCP_1,Core1,Core0,time_samples_0,1024*sizeof(char));
        ReleaseSemaphore(sem[0],1,NULL); //full
        WaitForSingleObject(sem[3],INFINITE); //empty
        receiveData(TCP_1,Core1,Core0,fft_freq_samples_1,1024*sizeof(char));
        ReleaseSemaphore(sem[2],1,NULL); //full
    }

    return 0;
}//communicationThread
```

Figure 9.31: *Code generation for core 1.*

Chapter 10

Loop partitioning techniques for Interface based Synchronous Dataflow

10.1 Introduction

The synchronous data flow model allows to specify a network exposing most of the parallelism of the application. The network behavior is deduced from the data-dependencies between the actors, and extracting a valid schedule is then relatively simple. In a rapid prototyping framework, the application specification is optimized before scheduling as depicted in Figure 10.1. When trying to optimize such network for a parallel architecture, one must consider the parallelism existing between multiple invocations of the actors. The homogeneous synchronous data flow model that can be extracted from the SDF representation exposes all the parallelism of the application but leads to an explosion of the number of vertices to schedule. As no architecture has an infinite degree of parallelism, one could choose to extract an amount of parallelism adapted to the target architecture. When using a hierarchical specification of the application, all the parallelism embedded into the hierarchy levels remains unavailable for the scheduler. To extract all the parallelism, the best approach would be to flatten all the hierarchy levels to expose the fine grain parallelism. Once again all this available parallelism might not be adapted to the level of parallelism available on the target architecture.

Those two issues can be independently tackled but are tightly coupled and must be treated jointly to provide the best solution. In the following we will introduce two methods inherited from researches on systolic arrays.

In the first part we will show how the iteration domain projection technique described in Section 7.4 can be applied onto Interface based SDF graphs [PBR10].

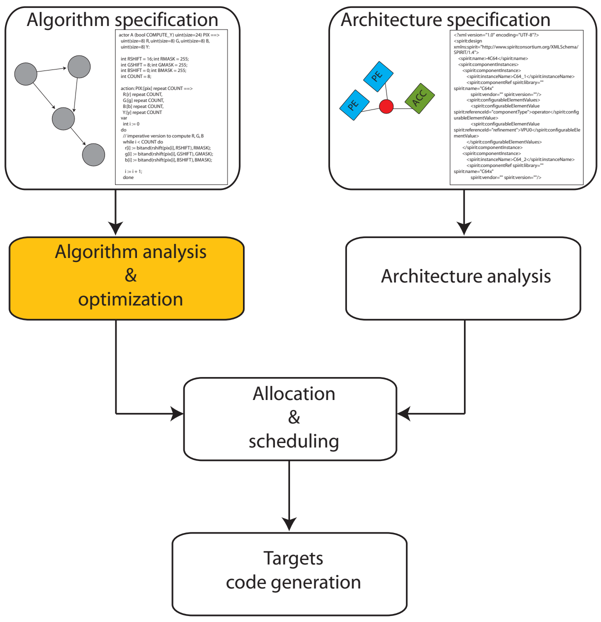


Figure 10.1: *Specification optimization phase in the rapid prototyping workflow.*

The second part shows that tiling technique can also be applied with the same requirements.

10.2 Iteration domain projection technique

Loop partitioning technique described in previous section reveals the parallelism nested into the loops by using basic linear algebra and gives a set of results. As seen previously, interface-based hierarchy suffers from a lack of parallelism. All the embedded parallelism remains unavailable for the scheduler, making an application hard to optimize on a parallel architecture. Interface-based hierarchy being close to code nesting, it seems appropriate to tap into nested loops partitioning techniques to extract parallelism. The nested loops code structure could be defined as follow in the Interface-based Synchronous Data-Flow model:

Definition A nested loop of depth n is a structure composed of n nested hierarchical actors with a repetition factor greater than one, for which each actor, excluded the n^{th} one, contains only one actor.

In order to exploit this optimization technique we must be able to extract the distance vector from the hierarchical description, thus allowing to have a relevant representation for the partitioning. Then having the different projection vectors and their respective resulting execution domains, we must be able to map back this representation into a SDF graph.

10.2.1 Distance vector extraction from interface-based SDF

The Synchronous Data-flow paradigm brings some limitations to the representation.

- In the data-flow paradigm, actors produce tokens that can then be consumed. A data-flow representation cannot contain other dependencies than the flow dependency.
- In the SDF paradigm all the data are represented by edges. Thus all the data of a network are considered disjoint.
- In the SDF model, data are uni-dimensional and atomic (token). It means that you cannot have multi-dimensional access to a data.

The third limitation shows that, the basic SDF representation does not allow to extract distance vector. The hierarchical SDF allows factorized representation and therefore allows to represent edges as multi-dimensional data over the iteration domain. As data are being disjoint, only recursive edge ($source(e) = sink(e)$) can carry an inter iteration dependency. It means that the analyze only has to be carried out on this specific kind of edge. For our purpose, we will consider a recursive edge as an array of size $(q(source(e)) * c(e)) + d(e)$.

Given a vertex a with $q(a) > 1$ and a recursive edge e_0 with $source(e_0) = target(e_0) = a$ and $d(e_0)\%c(e_0) = 0$. The index vector for the *read* accesses to the data carried by e_0 is $\vec{r} = [i_0 - (d(e_0)/c(e_0))]$, and the index vector for the *write* accesses to the data carried by e_0 is $\vec{w} = [i_0]$. Thus the distance vector between the iteration of a is $\vec{\tau} = \vec{w} - \vec{r} = [d(e_0)/c(e_0)]$.

Let us now consider that a is a hierarchical actor that contains one actor b with $q(b) > 1$ and a recursive edge e_1 with $source(e_1) = target(e_1) = b$ and $d(e_1)\%c(e_1) = 0$. Given that edge e_1 has a local scope in a hierarchical representation, the data carried by e_1 can be represented as an array of size $(q(source(e_1)) * c(e_1)) + d(e_1)$ itself contained in an array of size $q(a)$. Then the index vector for the *read* accesses to the data carried by e_1 is $\vec{r} = [i_0, i_1 - (d(e_1)/c(e_1))]$, and the index vector for the *write* accesses to the data carried by e_1 is $\vec{w} = [i_0, i_1]$. Thus the distance vector between iteration of b is $\vec{\tau} = \vec{w} - \vec{r} = [0, (d(e_1)/c(e_1))]$.

By extension the distance vector for a recursive edge at the N^{th} loop of a nested loops structure is a vector of size N with the $(N-1)^{th}$ element being $d(e_{N-1})$. Thus the dependency matrix of an SDF nested loops is a triangular matrix.

Using the distance vector of the application we can now directly perform the analysis described in the method above and reveal the parallelism nested into the hierarchy. Using the analysis results, it is then possible to synthesize a new SDF network performing the same computation.

10.2.2 SDF network synthesis using analysis results

The network of computing element resulting from the projection is itself an SDF graph. Using information given by the allocation vector we can determine the points of the execution domains computed by each cell and consequently distribute the input data among the cells using explode and broadcast vertices. The output data can also be sorted out using implode vertices and circular buffers.

The SDF graph then needs to be timed using delay to ensure a proper execution of strongly connected components. In a systolic array all the cell are active synchronously. Thus in order to synchronize the computation on the cell network, the communication channel must consist of a register whose size allows to synchronize the computation. In the SDF paradigm, computations are synchronized by data, and actors are not triggered synchronously but sequentially if they share data. Thus if the resulting network contains strongly connected components, delays have to be added in order to time the graph. A proper execution guarantees that the last data available on a communication link is the valid one for the execution of the sub-graph.

For a set of strongly connected components C and for each computing element $E_n \in C$ we can determine the hyperplane in the iteration domain containing the last computation performed by E_n. The element with the hyperplane that has the shortest distance to the hyperplane of points computed at $t = 0$, is the element that must be scheduled first. This means that its incoming edges of belonging to the strongly connected set must carry a delay.

The synthesized network can then be used with ring_buffer vertex, to ensure the output data to be the last. The computation performed by the network is strictly the same, with some vertices performing computation outside of the iteration domain (which should be considered null time).

10.2.3 The matrix vector product example

In this section we will use the matrix vector product as a test case for the method described above.

Given a vector V and matrix M, the product $V \times M = R$ can be described using a set of recurrent equations.

$$\begin{cases} R_{i,k} = 0 & if \quad k = 0 \\ R_{i,k} = R_{i,k-1} + v_i m_{i,k} & if \quad 1 \leq k \leq N \\ r_i = R_{i,N} & 0 \end{cases}$$

from this set we can extract a the following system.

Initial state

$$\begin{cases} R_{i,k} = 0 & if \quad k = 0 \\ V_{i,k} = v_k & if \quad i = 0 \\ M_{i,k} = m_{i,k} \end{cases}$$

Calculus equations

$$\begin{cases} R_{i,k} = R_{i,k-1} + V_{i-1,k} M_{i,k} & i \quad 1 \leq k \leq N \\ V_{i,k} = V_{i-1,k} & i \quad 1 \leq k \leq N \\ M_{i,k} = m_{i,k} \end{cases}$$

Output equation

$Ri = R_{i,N}$

The SDF representation extracted from those recurrent equations exposes two levels of hierarchy (see Figure 10.2). The first hierarchy level contains a *vector* $\times$ *scalar* product, and the second hierarchy level represents a *scalar* $\times$ *scalar* product.

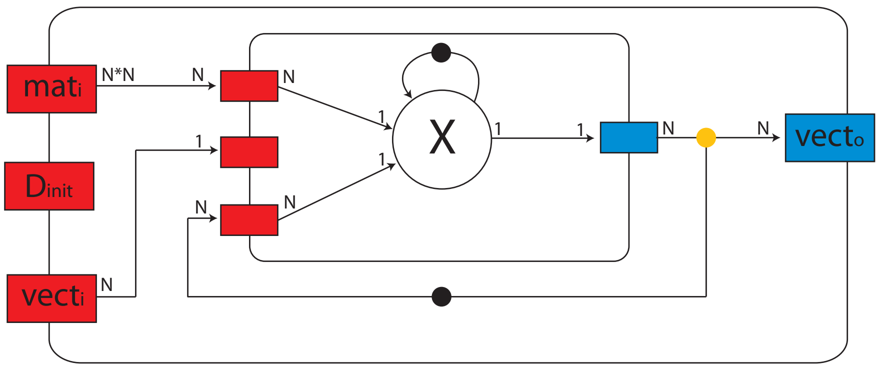


Figure 10.2: *Matrix vector product.*

Network description

The matrix vector product networks, takes a $N \times N$ matrix and a N vector as inputs, and outputs the result as a N vector. The M_i port is the matrix input and the V_i is the vector input. The V_o port is the vector output. The $vect_s cal$ vertex takes two inputs, V_i being a line of the matrix, and $D_i nit$ being an element of the vector. The element in $D_i nit$ initializes the delay token on the recursive edge around the *mac* operation. The Acc_i port takes the vector in which the result is accumulated. The mac operation takes two scalars, one from the the matrix line, one from the delay (being an element of the input vector), multiply them and adds the result with the input accumulating vector. The valid schedule for the graph is then:

$N \times \{N \times mac\}$

The schedule takes advantage of the special behavior of the port V_o, which behaves as a ring buffer of size N. Thus the data contained in V_o at the end of the schedule, is the result of the last N^{th} iteration of the mac operation, that is the valid result.

Distance vector extraction

The index vector for the read operation on the top recursive edge is $\vec{r_o} = [i_0 - 1, i_1]$, and the index vector for the write operation on the top recursive edge is $\vec{w_o} = [i_0, i_1]$. Thus the distance vector is $\vec{\tau_0} = \vec{w_o} - \vec{r_o} = [1, 0]$. The index vector for the read operation on the inner recursive edge is $\vec{r_1} = [i_0, i_1 - 1]$, and the index vector for the write operation on the inner recursive edge is $\vec{w_1} = [i_0, i_1]$. Thus the distance vector is $\vec{\tau_1} = \vec{w_1} - \vec{r_1} = [0, 1]$.

Using Lamport's method [Lam74] we can determine that the time vector minimizing parallel execution time for this application is $\tau = [1, 1]$. Based on this time vector, a set of projection vector can be determined :

$$s_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \; s_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \; s_3 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

The following analysis will be carried out using the projection vector s_3. The uni-modular matrix S_3 is

$$S_3 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

$$S_3^{-1} = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}$$

The allocation function is then $A_3 = [-1, 1]$. Using this allocation function we can now determine how the points of the computation domain are allocated onto the execution domain. The extremes of the allocation function in the iteration domain are $-N$ and N meaning that the execution domain is of size $(2 \times N) - 1$. The end of the analysis will be performed with $N = 3$.

Network synthesis

N being three, the resulting network is composed of 5 vertices. Using the allocation function we can determine the point of the iteration domain that will be computed by each vertex.

- Vertex 0: compute the point $[1, 3]$
- Vertex 1: compute the points $[1, 2]$ and $[2, 3]$
- Vertex 2: compute the points $[1, 1]$, $[2, 2]$ $[3, 3]$
- Vertex 3: compute the points $[2, 1]$ and $[3, 2]$
- Vertex 4: compute the point $[3, 1]$

Computing the topology matrix of the network shows that the repetition factor for each of the actor is 3, as the computation load must be balanced in an SDF. Thus vertices $0, 1, 3, 4$, will compute points outside of the iteration domain. This means that we must consequently time the graph to get sure that the valid data will be the last produced data. For the first strongly connected set $\{V_0, V_1\}$, the hyperplane containing the point $[1, 3]$ as a shorter distance to the hyplerplane containing $[0, 0]$, than the hyperplane containing $[2, 3]$. This means that V_0 must be scheduled before V_1. To consequently time the network we must add a delay on the arc going from V_1 to V_0 (see Figure 10.3). Timing all the strongly connected sets that way leads to a translation of the iteration domain for the vertices $0, 1, 3, 4$ (see Figure 10.4).

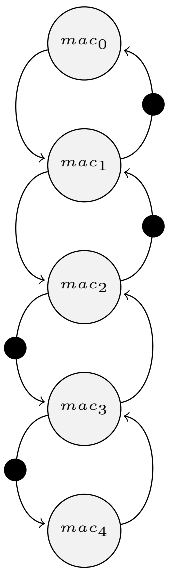


Figure 10.3: *Valid timed network.*

The resulting timed network needs to be connected to input and output ports. Knowing which point of the iteration each vertex implements, we must instantiate a computation less actor that creates for each vertex a valid sequence of input tokens. A second computation less actor gather the output of the actors and outputs the relevant data. For some kind of execution (vector s_1 and s_2) the input distributing actor behaves as a fork, and the output gathering actor behaves as a join. For vector s_3 the actor behavior is more complicated and must output dummy (or zeroed) data for the out of execution domain actors iterations. The information available are sufficient to generate these actors, but adds some complexity to the process.

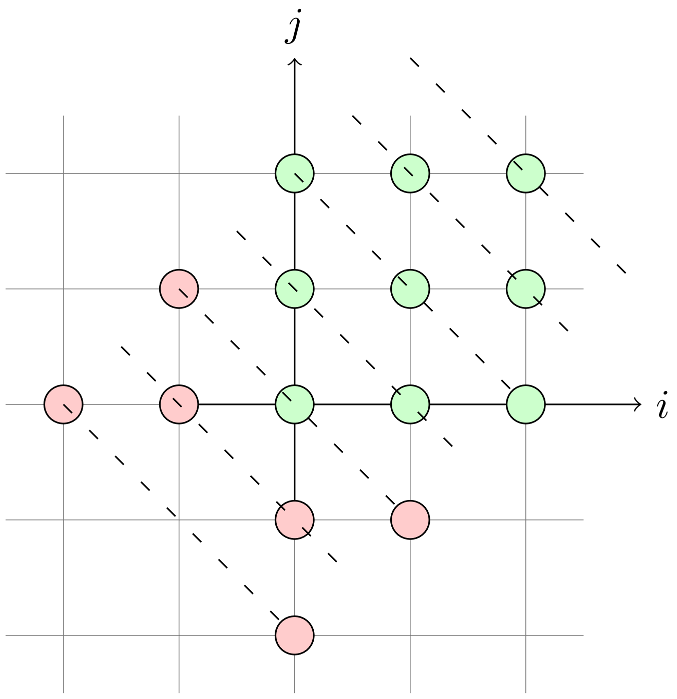


Figure 10.4: *Iteration domain after graph timing.*

The original hierarchical representation had no degree of parallelism, but the resulting representation after transformation reveals five degrees out of nine available for the flat representation. The other available projection vectors would give less parallelism, with more regularity in the computation as the activity rate of cells would be homogeneous over the network.

This technique advantage is that it generates execution pattern like pipeline (vector s_3 in example), vectorization (vectors s_1 and s_2 in example). Its main drawback is that the resulting network size depends on the iteration domain size, and the minimum reachable size is equal to the smallest dimension of the iteration domain.

10.3 Iteration domain tiling

The technique presented in Section 7.5 proposes to partition an iteration domain into small blocks thus reducing communication load and increasing computation granularity. Hierarchical representation in SDF graphs have to face the issue of finding a trade-off between parallelism and computation granularity for better multi-processor implementations. All the available parallelism can be revealed by flattening the hierarchical representation but dramatically increases the number of vertices to schedule and the amount of communications. The technique described previously proposes to partition the iteration domain by finding the best projection of the iteration domain onto an execution domain. This technique increases the parallelism available compared to the hierarchical representation while maintaining the number of vertices to schedule at an average level. While this technique provides the scheduler with a clustered representation, it does not tackle the problem of actor synchronizations. The nested loop partitioning technique described in section 7.5 proposes to partition the iteration domain using “tiles”. Those tiles can be shaped and scaled to fit memory properties of the target architecture and minimize synchronizations between processing elements. This technique appears to be well suited to SDF graph hierarchy partitioning as it deals with the sames issues.

In the following we will demonstrate the use of this technique in the SDF context using results from the previous section and the same matrix vector product example. This technique being based on distance vector extraction from an application, the distance vector extraction from an interface based hierarchical SDF representation is the same than the previously described. Computing the tile shape and size is then the same as described in Section 10.2. The next section will explain the network synthesis from a tiled iteration domain.

10.3.1 Limitations

As to not increase the complexity of the transformed graph, it was chosen to limit this transformation to the use of canonical tiles that can be defined as a scaling of the tile whose edges are the N linearly independent vector of H, with N being the depth of the nested loops. If multiple vectors of D are linearly dependent, one must choose the greatest vector. This allows to have homogeneous tiles over the network, when other tile shape can lead to constructing different tiles on the borders of the iteration domain. In addition different tile shape gives harder to synthesize tile interconnections with CSDF behavior in some cases. Limiting the problem of such tiles also decreases the complexity of the transformation as finding the optimal tile shape is reduced to finding the optimal scaling factor.

10.3.2 SDF network synthesis from partitioned iteration domain

For a tile shape as described in the limitation, synthesizing the network is made simple. In fact such tile shape leads to a canonical tile being a scale-down of the whole iteration domain. The only requirement of this tile compared to the factorized representation is to output data produced by its inner loops. Thus the construction of the canonical tile is performed by taking the SDF nested loop representation and parameterized its iteration domain in respect to the tile bounds. Then arcs corresponding to dependencies in the inner loop are connected through the hierarchical levels by ports. As mentioned in the limitations, the canonical tile is just a scaling of the tile whose edges are the dependency vectors N linearly independent vectors from the greatest vectors of D. Thus the scaling of the tile is performed by a multiplication of the canonical tile by the scaling vector s. As the shape of the tile is limited to the one described above, the communication and computation volume ratio remain the same when scaling. Thus one must determine the best scaling factor based on the available parallelism of the target architecture.

Bringing inner loops iteration values to the outside of the tile must be performed in the following way. For each level of hierarchy of the nested loop starting from the inner loop, for all edges carrying a delay greater than zero:

1. The data carried by the edge must be broadcasted and outputs on a sink interface if not already with a consumption equal to the delay value as depicted in Figure 10.5 and 10.7.

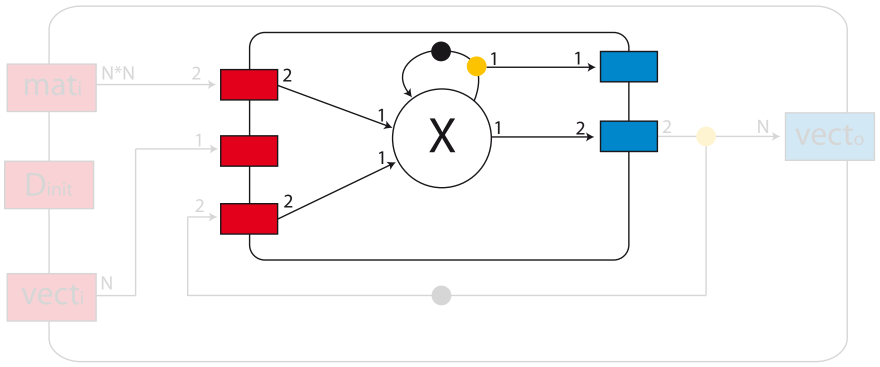


Figure 10.5: *Connecting inner loop* 0 *iteration data to the outside.*

2. All unconnected output ports p_o (created by the previous step in a previous hierarchy level) of the single actor A_i (except for the inner most hierarchy level) must get connected to a new sink interface with a consumption value equal to $q(A_i) * prod(p_o)$ as depicted in Figure 10.6.

When the canonical tile has been defined, one must create the tile interconnections to form the final tiled network. To do so, one has to take the canonical tile, and duplicate it as many time as needed in the final network. Then, we can create the interconnections with its neighbors. As we limit the study to canonical tiles generated by the dependency matrix (see 10.3.1), the tile has just to be connected to its direct neighbors, which eases the connection step. We first connect the canonical tile to the actors that do not produce tokens involved in the iteration dependency. Then we connect the canonical tile to the actors that produce the token that initializes the data on the outer most delay (dimension 0). The canonical tile is then repeated along the dimension corresponding to the data mentioned above. Then the output

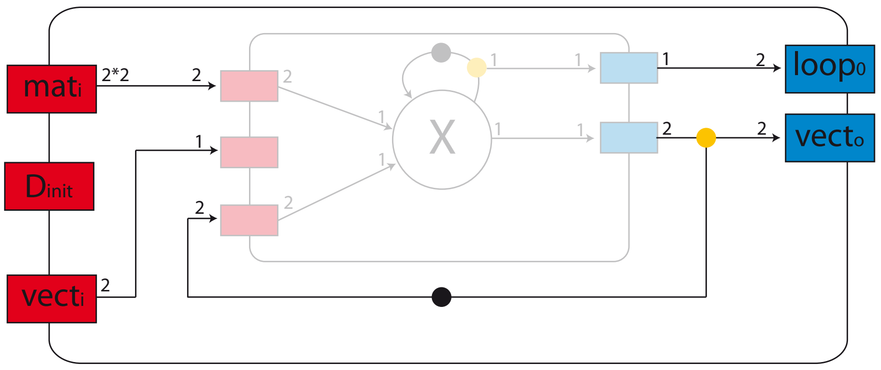


Figure 10.6: *Connecting unconnected output port of the inner actor.*

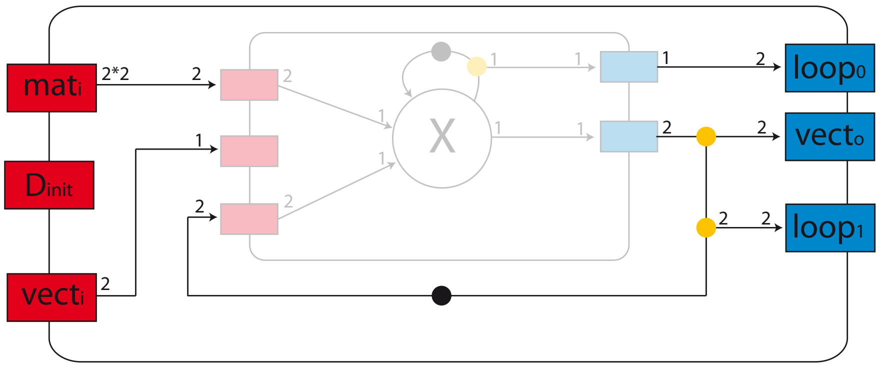


Figure 10.7: *Connecting inner loop 1 iteration data to the oustide.*

data are connected to the relevant actor. In a second step, the created dimension (dimension 0) is connected to the input data that initializes the delay on the dimension 1 (one level of hierarchy deeper), and the dimension 0 is repeated along dimension 1. The output data of dimension 1 are connected to the relevant actor. The procedure iterates as many times as needed by the network. This procedure is depicted in Figure 10.10 and the equivalent java object code is described in Figure 10.8.

```
public void connectTiles(Vertex cannonicalTile, int * dims, int nbDims){
cannonicalTile.connectToInputData(0);
Dimension<Vertex> dim = new Dimension(cannonicalTile, dims[0]);
dim.connectToOutputData(0);
for(int i = 1 ; i < nbDims ; i ++){
        dim.connectToInputData(i);
        Dimension<Vertex> dim = new Dimension(dim, dims[i]);
        dim.connectToOutputData(i);
}
}
```

Figure 10.8: *Procedure for connecting tiles into the network.*

10.3.3 The matrix vector product example

The application used for this example is the same factorized representation of the matrix vector product used in Section 7.4. In the following the number of column N_c and the number of row N_l equal to four $N_c = N_l = 4$. The dependency matrix for this matrix vector product is:

$$D = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

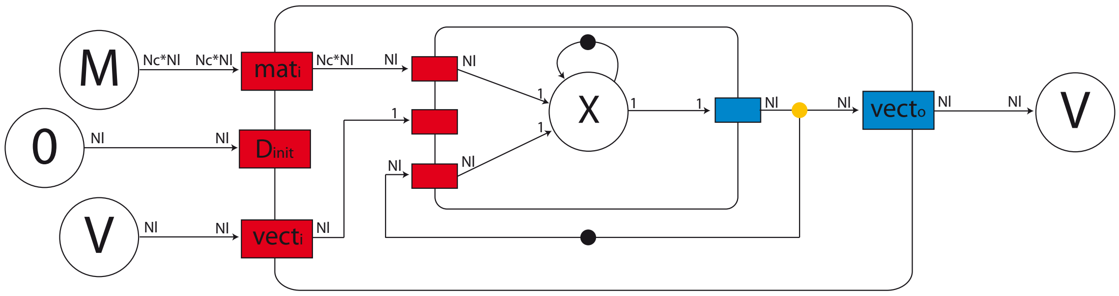


Figure 10.9: *Network before tiling.*

Based on this shape one must then determine the scaling vector for this tile that fits best the target architecture. In the following we will use the scaling vector $S = [22]$ that gives the canonical tile:

$$P = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$$

Figures 10.5, 10.6 and 10.7 show the transformation of the nested actors to become the canonical tile based on the steps described previously. Figure 10.10 shows the network synthesis steps.

The resulting network (see Figure 10.11)exposes most of the parallelism for a given architecture with 4 degrees of parallelism, in a factorized network that minimizes the number of vertices to schedule by a factor 4 compared to a flat network. The scaling vector can easily be adapted to fit another kind of architecture, and does not have to be homogeneous, meaning that the tile can be scaled in a preferred dimension to fit communication volume reduction purpose.

10.4 Conclusion

We proved that the domain projection method and the domain tiling method can be used in the context of Synchronous Data Flow to extract parallelism from a hierarchical representation using the interface based model. Limitations inherent to synchronous data flow limit the level of optimization that could be performed, but still brings improvements regarding the parallelism extraction. The matrix vector

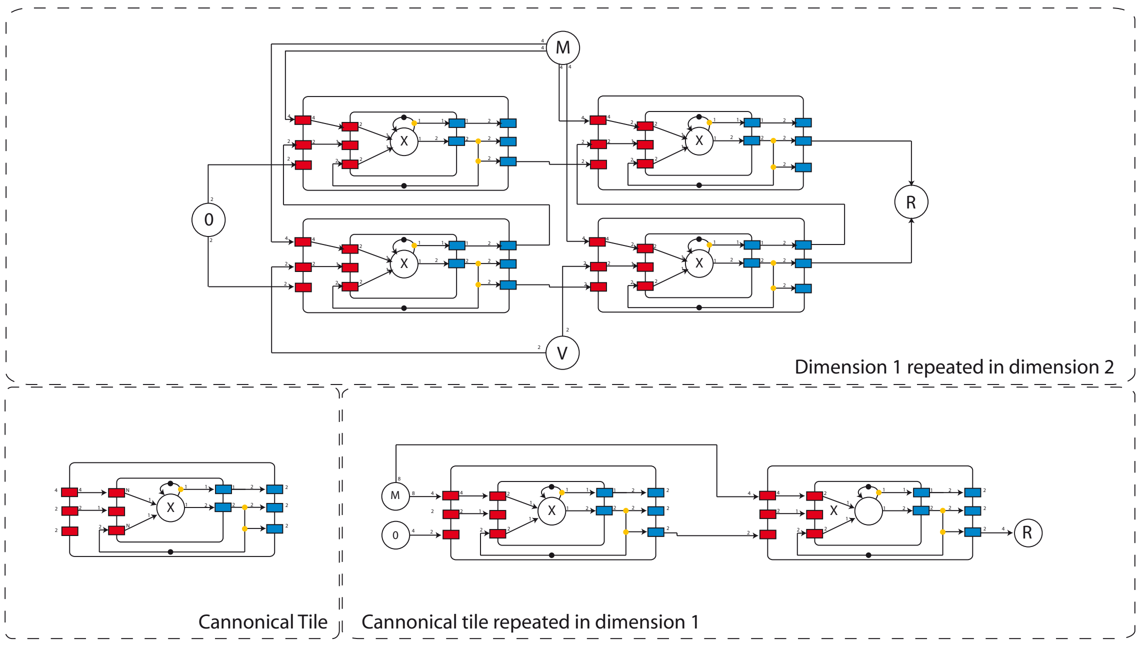

M
0
R
V
X
Dimension 1 repeated in dimension 2
Cannonical Tile
Cannonical tile repeated in dimension 1

Figure 10.10: *Steps of network synthesis.*

example used shows the kind of structural optimization that can be extracted from the hierarchical representation.

Those methods can find application in other multi-dimensional problems such as video or image coding/decoding. We have identify critical zones in the MPEG-4 encoder that could benefit from these transformations. In the DC intra prediction mode of the MPEG-4 AVC/H.264 encoder (mode 2 of intra prediction), blocks of a macro-block are constructed using data from the upper and left blocks (see Figure 10.12). It means for macro-block encoding the blocks that reside on a macro-block top and left edges, the encoder needs data from the upper and left macro-block. Thus it is possible to see this problem as a nested loop, which outer loop iterates on the lines, and inner loop iterates on the macro-blocks of a line with dependencies between lines (from top to bottom) and dependencies between macro-blocks (from left to right). The two described techniques allow to extract parallelism from the factorized description of the MPEG-4 AVC/H.264 encoder at macro-block level. For a full HD picture (1080 lines of 1920 pixels) the number of luminance macro-blocks is 8100 thus, the flat representation exposes too much parallelism for most multi-core architectures. Using the tiling technique, the available parallelism could be greatly improved to fit onto today's multi dsp-cores architecture (up to 6 DSP cores on Texas Instrument TMS320C6472). Moreover using the tiling technique followed by a domain projection of the tiled network, one could construct a pipelined execution of the decoder. Unfortunately due to a lack of time, we did not manage to implement the MPEG-4 AVC/H.264 encoder in PREESM to test the described features.

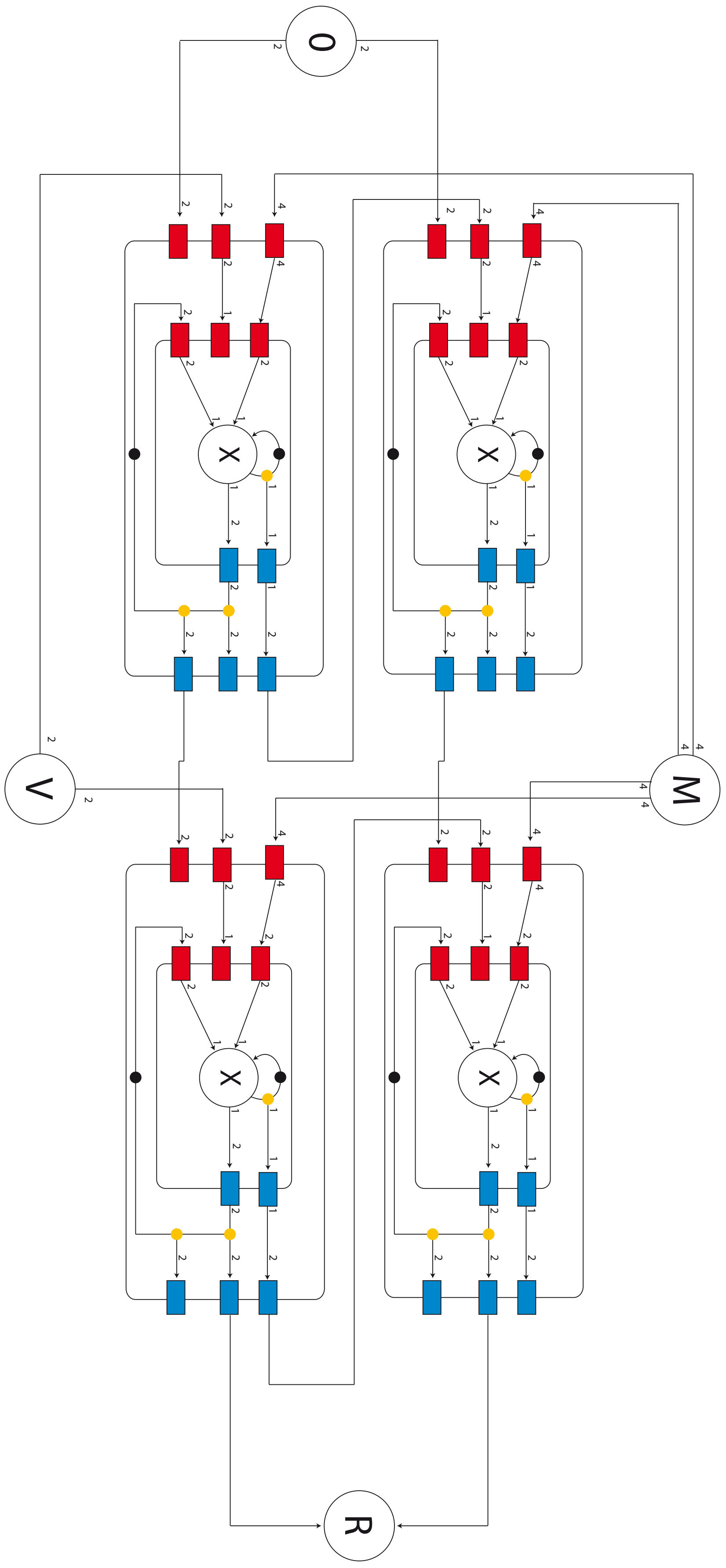


Figure 10.11: *Network after tiling.*

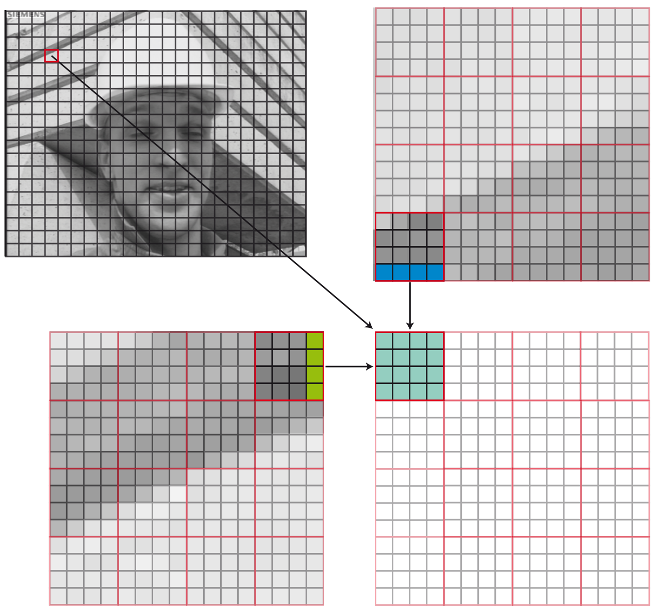


Figure 10.12: *Picture of* 16×16 *macro-blocks, and zoom in on macro-block encoding/decoding dependencies in DC mode (mode* 2*).*

Chapter 11

Conclusion, Current Status and Future Work

11.1 Conclusion

In this thesis we have presented the context of rapid prototyping and associated computation models. The contribution of this thesis targets implementation in the PREESM software but other tools such as DIF or Ptolemy could take advantage of this work. Chapter 8 shows how a new hierarchy model can ease the designer work and provide the designer with a new interpretation enabling new SDF behavior. Moreover the top-down design approach is closer to a design flow in which the application is described in several steps starting from coarse grain to lead to fine grain description.

Chapter 9 describes how the code generation can take advantage of the hierarchy semantic to generate C code in a comprehensible format. This code generation method also gives more flexibility for the designer in term of memory allocation and regarding the wide range of architecture it can target through the use of xslt transformation files. This code generation method could also be extended to language like VHDL, by defining new transformations files.

Chapter 10 gives methods to exploit parallelism embedded into hierarchy with strong connectivity. The first method proves the validity of distance vector extraction from a nested hierarchy in a synchronous data flow graph. The domain projection transformation leads to a valid solution, but the network synthesis is not simple and involves using data organization vertices for proper execution. The second method uses results from the first method, and gives a clustered representation that best fits the purpose of parallelism extraction. Moreover the second method is much more flexible, as the choice of the scaling vector allows to create tiles with computation volumes adapted to the target interface.

The works produced in this thesis converge onto improving the rapid prototyping framework PREESM. Other tools such as DIF, could also take advantage of the hierarchy model, and the transformations proposed in 10.

11.2 Current Status

The work presented in this thesis has been partially implemented in the tool PREESM. The hierarchical model is now the base specification model that PREESM takes as an input. This model is implemented in the graph library SDF4J (SDF for Java)[1]. The library provides basic transformation such as hierarchy flattening, that extracts the hierarchy information to provide a flat representation and HSDF transformation, transforming the SDF graph into an HSDF graph. The DAG transformation is also implemented to provide an input to the scheduling step of PREESM. The code generation is fully implemented as a PREESM plugin and is used to generate code for applications such as LTE, or MPEG4-SP decoder. Some allocation strategies such as buffer lifetime optimizations are still a work in progress, but should soon be fully implemented. The loop transformations are currently under work for implementation in SDF4J. The tiling transformation is a priority in order to improve our MPEG4-SP decoder description for multi-core code generation.

11.3 Future Work

The work developed on loop partitioning could be of interest for other Data Flow models. For example the CAL language rely on the Data Flow Process Network model, and is used in the Orcc[2] tool to generate single-processor code. To efficiently target multi-cores, one of the work in progress is to identify data flow models such as SDF or PSDF in the description in order to apply optimization techniques. SDF zone in a CAL description could benefit of the loop transformation work for optimized multi-processor implementation. Another contribution could be to extend the loop transformation to PSDF description in order to optimize applications with dynamically controlled parameters.The PSDF model is currently under implementation in the SDF4J library, but still lacks some flattening or transformations technique that would allow to extract parallelism from a PSDF description. A drawback of the SDF model is that it needs to be globally synchronous, while the PSDF introduces the concept of local synchrony. One may like to use the concept of local synchrony to specify conditional exclusive data-path in the graph. This would

1. http://sdf4j.sf.net
2. http://orcc.sf.net/

allow to describe conditional actors whose behavior could be specified by a finite set of disconnected networks. In an MPEG decoder for example, the macro-block decoding structure is conditioned by the kind of coding used. Thus to specify the macro-block decoding actor, one could want to specify one network for each behavior. This representation would lead to exclusive data-paths in the graph. This kind of specification can be made using PSDF but introduces a lot of complexity for the designer. It could be decided to describe a sub-set of PSDF that takes advantage of the fixed interface of Interface-based SDF to ensure local synchrony in all cases. Then each exclusive data-path could be independently treated and data routed at run-time. Thus the designer would not have to know much about the PSDF semantic, and would just have to specify the network description for each condition value. A conditioning port would carry the token that condition the network description to execute and would be interpreted as the sub-init stage of a PSDF graph.

Considering the rapid prototyping workflow, a future work would be to perform the graph transformations, and scheduling in a an iterative way (see Figure 11.1). Thus, in a first iteration of the workflow, the graph could get no transformation, then be scheduled and the solution evaluated. Then by analyzing the solution, the tool could decide if more or less parallelism has to be extracted to improve the solution, and pass parameters to the graph transformation step, and scheduling step for another run of the workflow. The workflow could keep running this way, until the solution found reaches a minimum. This kind of workflow would take much longer to perform, than a single pass workflow, but could provide better result. A lot of work would have to be done on the analyzing step, that must decide whether or not the solution as reached a minimum, and define parameters that would lead transformation and scheduling for the another workflow pass.

As a final conclusion, the introduced model of hierarchy coupled with the proposed transformation, seems to be of great interest for a rapid prototyping framework. A lot of effort has been put into the implementation of the tool PREESM to attract industrial partners to collaborate. Up to this point, Texas Instrument has shown a lot of interest in using the tool, and we hope to attract other users or contributors in the near future.

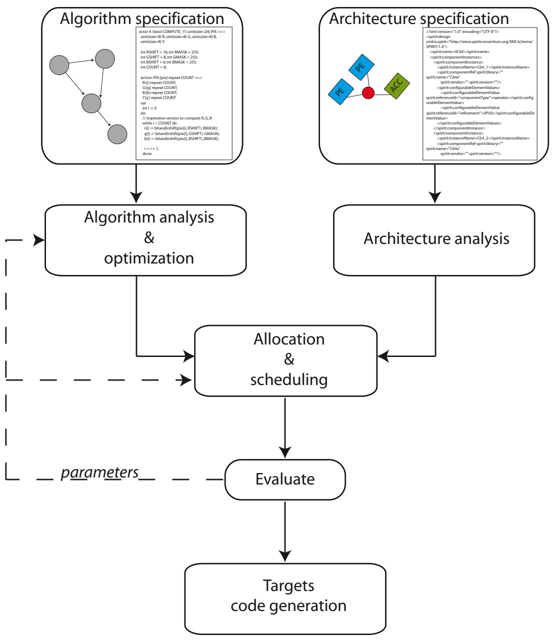


Figure 11.1: *The rapid prototyping workflow used in an iterative way*

List of Figures

Bibliography

[ACD74] Thomas L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Commun. ACM*, 17(12):685–690, 1974. 61

[AYF00] A.Darte, Y.Robert, and F.Vivien. *Scheduling and automatic parallelization*. Birkhauser, 2000. ISBN 0-8176-4149-1. 10

[BB01] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing*, 49(10):2408–2421, October 2001. 42, 51, 52, 85

[BBE$^+$08] S. Bhattacharyya, G. Brebner, J. Eker, J. Janneck, M. Mattavelli, C. von Platen, and M. Raulet. OpenDF - a dataflow toolset for reconfigurable hardware and multicore systems. SIGARCH Comput. Archit. News, 2008. 32

[BDRR94] Pierre Boulet, Alain Darte, Tanguy Risset, and Yves Robert. (pen)-ultimate tiling? *Integr. VLSI J.*, 17(1):33–51, 1994. 75

[BEJ$^+$09] S. S. Bhattacharyya, J. Eker, J. W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet. Overview of the MPEG Reconfigurable Video Coding Framework. *Springer journal of Signal Processing Systems. Special Issue on Reconfigurable Video Coding*, 2009. 59

[Bel06] P. Belanovic. *An Open Tool Integration Environment for Efficient Design of Embedded Systems in Wireless Communications*. PhD thesis, Technischen Universitat Wien, 2006. 32

[BELP95] G. Bilsen, M. Engels, R. Lauwereins, and JA Peperstraete. Cyclo-static data flow. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 5, 1995. 42

[BJK$^+$95a] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING*, 37:207—216, 1995. 32

[BJK+95b] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *ACM SigPlan Notices*, 30(8):216, 1995. 60

[Buc93] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model.* PhD thesis, EECS Department, University of California, Berkeley, 1993. 42

[CDS96] S. Carr, C. Ding, and P. Sweany. Improving software pipelining with unroll-and-jam. In *PROCEEDINGS OF THE HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES*, volume 29, pages 183–192. Citeseer, 1996. 69

[CI94] J. P. Calvez and D. Isidoro. A codesign experience with the mcse methodology. In *CODES '94: Proceedings of the 3rd international workshop on Hardware/software co-design*, pages 140–147, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. 59

[DEY+09] Andreas Dahlin, Johan Ersfolk, Guyfu Yang, Haitham Habli, and Johan Lilius. The canals language and its compiler. In *SCOPES '09: Proceedings of th 12th International Workshop on Software and Compilers for Embedded Systems*, pages 43–52, New York, NY, USA, 2009. ACM. 58

[DMI98] L. Dagum, R. Menon, and S.G. Inc. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998. 60

[DV95] A. Darte and F. Vivien. A classification of nested loops parallelization algorithms. In *Emerging Technologies and Factory Automation, 1995. ETFA '95, Proceedings., 1995 INRIA/IEEE Symposium on*, volume 1, pages 217 –234 vol.1, 10-13 1995. 69

[ecl] Eclipse Open Source IDE : Available Online. http://www.eclipse.org/downloads/. 30

[EJ03a] J. Eker and J. Janneck. CAL Language Report. Technical Report ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, December 2003. 59

[EJ03b] J. Eker and J. W. Janneck. CAL Language Report. Technical report, ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, December 2003. 32

[EJL+03] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendor, e Sonia, and S. Yuhong. Taming heterogeneity—the Ptolemy approach. In *Proceedings of the IEEE*, volume 91, January 2003. 59

[Ele97] A. Eleftheriadis. Flavor: A Language for Media Representation. *ACM Int'l Conf. on Multimedia*, pages 1–9, 1997. 35

[GJ79] M.R. Garey and D.S. Johnson. *Computers and intractability*. Freeman San Francisco, 1979. 106

[GJRB10] R. Gu, J.W. Janneck, M. Raulet, and S.S. Bhattacharyya. Exploiting statically schedulable regions in dataflow programs. *Journal of Signal Processing Systems*, pages 1–14, 2010. 62

[GLS99] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of 7th International Workshop on Hardware/Software Co-Design, CODES'99*, Rome, Italy, May 1999. 32

[GR96] R. Govindarajan and S. Rengarajan. Buffer allocation in regular dataflow networks: An approach based on coloring circular-arc graphs. In *HIPC '96: Proceedings of the Third International Conference on High-Performance Computing (HiPC '96)*, page 419, Washington, DC, USA, 1996. IEEE Computer Society. 105

[GS03] T. Grandpierre and Y. Sorel. From algorithm and architecture specification to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *Proceedings of First ACM and IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'03*, Mont Saint-Michel, France, June 2003. 28, 30, 32, 58

[GWP$^+$10] J Gorin, M Wipliez, J Piat, F Prêteux, and M Raulet. An LLVM-based decoder for MPEG Reconfigurable Video Coding. In *To appear in Signal Processing Systems (SiPS) 2010*, 2010. 29

[HKB05] Chia-Jui Hsu, Ming-Yung Ko, and Shuvra S. Bhattacharyya. Software synthesis from the dataflow interchange format. In *SCOPES '05: Proceedings of the 2005 workshop on Software and compilers for embedded systems*, pages 37–49, New York, NY, USA, 2005. ACM. 54

[HKK$^+$04] C.-J. Hsu, F. Keceli, M.-Y. Ko, S. Shahparnia, and S. S. Bhattacharyya. Dif: An interchange format for dataflow-based design tools. *In Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation, Samos 2004*, 2004. 32

[HPB08] C. Hsu, J. L. Pino, and S. S. Bhattacharyya. Multithreaded simulation for synchronous dataflow graphs. In *Proceedings of the Design Automation Conference*, pages 331–336, Anaheim, California, June 2008. 83

[Int] International Standard ISO/IEC FDIS 23001-5. *MPEG systems technologies - Part 5: Bitstream Syntax Description Language (BSDL).* 34

[iso04] *ISO/IEC14496 Coding of audio-visual objects.* 2004. 34, 35

[ISO09] ISO/IEC FDIS 23001-4. *MPEG systems technologies – Part 4: Codec Configuration Representation*, 2009. 59

[J. 07] J. Thomas-Kerr and I. Burnett and C. Ritz and S. Devillers and D. De Schijver and R. Van de Walle. Is That a Fish in Your Ear? A Universal Metalanguage for Multimedia. *IEEE Multimedia*, 14(2):72–77, 2007. 35

[jHKyK$^+$04] Chia jui Hsu, Fuat Keceli, Ming yung Ko, Shahrooz Shahparnia, and Shuvra S. Bhattacharyya. Dif: An interchange format for dataflow-based design tools. In *in Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation, Samos*, pages 423–432, 2004. 54

[KA98] Y.K. Kwok and I. Ahmad. Benchmarking the task graph scheduling algorithms. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998*, pages 531–537, 1998. 62

[KA01] K. Kennedy and J.R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2001. 69

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974. 41

[Kie91] HA Kierstead. A polynomial time approximation algorithm for dynamic storage allocation. *Discrete Mathematics*, 88(2-3):231–237, 1991. 106

[KKBP91] D. Kulkarni, K. G. Kumar, A. Basu, and A. Paulraj. Loop partitioning for distributed memory multiprocessors as unimodular transformations. In *ICS '91: Proceedings of the 5th international conference on Supercomputing*, pages 206–215, New York, NY, USA, 1991. ACM. 69

[KSLB03] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003. 32

[Lam74] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974. 13, 69, 70, 126

[Lee01] E.A. Lee. Overview of the ptolemy project. *Technical memorandum UCB/ERL M01/11, University of California at Berkeley*, 2001. 32

[Lee06] E.A. Lee. The problem with threads. *EECS in IEEE Computer*, 39(5):33–42, 2006. 30

[LHJ$^+$01] Edward A. Lee, C. Hylands, J. Janneck, J. Davis II, J. Liu, X. Liu, S. Neuendorffer, S. Sachs M. Stewart, K. Vissers, and P. Whitaker. Overview of the ptolemy project. Technical Report UCB/ERL M01/11, EECS Department, University of California, Berkeley, 2001. 56

[LHS90] L.S. Liu, C.W. Ho, and J.P. Sheu. On the parallelism of nested for-loops using index shift method. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume 2, pages 119–123, 1990. 70

[LM87a] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, 1987. 89

[LM87b] E.A Lee and D.G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, sept 1987. 41, 44, 45

[LMTKJ07] Christophe Lucarz, Marco Mattavelli, Joseph Thomas-Kerr, and Jörn Janneck. Reconfigurable Media Coding: A New Specification Model for Multimedia Coders. In *IEEE Workshop on Signal Processing Systems*, pages 481–486, 2007. 35

[LP95] Edward A. Lee and Thomas M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995. 41, 44

[LPM09] Christophe Lucarz, Jonathan Piat, and Marco Mattavelli. Automatic Synthesis of Parsers and Validation of Bitstreams Within the MPEG Reconfigurable Video Coding Framework. *Journal of Signal Processing Systems*, page online, 07 2009. 29

[Man97] Naraig Manjikian. Combining loop fusion with prefetching on shared-memory multiprocessors. In *ICPP*, pages 78–, 1997. 69

[MB01] P.K. Murthy and S.S. Bhattacharyya. Shared buffer implementations of signal processing systems usinglifetime analysis techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(2):177–198, 2001. 106

[MBL97] P.K. Murthy, S.S. Bhattacharyya, and E.A. Lee. Joint minimization of code and data for synchronous dataflow programs. *Formal Methods in System Design*, 11(1):41–70, 1997. 105

[mca] The Multicore Association. http://www.multicore-association.org/home.php. 32

[MF86] D.I. Moldovan and J.A.B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *Computers, IEEE Transactions on*, C-35(1):1–12, Jan. 1986. 71

[M.P10] M.Pelcat. *Rapid Prototyping and Dataflow-Based Code Generation for the 3GPP LTE eNodeB Physical Layer mapped onto Multi-Core DSPs.* PhD thesis, Institut National des Sciences Appliqués de Rennes, sept 2010. 96, 116

[opea] OpenCL. http://www.khronos.org/opencl/. 32

[opeb] OpenMP. http://openmp.org/wp/. 32

[PAN08] M. Pelcat, S. Aridhi, and J. F. Nezan. Optimization of automatically generated multi-core code for the LTE RACH-PD algorithm. *0811.0582*, November 2008. DASIP 2008, Bruxelles : Belgium. 30

[PBL95] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee. A hierarchical multiprocessor scheduling system for DSP applications. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, pages 122–126 vol.1, Pacific Grove, California, November 1995. 47, 83, 84

[PBPR09] Jonathan Piat, Shuvra S. Bhattacharyya, Maxime Pelcat, and Mickaël Raulet. Multi-Core Code Generation From Interface Based Hierarchy. In *Conference on Design and Architectures for Signal and Image Processing (DASIP) 2009 Conference on Design and Architectures for Signal and Image Processing (DASIP) 2009*, page online, Sophia Antipolis France, 12 2009. 18, 33, 96

[PBR09] J. Piat, S. S. Bhattacharyya, and M. Raulet. Interface-based hierarchy for Synchronous Data-Flow Graphs. *in Signal Processing Systems (SiPS)*, 2009. 18, 57, 101, 103

[PBR10] J Piat, S.S Bhattacharyya, and M Raulet. Loop transformations for interface-based hierarchies in sdf graphs. In *ASAP 2010, 21st IEEE International Conference on Application-specific Systems, Architectures and Processors*, july 2010. 21, 121

[PBRP09] J. Piat, S. S. Bhattacharyya, M. Raulet, and M. Pelcat. Multi-core code generation from Interface based hierarchy. *in DASIP (DASIP)*, 2009. 30

[PC89] J. K. Peir and R. Cytron. Minimum distance: a method for partitioning recurrences for multiprocessors. *IEEE Transactions on Computers*, 38(8):1203–1211, August 1989. 70

[PMAN09] M. Pelcat, P. Menuet, S. Aridhi, and J.-F. Nezan. Scalable compile-time scheduler for multi-core architectures. *DATE 2009*, 2009. 30

[pol] PolyCore Software Poly-Mapper tool. http://www.polycoresoftware.com/products3.php. 32

[Pol88] C. D. Polychronopoulos. Compiler Optimization for enhancing Parallelism and their Impact on Architecture Design. *IEEE Transactions on Computers*, 37(8):991–1004, August 1988. 70

[Pri91] H. W. Printz. *Automatic mapping of large signal processing systems to a parallel machine*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991. 46

[PRP$^+$08] J. Piat, M. Raulet, M. Pelcat, P. Mu, and O. Déforges. An extensible framework for fast prototyping of multiprocessor dataflow applications. In *IDT08: Proceedings of the 3rd International Design and Test Workshop*, Monastir, Tunisia, december 2008. 28, 60

[Rau06] M. Raulet. *Optimisations Mémoire dans la méthodologie «Adéquation Algorithme Architecture» pour Code Embarqué sur Architectures Parallèles*. PhD thesis, Institut National des Sciences Appliqués de Rennes, 2006. 106

[RPLM08] Mickaël Raulet, Jonathan Piat, Christophe Lucarz, and Marco Mattavelli. Validation of bitstream syntax and synthesis of parsers in the MPEG Reconfigurable Video Coding framework. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 293 – 298, Washinghton États-Unis, 2008. 29

[SGB06] S. Stuijk, M.C.W. Geilen, and T. Basten. SDF^3: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*, pages 276–278. IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006. 32, 55

[Sin07] Oliver Sinnen. *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2007. 61

[SL90] G.C. Sih and E.A. Lee. Dynamic-level scheduling for heterogeneous processor networks. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pages 42–49, October 1990. 46

[SOIH97] W. Sung, M. Oh, C. Im, and S. Ha. Demonstration Of Codesign Workflow In PeaCE. In *in Proc. of International Conference of VLSI Circuit, Seoul, Korea*, 1997. 28

[SPI08] SPIRIT Schema Working Group. IP-XACT v1.4: A specification for XML meta-data and tool interfaces. Technical report, The SPIRIT Consortium, March 2008. 57

[SS05] Oliver Sinnen and Leonel A. Sousa. Communication contention in task scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 16(6):503–515, 2005. 61

[TGS$^+$08] BD Theelen, MCW Geilen, S. Stuijk, SV Gheorghita, T. Basten, JPM Voeten, and AH Ghamarian. Scenario-aware dataflow. 2008. 32

[The07] B.D. Theelen. A performance analysis tool for scenario-aware streaming applications. *Quantitative Evaluation of Systems, International Conference on*, 0:269–270, 2007. 55

[TKG$^+$01] William Thies, Michal Karczmarek, Michael Gordon, David Z. Maze, Jeremy Wong, Henry Hoffman, Matthew Brown, and Saman Amarasinghe. Streamit: A compiler for streaming applications. Technical Report MIT/LCS Technical Memo LCS-TM-622, Massachusetts Institute of Technology, Cambridge, MA, Dec 2001. 56

[TKJM$^+$07] Joseph Thomas-Kerr, Jorn Janneck, Marco Mattavelli, Ian Burnett, and Christian Ritz. Reconfigurable Media Coding: Self-Describing Multimedia Bitstreams. In *SIPS 2007*. IEEE, 2007. 35

[WG90] M.Y. Wu and D.D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, 1990. 62

[WL91] M.E. Wolf and M.S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, pages 452–471, 1991. 69

[Wol90] M.J. Wolfe. *Optimizing supercompilers for supercomputers*. MIT Press Cambridge, MA, USA, 1990. 70

Résumé

Face au défi que représente la programmation des architectures multi-cœurs/processeurs, il est devenu nécessaire de proposer aux développeurs des outils adaptés permettant d'abstraire les notions inhérentes au parallélisme et facilitant le portage d'une application sur différentes architectures. La méthodologie AAA (Adéquation Algorithme Architecture) propose au développeur d'automatiser les étapes de partitionnement, ordonnancement à partir d'une description haut niveau de l'application et de l'architecture. Cette méthodologie permet donc le prototypage rapide d'une application sur différentes architectures avec un minimum d'effort et un résultat approchant l'optimal. Les apports de cette thèse se situent à la fois au niveau du modèle de spécification et de ses optimisations relatives au contexte des architectures parallèles.

Le modèle flux de données répond aux problèmes de modélisation des applications fortement synchronisées par les données. Le sous-ensemble SDF (Synchronous Data Flow), limite l'expressivité du modèle mais apporte un complément d'information permettant une optimisation efficace et garantissant l'intégrité du calcul dans tous les contextes. Les travaux développés dans ce mémoire introduisent un nouveau modèle de hiérarchie dans SDF afin d'améliorer l'expressivité tout en préservant les propriétés du modèle initial. Ce modèle basé sur des interfaces, permet une approche plus naturelle pour le développeur accoutumé au langage C.

Ce nouveau modèle apportant un complément d'information, nous proposons également un ensemble de traitement améliorant la prise en charge des motifs de répétition imbriqués. En effet le modèle de hiérarchie introduit en première partie permet la spécification de motifs dit de « nids de boucles » pouvant masquer le parallélisme potentiel. Il est donc nécessaire d'associer au modèle des traitements permettant de révéler ce parallélisme tout en préservant l'aspect factorisé du calcul. Les méthodes présentées sont adaptées du contexte des compilateurs pour supercalculateurs et de l'univers des réseaux systoliques.

Abstract

Since applications such as video coding/decoding or digital communications with advanced features are becoming more complex, the need for computational power is rapidly increasing. In order to satisfy software requirements, the use of parallel architecture is a common answer. To reduce the software development effort for such architectures, it is necessary to provide the programmer with efficient tools capable of automatically solving communications and software partitioning/scheduling concerns. The Algorithm Architecture Matching methodology helps the programmer by providing automatic transformation, partitioning and scheduling of an application for a given architecture This methodology relies on an application model that allows to extract the available parallelism. The contributions of this thesis tackle both the problem of the model and the associated optimization for parallelism extraction.

The Data flow model is indeed a natural representation for data-oriented applications since it represents data dependencies between the operations allowing to extract parallelism. In this model, the application is described as a graph in which nodes represent computations and edges carry the stream of data-tokens between operations. A restricted version of data-flow, termed synchronous data-flow (SDF), offers strong compile-time predictability properties, but has limited expressive power. In this thesis we propose a new type of hierarchy based on interfaces (Interface-based SDF) allowing more expressiveness while maintaining its predictability. This interface-based hierarchy gives the application designer more flexibility to apply iterative design approaches, and to make optimizing choices at the design level. This type of hierarchy is also closer to the host language semantics such as C because hierarchy levels can be interpreted as code closures (i.e., semantic boundaries), and allow one to design iterative patterns.

One of the main problems with this hierarchical SDF model is the lack of trade-off between parallelism and network clustering. In this thesis we present a systematic method for applying an important class of loop transformation techniques in the context of interface-based SDF semantics. The resulting approach provides novel capabilities for integrating parallelism extraction properties of the targeted loop transformations with the useful modeling, analysis, and code reuse properties provided by SDF.

N° d'ordre : D10-13
Institut National des Sciences Appliquées de Rennes
20, Avenue des Buttes de Coësmes . CS 14315 . F-35043 Rennes Cedex
Tel : 02 23 23 82 00 – Fax : 02 23 23 83 96