Flexible and Scalable Algorithm/Architecture Platform for MP-SoC Design of High Definition Video Compression Algorithms

M. Bonaciu

▶ To cite this version:

M. Bonaciu. Flexible and Scalable Algorithm/Architecture Platform for MP-SoC Design of High Definition Video Compression Algorithms. Micro et nanotechnologies/Microélectronique. Institut National Polytechnique de Grenoble - INPG, 2006. Français. NNT : . tel-00086779

HAL Id: tel-00086779
https://theses.hal.science/tel-00086779

Submitted on 19 Jul 2006

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

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

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|_|_|_|_|_|_|_|_|_|_|

T H E S E

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Micro et Nano Electronique

préparée au **Laboratoire TIMA** dans le cadre de **l'Ecole Doctorale**
« Electronique, Electrotechnique, Automatique, Télécommunications, Signal »

présentée et soutenue publiquement

par

Marius Petru BONACIU

le 4 Juillet 2006

Titre :

Plateforme flexible pour l'exploitation d'algorithmes et d'architectures en vue de la réalisation d'application vidéo haute définition sur des architectures multiprocesseurs mono puces

Directeur de thèse : Ahmed Amine JERRAYA

Jury :

M. Frédéric PÉTROT , Président
M. Eugenio VILLAR , Rapporteur
M. Christophe WOLINSKI , Rapporteur
M. Ahmed Amine JERRAYA , Directeur de thèse
M. Omar HAMMAMI , Examinateur
M. Peter NOWOTTNICK , Examinateur

Every impossible thing will be solved by somebody who is not aware that that thing is impossible

For my family

Thanks

I want to thank Mr. Ahmed Amine JERRAYA, research director at CNRS and the leader of the SLS group from TIMA Laboratory, for directing my thesis. I want to thank him for helping me to discover the world of chips design, and to increase my knowledge in software design and video design. Moreover, I thank him for the time he spent providing me useful advices, either professional or personal, for supporting me in the difficult moments, and for trusting in my capabilities and me, since the beginning of our collaboration.

I thank Mr. Bernard COURTOIS, research director at CNRS and the director of the TIMA Laboratory, for providing me the possibility to do my thesis in this laboratory.

I want to thank Mr. Frédéric PÉTROT for accepting to be the president of the jury, with the occasion of my defense. Additionally, I thank him for all the interesting technical discussions that we had during my thesis.

I want to thank Mr. Eugenio VILLAR, Mr. Christophe WOLINSKI, Mr. Omar HAMMAMI and Mr. Peter NOWOTTNICK for accepting to be part of the jury for my defense. I want to thank all of them for their advices and remarks, regarding the thesis report, and those posed during the defense.

Big thanks go to Nacer-Eddine ZERGAINOH and Lorena ANGHEL, for their help during my first days at TIMA, and all their advices and directions that they provided during my entire thesis.

I want to specially thank Sonja AMADOU, for all her effort and time spent to correct my thesis report, specially the French part. I want to thank for her moral support, and all her advices. Not at last, I thank her for the hours of joking and laughing about all sorts of things. And yes, beer beats the wine.

Milioane de multumiri merg catre colegii mei romani din TIMA, pe care imi permit sa-i consider prieteni: Kati (zana zambareata), Cosmin (geniul nano-tevilor), Iuli (printesa kung-fu), Florin (bucale), Florin (olteanu), Bogdan (Star-Craft), Adi (vijelie), Claudia and Marius (casa de piatra si bafta in continuare), Vladimir (baga tare ardelene), Adina and Iulia (printesele din epilog). Am lasat-o la final pe Gabriela, careia vreau sa-i multumesc pentru tot suportul ei.

I want to thank all my colleagues from the SLS group: Abdel, Adriano, Aimen, Alex, Amer, Arif, Arnaud, Benaoumeur, Fred, Ferid, Giedrius, Hao, Ivan, Lobna, Lorenzo, Patrice, Pierre, Sang-Il, Sami, Wander, Wassim, Xi, Xavier, Yanick, Youngchul, Youssef and many others (I apologize if I forgot somebody). We have shared many months or years of our lives under the same scope, to become smarter and better persons. I will never forget you.

Many thanks to my colleagues from the other groups: Achraf, Ahcene, Amel, Fredi, Guillaume, Kamel, Luis, Manu, Sophie, and soooooo many others. We have shared a lot of coffee, many parties and trips, moments that have spiced up these 3 years of thesis. Thank you.

In final, vreau sa multumesc familiei mele. Fara voi, nu as fi reusit. Gandul ca sunteti alaturi de mine imi da putere sa trec peste momentele grele, si fac momentele frumoase si mai frumoase. Va iubesc, si am sa va iubesc mereu. Va multumesc din tot sufletul.

Plateforme flexible pour l'exploitation d'algorithmes et d'architectures en vue de la réalisation d'application vidéo haute définition sur des architectures multiprocesseurs mono puces

Sommaire

1 Introduction.. 1

2 L'algorithme d'encodage vidéo MPEG4 7

3 Exploration d'algorithme et d'architecture á un haut niveau d'abstraction 11

4 L'implémentation d'architecture pour l'encodeur MPEG4 15

5 Conclusions 19

1 Introduction

Ces dernières années, la complexité des puces a augmenté exponentiellement. La possibilité d'intégrer plusieurs processeurs sur la même puce représente un gain important, et a mène au concept du système multiprocesseur hétérogène sur puce (MP-SoC). Cet aspect a permis d'amplifier de manière significative la puissance de calcule fourni par ce type de puce. Il est même devenu possible d'intégrer des applications complexes sur une seule puce, applications qui nécessitent beaucoup de calculs, de communications et de mémoires.

Dans cette catégorie, on peut trouver les applications de traitement vidéo, comme la famille MPEG. En autre, ces algorithmes ont connu une évolution continue. Si les premières applications (ex. MJPEG, MPEG1) contenaient des algorithmes relativement simples, les nouvelles applications (ex. MPEG4, H264) contiennent des algorithmes complexes. Le travail présenté dans ce document est concentré sur l'algorithme de l'encodage vidéo MPEG4.

Des architectures MP-SoC complexes doivent être mises en application, afin d'assurer les demandes de fonctionnalité de l'encodeur vidéo MPEG4 (ex. codage en temps réel). Par la nature de l'algorithme, la fonctionnalité peut être dynamique (ex. la quantité de calculs exigée dépend de la nature de la vidéo d'entrée) et dépendante des paramètres/besoins de l'application (ex. résolution vidéo). De plus, en fonction du domaine d'application visé (ex. portables, home-cinema), différentes restrictions algorithmique et architecturales sont imposées. Le concepteur est donc confronté à la tâche difficile de trouver et de mettre en application la bonne solution, choisissant dans un grand espace de solutions possible. Aussi, l'implémentation d'algorithme et d'architecture est un processus long et difficile. Tous ces aspects compliquent le processus de conception et de mise en application de l'encodeur vidéo MPEG4 sur une architecture MP-SoC, ce qui augmente le délai de mise sur le marché et diminue considérablement la qualité des résultats.

Afin d'obtenir un encodeur vidéo MPEG4 sur une architecture MP-SoC, il y a 3 défis majeurs que le concepteur doit relever :

1) l’implémentation de l'algorithme de l'encodeur vidéo MPEG4

En raison de la grande complexité de l'algorithme de l'encodeur MPEG4, une quantité significative de code doit être écrit. L'application finale de l'encodeur MPEG4 pourrait être choisie en fonction des différents domaines d'applications (ex. portables) ou de configurations (ex. résolution vidéo), exigeant tous une fonctionnalité d'algorithme différente. Par ailleurs, en mettant l’algorithme d’encodage MPEG4 dans une architecture multiprocesseurs, le besoin de différentes fonctionnalités parallèles et «pipelines» pourrait s’imposer. Implémenter différents algorithmes d’encodeur MPEG4 pour chacun de ces cas représente un effort considérable an temps et en main d’oeuvre.

2) trouver les configurations correctes d'algorithme et d'architecture

Afin d'obtenir une architecture MP-SoC efficace contenant l'algorithme de l'encodeur MPEG4, le concepteur doit trouver et utiliser les bonnes configurations d'algorithme et d'architecture. Pur les deux, il existe un grand nombre de paramètres/configurations à partir desquels le concepteur doit choisir les meilleurs. De, les configurations d'algorithme sont dépendantes des demandes du client, mais également des configurations choisies pour l'architecture. La même situation sont pour les configurations d'architecture, dont certaines dépendent des demandes du client", mais aussi des configurations choisies pour l'algorithme.

Comme simple exemple, il pourrait être nécessaire d’utiliser en fonction de la résolution vidéo (paramètre d’algorithme), un nombre et un type spécifiques de processeurs fonctionnant en parallèle (paramètres d'architecture) pour assurer la puissance de calcul requise. Ceci impose l’adaptation de l'algorithme pour ce niveau de parallélisme (paramètre d'algorithme), qui pourrait augmenter les demandes de trafic pour l’architecture de communication (paramètre d'architecture). Si ce n'est pas possible, le concepteur pourrait être amené à diminuer la qualité d'image codée (paramètre d'algorithme). Cette modification réduira la puissance de calcul requise, menant probablement au besoin de réduire le nombre de processeurs (paramètres d'architecture), ce qui pourrait exiger de réadapter l'algorithme pour un autre niveau de parallélisme (paramètre d'algorithme), et ainsi de suite. Il est à noter que, les bons paramètres d'algorithme sont directement liés aux paramètres d'architecture, et vice-versa. C’est pourquoi, toutes les explorations devraient se concentrer sur ces deux

aspects (algorithme et architecture) à la fois. Prise en considération d'un seul de ces aspects n'est pas toujours suffisante.

Ce procédé d'exploration, qui doit être répété plusieurs fois, prend beaucoup de temps. Ceci est dû au fait que :

a) Le concepteur pourrait avoir à implémenter manuellement et re-implémenter plusieurs fois les modèles d'algorithme et d'architecture, en utilisant des configurations spécifiques d'un algorithme/architecture choisies dans un grand espace de solutions, et vérifier chaque fois si les résultats obtenus arrivent à fournir les résultats d'exécutions requis. Ceci représente un travail difficile et répétitif, et prend beaucoup des temps.

b) En grande partie, la simulation des modèles d'algorithme et d'architecture utilisés dépend du niveau d'abstraction auquel les modèles d'algorithme et d'architecture sont décrits. Au niveau RTL (Register Transfer Level), le temps de simulation est très long.

c) La précision des résultats de simulation dépend également du niveau d'abstraction utilisé. Cependant, elle représente une aspect critique quelque soit le niveau d'abstraction utilisé, pour rassurer le concepteur au sujet de la qualité de ses résultats et mesures.

<u>3) implémentation de l'architecture RTL qui contient l'application de l'encodeur MPEG4</u>

Implémenter manuellement l'architecture MP-SoC pour l'encodeur MPEG4, jusqu'au niveau RTL, pourrait nécessiter plusieurs mois de temps de travail. En raison de la grande quantité de détails architecturaux de bas niveau qui doit être prise en considération (ex. les interfaces, les signaux, les protocoles, des synchronisations, décodage d'adresses, les arbitres, etc.), et du fait que cette architecture doit "servir" de façon pertinente à la bonne fonctionnalité de l'algorithme. Par exemple, si *Tâche1* doit envoyer des données à *Tâche2*, l'architecture doit faire de telle sorte que ceci va fonctionner correctement. A tout ce temps de travail est ajouté le temps nécessaire pour simuler, valider et corriger l'architecture obtenue, processus qui pourrait doubler le temps pour obtenir l'architecture finale. Comme il

sera montré dans ce document, le temps total a pris presque 6 mois pour notre expérimentation. Ainsi, dans le cas où l'architecture et l'algorithme auraient été implémenté à partir d'un mauvais choix de configurations, il est possible que le processus entier de développement doive être recommencé afin d'utiliser une autre configuration. Ceci augmentera le délai de mise sur le marché, et en réalité il pourrait représenter l'échec du projet.

En outre, le concepteur pourrait devoir implémenter l'application de l'encodeur MPEG4 sur différents types d'architectures. Par exemple, il doit implémenter l'algorithme de l'encodeur MPEG4 plus l'architecture MP-SoC entière, ou il doit juste implémenter l'encodeur MPEG4 sur une architecture MP-SoC déjà existante. Le concepteur devra donc se familiariser avec plusieurs flots de conception spécifiques à chacun de ces cas.

Les contributions présentées dans ce document sont 3 solutions qui pourraient aider à réduire ces 3 problèmes :

1) Encodeur MPEG4 Parallèle

Puisque l'encodeur MPEG4 est une application qui nécessite une grande quantité de calculs, l'intégration sur des architectures parallèle/pipeline pourrait être nécessaire. Nous devons être capables d'adapter facilement l'algorithme en fonctions des différentes fonctionnalités parallèle/pipeline. De plus, pour chacune de ces architectures, l'algorithme devrait être facilement adaptable pour différents paramètres algorithmiques (ex. résolution vidéo, qualité, précision d'évaluation de mouvement, bitrates, etc.).

Ce document présente un encodeur MPEG4 flexible, qui peut être facilement adapté pour différents types de paramètres d'algorithme, mais également différents niveaux de parallélisme/pipeline.

2) Exploration à haut niveau d'algorithme et d'architecture pour l'encodeur MPEG4 avec des paramètres taillés sur mesure

Pour trouver les configurations optimales d'algorithme et d'architecture, le concepteur doit être capable d'examiner et d'explorer rapidement différentes configurations. Ceci est possible si :

a) Le concepteur peut automatiquement obtenir les modèles d'algorithme et d'architecture nécessaires à l'exploration de l'espace des solutions.

b) Le temps nécessaire pour faire les mesures de performance est diminué en faisant l'exploration d'architecture à un haut niveau d'abstraction, et pas au niveau RTL, en ignorant beaucoup de détails d'architecture de bas niveau. Ainsi, en ne prenant pas en compte tous les détails, les simulations deviennent plus rapides.

c) La précision d'estimation est assurée en estimant les temps de calculs et les temps de communications.

Ce document propose une méthode d'exploration d'algorithme et d'architecture à haut niveau pour l'encodeur MPEG4 avec des paramètres taillés sur mesure, qui pressentent les avantages décrits précédemment. En utilisant cette approche, plusieurs configurations d'algorithme et d'architectures peuvent être explorées plus vite qu'avec à une exploration faite au niveau RTL. Aussi, à un haut niveau d'abstraction, en ignorant beaucoup de détails de bas niveau, la vitesse de simulation, validation et correction est sensiblement augmentée.

3) Flot commun utilisé pour l'implémentation rapide de l'encodeur MPEG4 sur différents types d'architectures

Quand l'encodeur MPEG4 doit être mis en application dans un MP-SoC, l'architecture visée peut être complètement nouvelle, ou déjà existante (habituellement une architecture basée sur des processeurs). Utiliser différents flots d'implémentations pour ces deux cas pourrait se révéler inefficace. Ceci est dû principalement par le besoin de transférer les résultats après chaque étape de développement entre des environnements multiples (outils), l'incompatibilité entre les outils et les spécifications, les limitations d'outils, etc. L'obtention du résultat final pourrait demander beaucoup de temps, et pourrait générer une quantité significative d'erreurs.

Ce document propose un flot commun pour l'implémentation de l'encodeur MPEG4 sur MP - SoC pour différents types d'architectures. De plus, parce que plusieurs étapes de ce flot sont automatisées, il aide à obtenir les résultats finaux en peu de temps. En utilisant

cette approche, plusieurs architectures MP-SoC complètes au niveau RTL ont été obtenues. En outre, en utilisant le même flot, l'encodeur MPEG4 a également été implémenté sur une architecture quadri-processeurs déjà existante.

2 L’algorithme d’encodage vidéo MPEG4

La norme de l'encodeur MPEG4 a été développée par le Motion Picture Experts Group autour de 1994. Depuis, beaucoup de variantes de MPEG4 ont été réalisées, dont les plus populaires sont OpenDivX, DivX et XviD. Entre chacune de ces réalisations, il y a quelques différences en ce qui concerne les précisions de calculs et la qualité d’image compressée. Cette variation peut exister, parce que les spécifications MPEG4 ne limite pas de façon stricte les calculs de toutes les étapes dans l’algorithme. Cette souplesse est une des nombreuses raisons pour les quelles le MPEG4 est devenu si populaire.

Une autre raison pour laquelle le MPEG4 est populaire est sa capacité de compression. Les expériences pratiques ont prouvé que tandis que l'encodeur vidéo MPEG2 (utilisé dans le DVD) compresse un film de 2 heures dans 4.7Gbytes, le MPEG4 est capable de compresser le même film sur un CD-ROM de 700Mbytes. Il est vrai qu’avec le MPEG2, la qualité d’image obtenue est meilleure qu’avec le MPEG4, mai la qualité d’image obtenu est acceptable. Par contre, l’inconvénient de l'encodeur MPEG4 est sa complexité algorithmique. Comparant l'algorithme du MPEG4 à celui du MPEG2, la complexité du MPEG4 est environ 3 fois plus grande que le MPEG2. Mais, cet inconvénient ne nuit pas au succès du MPEG4.

Le principe sous-jacent est de comprimer seulement les différences spatio-temporelles entre les images consécutives. Ainsi, au lieu de sauvegarder une image complète, l'algorithme sauvegarde seulement les différences entre cette image et l’image précédente. Ces différences sont déterminées en utilisant des algorithmes complexes, et le résultat de l'algorithme est un film fortement comprimé, connu sous le nom de MPEG4 bitstream. Même le facteur de compression peut être modifié basé sur les préférences des utilisateurs, en augmentant ou en diminuant la qualité d'images stockées dans le bitstream.

Un diagramme simplifié de l'algorithme d'encodage MPEG4 est présenté dans la Figure 1. Le but de chaque fonction sera détaillé plus tard dans ce document. Ces fonctions peuvent être distribués en 2 catégories : les fonctions de traitement d'image (groupées dans la tâche *MainDivX*), et les fonctions de compression (groupées dans la tâche *VLC*). La tâche

MainDivX reçoit l'image courante non compressée, et détermine les différences spatio-temporelles entre cette image et l'image précédente de la vidéo. Ces résultats (qui ne sont pas encore compressée) sont envoyés à la tâche *VLC*, qui les comprime, pour obtenir à la fin le MPEG4 bitstream final. La taille du bitstream MPEG4 est habituellement beaucoup plus petite que la taille d'image originale en entrée.

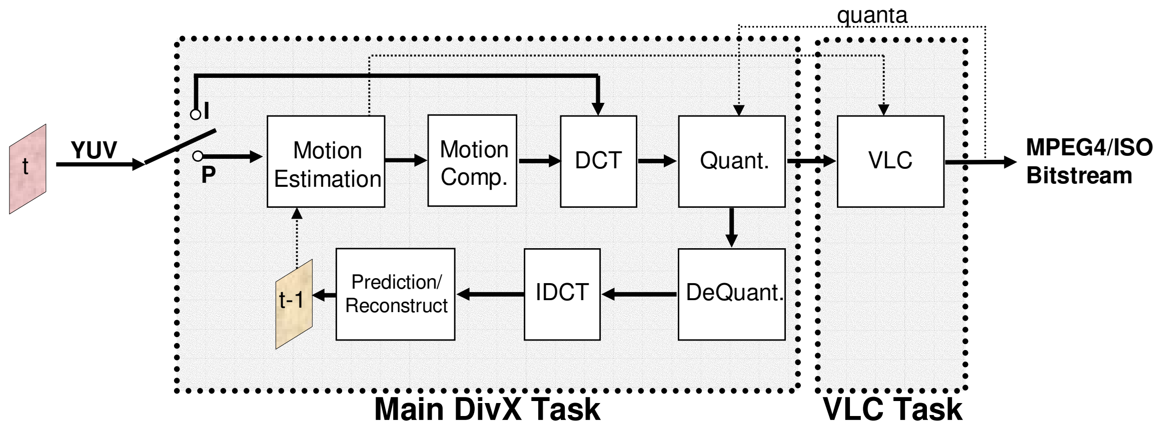


Figure 1. Diagramme de l'algorithme d'encodage MPEG4

Par sa nature, l'algorithme d'encodeur MPEG4 est un algorithme séquentiel. Cependant, le comportement de chacune de ses fonctions internes dépend fortement de vidéo à l'entrée et des paramètres de l'algorithme utilisé. Dans le cas où la vidéo d'entrée contient des images et des séquences complexes (ex. quelqu'un qui court dans la forêt), l'algorithme exécutera plus de calculs. Dans le cas où la vidéo d'entrée contient des images et des séquences simples (ex. une interview devant un mur blanc), l'algorithme exécutera moins de calculs. En réalité, le concepteur ne devrait pas faire ses implémentations basées sur un seul type de vidéo d'entrée. L'algorithme MPEG4 devrait être capable de supporter n'importe quels types de vidéo, complexe ou simple.

Il y a beaucoup de paramètres d'algorithme, proposés par les spécifications ISO de MPEG4, avec lesquelles le concepteur a la possibilité d'ajuster les comportement/résultats de l'algorithme. Ces paramètres sont : résolution, frame_rate, bitrate, précision de l'estimation de mouvement, surface de recherche de mouvement, images Progressives/Entrelacées, détection de changement de scène, intervalle de quantification, type de quantification, rate_delay, etc. Chacun de ces paramètres a un impact, plus ou moins im-

portant, sur les résultats et le comportement d'algorithme. De plus, en ajustant un de ces paramètres, aucune modification de code de l'algorithme ne devrait être exigée.

Cependant, quand l'algorithme est destiné pour supporter une fonctionnalité parallèle/pipeline (habituellement requise pour des architectures multiprocesseur), le code séquentiel de l'algorithme doit être réécrit pour supporter ce parallélisme/pipeline. En changeant le niveau de parallélisme/pipeline, il y a un risque que le code doive encore être réécrit pour cette nouvelle exigence. Pour éviter cela, nous avons adapté l'algorithme original MPEG4 séquentiel pour supporter différents niveaux de parallélisme/pipeline en rajoutant un paramètre additionnel qui contient le niveau désiré du parallélisme. En modifiant ce paramètre, le code de l'algorithme ne change pas, seulement son comportement.

Ceci a été possible en découpant les images d'entrée dans des sub-images multiples (secteurs), et en exécutant plusieurs instances de la même tâche *MainDivX* et *VLC* pour ces secteurs (Figure 2). De plus, pour chacun de ces instances, un comportement pipeline a été ajouté. L'image est fournie par une tâche de test *Video*, image qui sera découpée en plusieurs secteurs par une tâche *Splitter*. Chacun de ces secteurs est traité par une tâche *MainDivX*. Leurs résultats sont ensuite comprimé par plusieurs tâches *VLC*, chacune d'elles obtenant une "petite partie" du MPEG4. Enfin, la tâche *Combiner* reçoit toutes ces "petites parties", les trie et les enchaîne finalement pour obtenir le MPEG4 bitstream final, correspondant à l'image courante. Après, ce bitstream est envoyé à une autre tâche de test, appelée *Storage*, qui simule le comportement d'un support de stockage (ex. HDD).

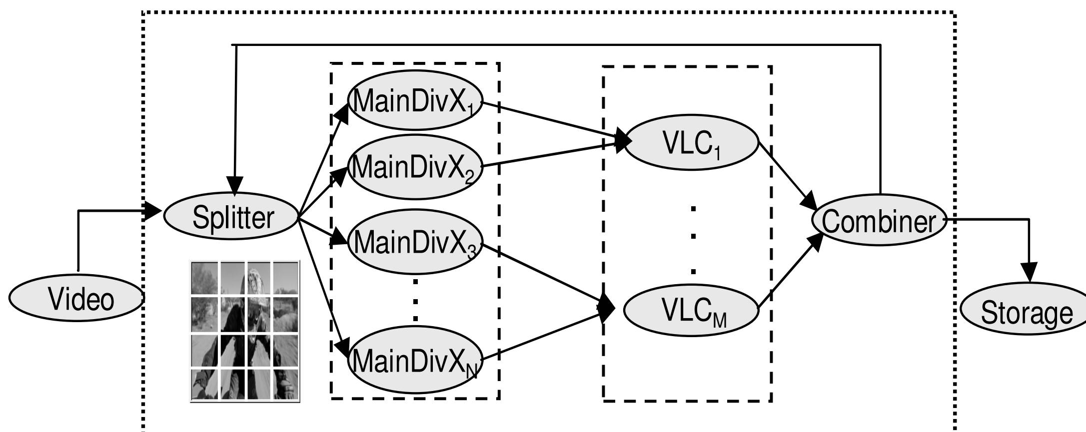


Figure 2. Structure général d'algorithme MPEG4 parallélise/pipeline

Ainsi, l'algorithme se comporte comme plusieurs SMP (MainDivX-SMP et VLC-SMP). Cependant, il est à noter que, quand l'image est divisée en un seul secteur, l'algorithme se comportera comme l'algorithme séquentiel, ce qui est utile lorsque le concepteur ne veut pas utiliser de parallélisme dans l'algorithme.

Comme résultat, nous avons obtenu un algorithme MPEG4 flexible, qui supporte tous les paramètres imposés par les spécifications ISO de MPEG4, en plus de nos paramètres additionnels utilisés pour ajuster le comportement de l'algorithme pour différents niveaux de parallélisme/pipeline. Ainsi, l'algorithme peut être facilement adapté et utilisé pour l'exploration de différentes architectures MP-SoC contenant différents nombres de processeurs.

3 Exploration d'algorithme et d'architecture á un haut niveau d'abstraction

Implémenter une architecture MP-SoC jusqu'au niveau RTL (ce qui nécessite temps de conception très long), à partir d'un ensemble de paramètres mal choisis (ex. nombres/type de processeurs, ou topologie de communication), pourrait être onéreux. N'importe quelle modification des paramètres utilisés pourrait nécessiter des modifications complexes d'algorithme et d'architecture, ou dans le pire des cas l'obligation de re-implémenter complètement l'algorithme et l'architecture. Peu de produits peuvent justifier un tel budget de conception, et une solution pour augmenter les performances est de faire l'exploration d'algorithme et d'architecture à un haut niveau d'abstraction, avant l'étape d'implémentation de l'architecture RTL. La Figure 3 présente notre flot d'exploration d'algorithme et d'architecture à un haut niveau d'abstraction.

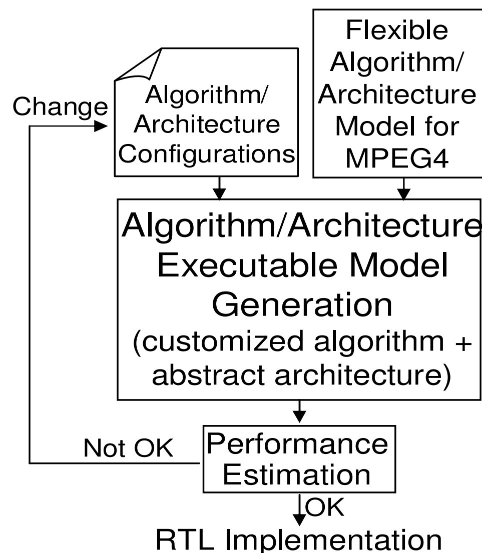


Figure 3. Exploration d'algorithme et d'architecture a un haut niveau d'abstraction

Les entrées de ce flot sont un ensemble de paramètres de *Configurations d'Algorithme/Architecture* (que le concepteur veut explorer) et un *Modèle Flexible d'Algorithme/Architecture pour l'Encodeur MPEG4*. En utilisant ce modèle et les configurations d'algorithme/architecture, différents *Modèles Exécutable Taillés sur Mesure d'Algorithme/Architecture* peuvent être obtenus. Chaque fois que les *Configurations d'Algorithme/Architecture* sont changées, un nouveau *Modèle Exécutable Taillé sur Mesure*

d'Algorithme/Architecture peut être obtenu. Enfin, ce modèle est utilisé pour des évaluations de performances. Si les résultats obtenus ne satisfont pas aux exigences du concepteur, il doit réadapter les *Configurations d'Algorithme/Architecture*, et régénérer un nouveau *Modèle Exécutable Taillé sur Mesure d'Algorithme/Architecture*. Cette itération sera répètee jusqu'à ce que le concepteur trouve une solution de *Configurations d'Algorithme/Architecture* pour laquelle les performances estimées répondent aux exigences. Plus tard, ces configurations seront utilisées pour l'implémentation d'architecture RTL.

Les *Configurations d'Algorithme/Architecture* (Figure 4) contient un ensemble de paramètres distribués en 2 catégories : Paramètres d'algorithme et Paramètres d'architecture.

Paramètres d'algorithme	**Paramètres d'architecture**
Niveau de Parallélisme/Pipeline	Nombre de processeurs
Résolution vidéo	Type de processeurs
Frame_rate	Partitionnement HW/SW
Bitrate	Topologie de communication
Key_frame	Type d'arbitrage
Précision de l'estimation de mouvement	Taille de messages
Surface de recherche de mouvement	Taille de données
Mode Progressif/Entrelacer	Latence de transfert de données
Détection de changement de scène	Latence d'initialisation de transfert
Intervalle de quantification	Latence de fin de transfert
...	...

Figure 4. Configurations d'algorithme et d'architecture à explorer

Le *Modèle Flexible d'Algorithme/Architecture pour MPEG4* (Figure 5) représente un modèle *template* à haut niveau (décrit en utilisant une langage macro) à partir duquel différents modèles taillés sur mesure peuvent être obtenus (par une macro génération). Ce modèle est constitué de plusieurs modules SystemC, contenant les tâches de l'algorithme MPEG4 flexible présenté avant. Ces tâches communiquent en utilisant un *Modèle d'Exécution d'Interconnections Abstrait*, qui est en charge de la gestion des communications et les synchronisations entre les tâches. Tous les détails de bas niveau (ex. OS, Adaptateurs, RTL signale, etc.) sont complètement abstraits. Plus de détails au sujet de ce modèle sont présentés plus tard dans ce document.

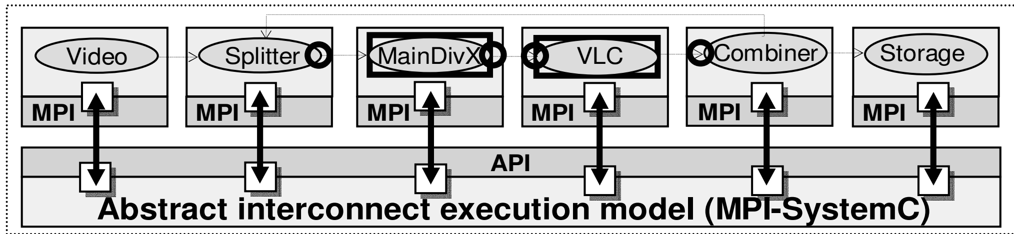


Figure 5. Modèle Flexible Algorithme/Architecture pour l'encodeur MPEG4

En utilisant ce modèle et les configurations d'algorithme/architecture, différents modèles taillés sur mesure peuvent être obtenus. Un exemple obtenu d'un tel modèle est présenté dans la Figure 6. Dans ce cas, un modèle exécutable a été macro-généré contenant 4 tâches *MainDivX* et 2 tâches *VLC*. Le code appartenant à chacune des tâches de l'application a été adapté basé sur les paramètres d'algorithme choisis (ex. résolution vidéo, bitrate), et les aspects architecturaux (ex. types de processeurs, tailles de messages) ont été taillés sur mesure basé sur les paramètres d'architecture. En d'autres termes, ce modèle représente un modèle d'algorithme/architecture fixe et déjà taillé sur mesure en tenant compte des préférences du concepteur.

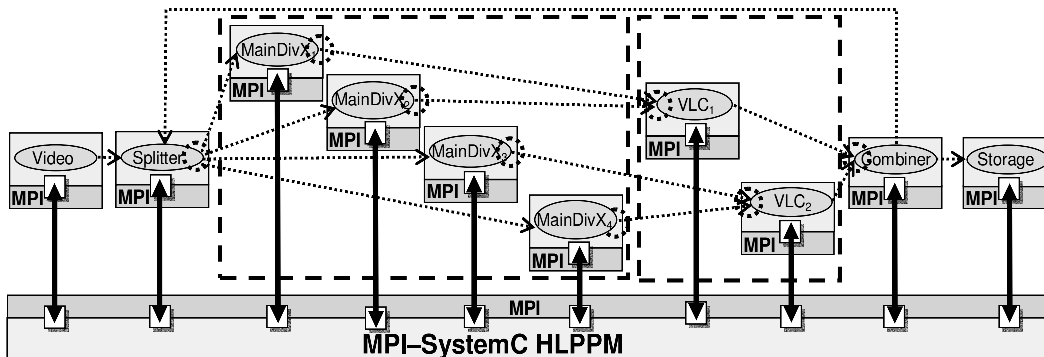


Figure 6. Modèle Exécutable Taillé sur Mesure d'Algorithme/Architecture pour l'Encodeur MPEG4 utilisé pour les évaluations de performances

Afin de faire les évaluations de performances, des annotations de temps seront employées pour les calculs et communications, des temps qui dépendent fortement des configurations d'algorithme/architecture utilisées. Par l'exécution de ce modèle taillé sur mesure et annotée avec le temps d'exécution, une estimation de ses performances peut être déterminée. Si ces performances ne répondent pas aux exigences, le concepteur va réadapter les *Configurations d'Algorithme/Architecture*, obtenant un nouveau *Modèle Taillé sur Mesure*

d'Algorithme/Architecture avec les résultats d'évaluation de performances, de ce nouveau modèle.

En utilisant cette approche, différentes configurations des architectures pour l'encodeur MPEG4 ont été explorées avec succès en peu de temps, pour différentes résolutions vidéo, frame_rates, bitrates, nombre de processeurs, types de processeurs, configurations de communication, etc. Cette exploration s'est avérée sensiblement plus rapide comparée à l'exploration au niveau RTL. Ainsi, les performances estimées à haut niveau se sont avérées assez proches des performances mesurées au niveau RTL, ce qui confirme la possibilité d'utiliser cette approche pour nos expériences. Les points suivants one été réalisés :

a) La nécessité d'explorer un grand espace de solution a été facilitée en générant automatiquement différents Modèles Exécutable Taillés sur Mesure d'Algorithme/Architecture. Ceux-ci peuvent être obtenus à partir d'un *Modèle Flexible d'Algorithme/Architecture pour MPEG4* unique, qui a fourni la possibilité générer automatiquement l'architecture abstraite basée sur un ensemble de *Configurations d'Algorithme/Architecture* choisies par le concepteur.

b) Une simulation rapide est possible en faisant l'exploration d'architecture à un haut niveau d'abstraction. En ignorant beaucoup de détails d'architecture des bas niveaux dans le Modèle Exécutable Taillé sur Mesure d' Algorithme/Architecture, la simulation devient rapid.

c) Pour satisfaire au besoin de résultats précis d'estimations, on utilise l'*Exploration d'Algorithme/Architecture a un Haut Niveau d'Abstraction* qui fournit des résultats d'estimations avec une précision élevée en termes des temps de calculs et des communications, par annotations basés sur les *Configurations d'Algorithme/Architecture*. De plus, l'exploration capture les calculs et les communications fonctionnant ensemble, pour estimer les exécutions du système entier.

4 L'implémentation d'architecture pour l'encodeur MPEG4

Lorsqu'il implémente l'encodeur MPEG4 dans un MP-SoC, le concepteur a deux possibilités :

a) Il peut implémenter l'architecture MP-SoC complète de l'encodeur. Dans ce cas, le concepteur doit implémenter les tâches de l'algorithme MPEG4 ainsi que les détails logiciels de plus bas niveau (i.e. OS) qui assurent la fonctionnalité des tâches de plus haut niveau sur les processeurs. De plus, il doit implémenter l'architecture MP-SoC avec tous les détails matériels de bas niveau (ex. adaptateurs, sous-système de processeurs). Les résultats obtenus doivent ensuite être exécutés pour une validation finale par une approche de co-simulation précise au cycle près.

b) l'encodeur MPEG4 peut être implémenté sur une architecture déjà existante. Dans ce cas, le concepteur doit implémenter seulement les tâches de l'algorithme d'encodage MPEG4 ainsi que les détails logiciels de bas niveau. L'implémentation matérielle n'est plus nécessaire puisque l'architecture existe déjà. Pour la validation finale, les résultats seront exécutés nativement sur l'architecture existante.

Dans le cadre de notre approche, pour ces deux possibilités, nous pouvons utiliser comme entrées les mêmes spécifications d'algorithme et d'architecture. Il s'agit plus exactement de la même *Configuration d'Algorithme/Architecture* obtenue après la phase d'exploration algorithme/architecture de haut niveau présentée précédemment, et du même *Modèle Flexible d'Algorithme/Architecture pour encodeur MPEG4* (Figure 7).

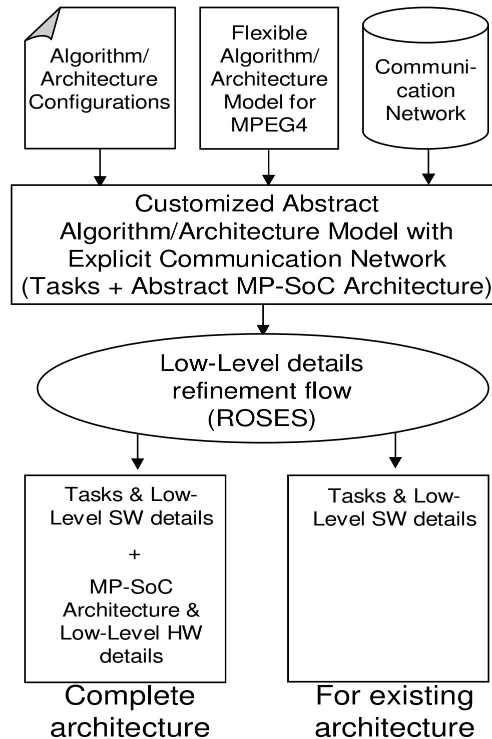


Figure 7. Flot proposé pour l'implémentation de l'encodeur MPEG4 sur différentes architectures

Toute comme dans le cas de l'exploration algorithme/architecture, un modèle taillé sur mesure devra être obtenu. La seule différence est, dans ce cas, que le modèle obtenu utilisera maintenant un réseau de communication explicite en tant qu'infrastructure d'interconnections, un réseau de communication provenant d'une bibliothèque contenant différents types de réseaux de communication. Le modèle obtenu est appelé *Modèle Abstrait Taillé sur Mesure d'Algorithme/Architecture avec Réseau Explicite*. Un exemple de tel modèle est présenté à la Figure 8.

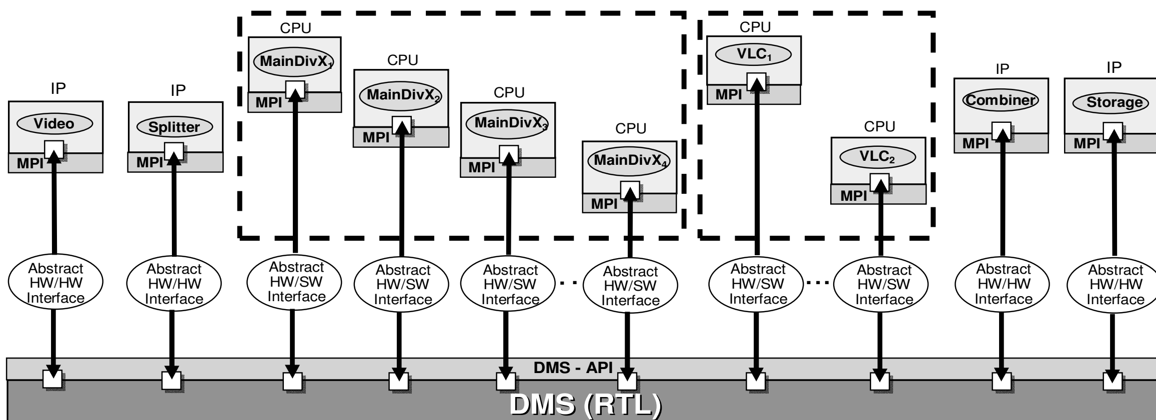


Figure 8. Modèle Abstrait Taillé sur Mesure d'Algorithme/Arch. avec Réseau Explicite

Comme nous pouvons le remarquer, les tâches des applications ont été personnalisées et sont prêtes, ainsi que le réseau de communication. Dans Figure 8, nous avons utilisé le réseau de communication DMS, qui sera décrit en détail plus loin dans ce document. Les seuls objets qui sont encore abstraits sont les interfaces HW/SW et HW/HW qui contiennent les détails d'implémentation de bas niveau du logiciel et du matériel.

Pour obtenir ces détails de bas niveau, nous avons utilisé le flot ROSES développé au laboratoire TIMA par le groupe SLS. Ce flot reçoit comme entrée l'architecture abstraite mentionnée ci-dessus et utilise plusieurs outils internes afin d'obtenir/raffiner et valider les détails de bas niveau logiciels et matériels. Plus de détails au sujet de ce flot seront présentés ultérieurement dans ce document.

Selon l'architecture ciblée, il est possible d'obtenir deux types de résultats :

a) Il est possible d'obtenir une architecture MP-SoC complète au niveau RTL et qui contient l'algorithme d'encodage MPEG4. Dans ce cas, le flot ROSES est utilisé afin d'obtenir tous les détails de bas niveau, HW et SW. Le résultat final, appelé architecture RTL, contient les tâches des l'application avec les détails logiciels de bas niveau (i.e. OS, aussi connu sous le nom d'adaptateurs logiciels), ainsi que la totalité de l'architecture MP-SoC au niveau RTL qui contient le réseau de communication, adaptateurs HW, sous-systèmes CPU (architecture locale du processeurs avec composants auxiliaires RTL). La figure 9 illustre une architecture de MP-SoC obtenue au niveau RTL qui contient l'encodeur MPEG4.

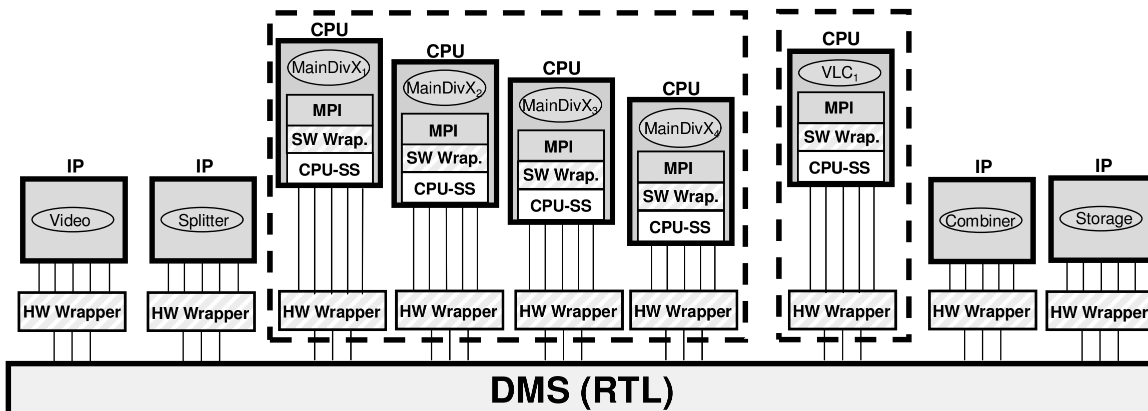


Figure 9. Architecture RTL pour l’encodeur MPEG4

Le modèle complet est un modèle exécutable décrit en SystemC et ses performances peuvent être mesurées en utilisant une co-simulation classique précise au niveau cycle. Le principal désavantage est que la co-simulation est très lente. Selon nos expériences, il a fallu plus de 4 heures pour co-simuler le processus d'encodage d'une seule image du film. L'avantage est qu'il est possible d'analyser le comportement de n'importe quel composant de l'architecture.

b) Si l'encodeur MPEG4 doit être implémenté sur une architecture existante, seules les tâches de l'application et les détails logiciels de bas niveau doivent être obtenus. Lors de nos expériences, l'architecture existante était une plateforme quadri-processeurs, contenant 4 processeurs Sparc. Sur une telle architecture, il est suffisant de faire une compilation croisée des tâches et des détails logiciels de bas niveau et ensuite d'allouer et d'exécuter les fichiers binaires résultants sur la plateforme. Puisque l'application est entièrement exécutée nativement, ceci confère comme avantage que les mesures de vitesse d'exécution et de performance sont élevées (temps réel). Le principal désavantage est la difficulté d'analyser le comportement des composants matériels. Cependant, ceci n'affecte pas l'analyse du logiciel et les post-optimisations du logiciel.

5 Conclusions

L'implémentation d'encodeurs vidéo, un encodeur MPEG4 par exemple, dans des MP-SoC est souvent nécessaire dans de nombreuses applications : télécommunication mobile, home cinéma, vidéo surveillance, etc. Etant donné que chacune de ces applications imposent différents algorithmes, contraintes architecturales et besoins, l'implémentation d'encodeur MPEG4 dans un MP-SoC doit faire face à de nombreux défis.

Ce document a présenté une nouvelle approche qui a été utilisée avec succès pour l'implémentation d'un encodeur MPEG4 dans un MP-SoC avec différentes configurations d'algorithmes et d'architectures. Ceci a été possible grâce à notre solution, parce que:

a) un Algorithme d'Encodeur MPEG4 Flexible a été implémenté avec 2 SMP pour des tâches nécessitant beaucoup de calcul. Cet algorithme peut être facilement paramétré (ex. résolution, frame-rate, bitrate, précision de l'estimation de mouvement, type de quantification, etc.) et aussi parallélisé, en ajustant simplement le niveau de parallélisme dans chacun de SMP. De cette manière, l'algorithme obtenu peut-être facilement configuré pour plusieurs besoins de l'application, mais également pour différentes architectures avec un petit ou un grand nombre de processeurs.

b) une Exploration d'Algorithme/Architecture à un Haut Niveau d'Abstraction avec des paramètres taillés sur mesure a été utilisée afin d'explorer rapidement plusieurs configurations d'algorithmes/architectures. Ceci nous a permis de trouver les bons paramètres satisfaisant aux besoins bien avant l'implémentation de l'architecture RTL. En faisant l'exploration à un niveau élevé, beaucoup de détails de l'architecture de bas niveau ont été complètement abstraits, rendant la simulation d'autant plus rapide. En outre, les modèles de simulation ont été automatiquement obtenus à partir d'un unique *Modèle d'Algorithme/Architecture Flexible pour l'Encodeur MPEG4*, à partir duquel beaucoup de models de simulation à haut niveau peuvent êtres obtenus automatiquement en utilisant une approche de macro génération. L'estimation de performance a été réalisée en insérant des annotations de temps pour les calculs et les communications dans les modèles de simulation résultants. A partir de notre ex-

périence, la précision de cette approche d'exploration de haut niveau s'est montrée suffisamment proche des mesures réelles à bas niveau, ce qui a donc confirmé la possibilité d'utiliser notre approche pour cette application.

c) une approche à base de composants a été utilisée pour l'implémentation de l'encodeur MPEG4 aussi bien sur des architectures MP-SoC complètes, que sur des architecture existantes. Cette approche est basée sur le flot ROSES, développée dans le Groupe SLS du laboratoire TIMA. Le principal objectif de ce flot est de raffiner les détails de bas niveaux (interfaces Matériel/Logiciel) qui étaient abstraits lors de l'exploration algorithme/architecture de haut niveau. En utilisant le flot présenté dans ce document, l'encodeur MPEG4 a été implémenté avec succès dans plusieurs architectures MP-SoC au niveau RTL. La même approche a été utilisée pour l'implémentation de l'encodeur MPEG4 sur une architecture quadri-processeurs existante, pour différentes résolutions, frame-rate, bitrates, etc.

Basé sur l'approche présentée dans ce document, plusieurs autres applications vidéo ont également été implémentées avec succès : encodeur MPEG1, décodeur MPEG1, décodeur MPEG2 et décodeur MPEG4. Ceci a été possible en adaptant le *Modèle d'Algorithme/Architecture Flexible pour MPEG4* pour supporter des paramètres supplémentaires de type d'algorithmes et d'encodeur/décodeur. Il est également prévu d'étendre cette approche au format H.264. Toutefois, l'extension de cette approche vers d'autres domaines d'applications que la vidéo reste un sujet de recherche ouvert pour un travail futur.

Flexible and Scalable Algorithm/Architecture Platform for MP-SoC Design of High Definition Video Compression Algorithms

Marius Petru BONACIU

TIMA Laboratory, SLS Group

Summary

1 Introduction 1

1.1 Introduction about the MP-SoC as a solution for the implementation of video applications on a chip 2

1.2 Motivations 3

1.3 Objectives 6

1.4 State of the art 6

1.5 Contributions 10

1.5.1 Flexible modeling style to describe the algorithm and architecture specifications for MPEG4 encoder 10

1.5.2 High-Level algorithm/architecture exploration for MPEG4 encoder with custom parameters 10

1.5.3 Common flow used for the implementation of MPEG4 encoder on different targeted architectures 11

1.6 Document outline 12

2 MPEG4 video encoder algorithm 13

2.1 Introducing the MPEG4 video encoder 14

2.1.1 MPEG4 video encoder as new solution to MPEG2 video encoder 14

2.1.2 Usual method of using the MPEG4 video encoder 15

2.1.3 Describing the YUV411 video format used as input by the MPEG4 video encoder 15

2.1.4 Presenting the structure of the MPEG4 video encoder algorithm 17

2.1.5 Different types of encoding principles used for the video's frames 18

2.2 Describing the encoding principle for an I frame 20

2.2.1 Introducing the concept of encoding an I frame 20

2.2.2 Representation of the algorithm and functions used for encoding an I frame 20

2.2.3 Describing the Fast Discrete Cosine Transformation (FDCT) 21

2.2.4 Describing the Quantization function 23

2.2.5 Describing the DeQuantization function 24

2.2.6 Describing the Inverse Discrete Cosine Transformation (IDCT) 24

2.2.7 Describing the Intra Prediction function 26

2.3 Describing the encoding principle for a P frame 28

2.3.1 Introducing the concept of encoding the P frame 28

2.3.2 Representation of the algorithm and functions used for encoding a P frame 28

2.3.3 Describing the Motion Estimation function 29

2.3.4 Describing the Motion Compensation function 32

2.3.5 Describing the FDCT, Quantization, DeQuantization and IDCT functions for the P MacroBlock 33

2.3.6 Describing the Reconstruction function 34

2.4 Detailed description of the VLC and Bitrate Controller 35

2.4.1 Description of the compression phase done by the MPEG4 video encoder 35

2.4.2 Applying ZigZag reordering on a MacroBlock 35
2.4.3 Compressing the ZigZag-ed MacroBlock using RLE and Huffman compression 36
2.4.4 Structure of the MPEG4 bitstream 37
2.4.5 Quantization adjustment by Bitrate Controller for the next frame encoding 38

2.5 MPEG4 algorithm parameters 38
2.5.1 Image resolution parameter 39
2.5.2 Frame rate parameter 39
2.5.3 Keyframe rate parameter 40
2.5.4 Compression rate (bitrate) parameter 40
2.5.5 Quantization range parameter 41
2.5.6 Rate control delay parameter 41
2.5.7 Computations precision parameter 42
2.5.8 Scene change detection parameter 42
2.5.9 Compression mode parameter 43
2.5.10 Algorithm type parameter 43

2.6 Conclusions 45

3 Flexible modeling style to represent the Combined Algorithm/ Architecture Model for MPEG4 47

3.1 Parallelism/pipeline support for the MPEG4 video encoder algorithm 48
3.1.1 Amount of computations required by the MPEG4 encoder 48
3.1.2 Objectives 49
3.1.3 State of the art 49
3.1.4 Contributions 52

3.2 Proposed MPEG4 video encoder supporting different parallel/pipeline configurations 53
3.2.1 Presenting the approach used for inserting parallelism and pipeline support into the MPEG video encoder 53
3.2.2 Describing the method used for dividing the input image into multiple smaller areas required for the parallelized MainDivX tasks 57

3.3 Flexible Architecture with 2 SMPs 58
3.3.1 Global view of the Flexible Architecture with 2 SMPs 58
3.3.2 Describing the functionality of the Splitter 59
3.3.3 Describing the functionality of the MainDivX 60
3.3.4 Description of the functionality of the VLC 61
3.3.5 Describing the functionality of the Combiner 62

3.4 Combined Algorithm/Architecture Executable Model 63
3.4.1 Concept of Combined Algorithm/Architecture Executable Model 63
3.4.2 SystemC used for the description of the Combined Algorithm/Architecture Executable Model 66
3.4.3 Combined Algorithm/Architecture Executable Model using MPI-SystemC HLPPM 68

3.5 Flexible Algorithm/Architecture Model for MPEG4 70
3.5.1 Concept of tasks with Flexible Computations 70
3.5.2 Concept of tasks with Flexible Output 73
3.5.3 Concept of tasks with Flexible Input 75
3.5.4 Flexible Algorithm/Architecture Model for MPEG4 76
3.5.5 Algorithm and Architecture configurations 78

3.5.6 Obtaining the Combined Algorithm/Architecture Executable Model 79

3.6 Conclusions 80

4 High-Level Algorithm/Architecture Exploration 82

4.1 Principle of the High-Level Algorithm/Architecture Exploration 83

4.1.1 Introduction 83

4.1.2 Solution space for MPEG4 encoder on MP-SoC 83

4.1.3 State of the art - classical exploration flow 84

4.1.4 Contribution of the proposed High-Level Algorithm/Architecture Exploration 87

4.2 High-level Algorithm/Architecture Exploration flow for MPEG4 88

4.2.1 Obtaining the Timed Executable Model required for performance estimations 90

4.2.2 Performance estimations and architecture exploration...91

4.2.3 Validation of the high-level simulation results..93

4.3 Experiments and results analysis 94

4.3.1 Performance estimated for QCIF, using ARM7, 60MHz..94

4.3.2 Performance estimated for QCIF, using ARM946E-S, 4kI$, 4kD$, 60MHz.............95

4.3.3 Performance estimated for CIF, using ARM7, 60MHz..95

4.3.4 Performance estimated for CIF, using ARM946E-S, 4kI$, 4kD$, 60MHz 96

4.3.5 Results analysis 96

4.4 Conclusions 98

5 MPEG4 video encoder architecture implementation 100

5.1 Introducing the MPEG4 video encoder architecture implementation 101

5.1.1 Difficulties of implementing the MPEG4 video encoder on an MP-SoC architecture101

5.1.2 Principle of the proposed flow used for implementing the MPEG4 video encoder on MP-SoC..102

5.2 ROSES: a component based approach used for Hardware/Software integration 104

5.2.1 Representing the ROSES flow used for Hardware/Software integration..................104

5.2.2 Describing the COLIF representation model..105

5.2.3 Describing the ASOG tool used for OS generation..108

5.2.4 Describing the ASAG tool used for hardware generation..110

5.2.5 Describing the CosimX tool used for mixed level architecture co-simulation..........112

5.3 Implementing a complete MP-SoC architecture for MPEG4 video encoder using ROSES tool 112

5.3.1 Describing the Executable Model with Explicit Network used for ROSES flow to build the MPEG4 encoder on a complete new MP-SoC architecture 112

5.3.2 Representing the flow used for the implementation of a complete MP-SoC architecture for MPEG4 video encoder using the ROSES tool................................115

5.3.3 Libraries used for the implementation of a complete MP-SoC architecture for MPEG4 video encoder 117

5.3.4 Architecture implementation results for the MPEG4 video encoder on MP-SoC.....119

5.3.5 Performance measurements for the final MPEG4 video encoder on MP-SoC 122

5.4 Implementing the MPEG4 video encoder on an existing Quadric-Processors platform using ROSES tool 123

5.4.1 Describing the targeted Quadric-Processors platform 123
5.4.2 Describing the Executable Model with Explicit Network used for ROSES flow to build the MPEG4 application and the Operating Systems for the Quadric-Processors Platform 126
5.4.3 Representing the flow used for mapping the MPEG4 video encoder on the Quadric-Processors platform using the ROSES tool 128
5.4.4 Libraries used for mapping the MPEG4 video encoder on the Quadric-Processors platform 130
5.4.5 Implementation results for MPEG4 video encoder on the Quadric-Processors platform 131
5.4.6 Performance analysis for the obtained MPEG4 video encoder on the Quadric-Processors platform 132
5.4.7 Optimizing the MPEG4 algorithm to increase the performances of the MPEG4 video encoder on the Quadric-Processors platform 134
5.4.8 Performance analysis for the obtained optimized MPEG4 video encoder on the Quadric-Processors platform 137

5.5 Conclusion 139

6 Conclusions and perspectives 141

7 Bibliography 144

List of figures

Figure 1. Different requirements for different applications 3
Figure 2. Using the MPEG4 video encoder 15
Figure 3. Color frame represented in YUV411 format 16
Figure 4. Conversion equation from RGB to YUV and vice-versa 16
Figure 5. Simplified block diagram of the MPEG4 video encoder 17
Figure 6. Representing the different types of encoding principles for the video frames 19
Figure 7. Frame partitioning into MacroBlocks and MicroBlocks 20
Figure 8. MicroBlock partitioning for Y, U and V colours 20
Figure 9. Algorithm and functions responsible for encoding an I frame 21
Figure 10. Obtaining from the original 8x8 MicroBlock the DCT values 22
Figure 11. FDCT transformation equation 22
Figure 12. Example of obtaining the quantized valued from the DCT values 23
Figure 13. Example of obtaining the decoded DCT values after the DeQuantization 24
Figure 14. Executing the Inverse Discrete Cosine Transformation over a MicroBlock 25
Figure 15. IDCT transformation equation 25
Figure 16. Predicting a MacroBlock based on the neighbor MacroBlocks 26
Figure 17. Values used from the neighboring MacroBlocks to predict the current MacroBlock 27
Figure 18. Obtaining the final Residue MacroBlock after prediction 27
Figure 19. Algorithm and functions responsible for encoding a P frame 28
Figure 20. Representing the interpolation of a frame 30
Figure 21. Close look at an original frame, and the resulted interpolated frame 30
Figure 22. Representing the Motion Estimation for a MacroBlock 31
Figure 23. SAD formula used to compute the similarity between two 16x16 regions 31
Figure 24. Determining the Difference MacroBlock by the Motion Compensation function 33
Figure 25. Obtaining the Current Decoded MacroBlock by the Reconstruction function 34
Figure 26. Block diagram of the VLC function 35
Figure 27. Reordering a MicroBlock using the ZigZag function 36
Figure 28. MPEG4 bitstream structure for movie, frame and MacroBlock 37
Figure 29. List of different standardized resolutions 39
Figure 30. List of frame rates for different examples of application domains 39
Figure 31. Image encoded in progressive mode VS. Image encoded in interlaced mode 43
Figure 32. Representing the algorithms inclusion based on their capabilities (features) 44
Figure 33. Listing the features supported by each video compression algorithm 45
Figure 34. Computations required by MPEG4 video encoder for different video resolutions 48
Figure 35. Partitioning the MPEG4 video encoder in multiple tasks 53

Figure 36. General data flow for the parallel/pipelined MPEG4 video encoder application 56
Figure 37. Three examples of frame division methods for the QCIF resolution 57
Figure 38. Flexible architecture with 2 SMP 59
Figure 39. Simplified representation of the Splitter 60
Figure 40. Simplified representation of the MainDivX 60
Figure 41. Simplified representation of the VLC 61
Figure 42. Simplified representation of the Combiner 62
Figure 43. Describing a system using *components* and *links* 64
Figure 44. Combined Algorithm/Architecture Executable Model using MPI-SystemC 68
Figure 45. Subset of MPI communication primitives 69
Figure 46. Task-to-Task communication using MPI-SystemC 70
Figure 47. Task with flexible computations used to obtain multiple customized tasks in a SMP 71
Figure 48. Macro-generation from a task with flexible computations 72
Figure 49. Task with flexible output used to obtain task with customized number of outputs 73
Figure 50. Macro-generating a task with flexible outputs 74
Figure 51. Task with flexible input used to obtain task with customized number of inputs 75
Figure 52. Macro-generating a task with flexible input 76
Figure 53. Flexible Algorithm/Architecture Model for MPEG4 77
Figure 54. Example of task description using MPI primitives 78
Figure 55. Algorithm and Architecture parameters 78
Figure 56. Example of obtained Combined Algorithm/Architecture Executable Model 79
Figure 57. Code of MainDivX1 task obtained after the macro-expansion 80
Figure 58. Classical exploration(a) vs. Proposed exploration(b) 88
Figure 59. Detailed representation of the design flow 89
Figure 60. Obtained time annotated code for MainDivX1 task 91
Figure 61. Performance estimated for QCIF, using ARM7, 60MHz 92
Figure 62. Performance estimated for QCIF, with ARM946E-S CPUs, 4kI$,4kD$, 60 MHz 93
Figure 63. Example of architecture configuration file 94
Figure 64. Performance estimated for CIF, using ARM7, 60MHz 95
Figure 65. Performance estimated for CIF, using ARM946E-S CPUs, 4kI$, 4kD$, 60 MHz 96
Figure 66. Estimated vs. Measured performance precision 97
Figure 67. Proposed flow for implementing the MPEG4 encoder on MP-SoC architectures 103
Figure 68. Representation of the ROSES flow used for HW/SW interfaces generation 104
Figure 69. Refining an Abstract architecture to RTL architecture using ROSES 105
Figure 70. Example of architecture containing modules, ports and nets 106
Figure 71. Structural representation of a virtual component 108
Figure 72. Representation of the flow used by ASOG tool for OS generation 109
Figure 73. Representation of the flow used by ASAG tool for HW generation 111

Figure 74. DMS internal architecture for MPEG4 video encoder QCIF,4 MainDivX,1 VLC 114
Figure 75. Executable Model for MPEG4 with DMS, 4 MainDivXs, 2 VLCs 115
Figure 76. Detailed flow used to implement the MPEG4 on a new MP-SoC architecture............................ 116
Figure 77. CPU SubSystem RTL architecture for the processor of MainDivX3 task 118
Figure 78. Replacing the ARM core by the CosimX with an ARM-ISS and BFM...................................... 119
Figure 79. RTL architecture for MPEG4, QCIF, ARM7, 4 MainDivX, 1 VLC... 120
Figure 80. Colif representation of RTL Arch. for QCIF, ARM7, 4 MainDivX, 1 VLC 120
Figure 81. Colif representation of RTL Arch. for CIF, ARM9, 8 MainDivX, 2 VLC 121
Figure 82. Estimated vs. Measured performances [QCIF, 1 frame, 2 ARM7 CPUs]................................... 122
Figure 83. System architecture of the Quadric-Processors Platform 124
Figure 84. Diagram of the CPU architecture used on the Quadric-Processors Platform 125
Figure 85. Executable Model for MPEG4 with 2 MainDivXs, 1 VLC, Splitter+Combiner 127
Figure 86. Executable Model for MPEG4 with 3 MainDivXs, Splitter+Combiner+VLC 127
Figure 87. Detailed flow used to map the MPEG4 on the Quadric-Processors Platform 128
Figure 88. Real-time execution method used for the MPEG4 video encoder on the platform 129
Figure 89. Encoding speeds for the MPEG4 encoder on the Quadric-Processors platform 131
Figure 90. Computation requirements for the functions used by the MPEG4 task 133
Figure 91. Number of calls for the functions used by the MPEG4 task 133
Figure 92. Principle of the memory access optimization used for the MPEG4 video encoder 135
Figure 93. Main part of SAD function, before and after optimization 136
Figure 94. Preloading optimization for loops 137
Figure 95. Cycles gained after the optimization of the functions used by the MPEG4 task.......................... 138
Figure 96. Percentage of cycles gained for each function from total gained cycles after optimizations 138
Figure 97. Encoding speeds for the MPEG4 video encoder after the optimizations 139

1 Introduction

This chapter is an introduction related to the implementation of the video applications on multi-processors systems on chip (MP-SoC). It also presents the difficulties of integrating the video applications on a single chip, along with some related work. Finally, it lists the objectives and the contributions brought during this work.

1.1 Introduction about the MP-SoC as a solution for the implementation of video applications on a chip

During the last years, a revolution took place in the domain of the integrated circuits due to the appearance of the Multi-Processor Systems on Chips (MP-SoC). These allowed the integration of a larger amount of computational/communicational units into the same chip, which allowed the designers to take advantage of a much higher computation power.

Because of their complexity, today's applications, like multimedia, gaming, telecommunication, etc. require more and more processors to be integrated in the same chip. 90% of new ASICs in 130nm technology [1] already include at least one CPU. Multimedia platforms, like Nomadik [2] and Nexperia [3], are already multi-processor systems on chip using different types of programmable processors. Heterogeneous cores are explored to meet the tight performance and cost constraints. This trend of building heterogeneous multi-processor systems on chips will even accelerate. Future SoCs will be composed of a high number of parallel processors for applications requiring large computations. The design of SoCs will consists of an assembly of processors executing concurrent tasks. It is easy to imagine that the design of a SoC with more then a hundred of processors will become a current practice in few years (e.g. with 65nm technology in 2007). Compared with the conventional ASIC design, such a multi-processor SoC is a fundamental change in chip design.

Designing such chips requires a large amount of flexibility, in order to assure fast design, reusability and verification. These can be achieved the using embedded software approach. Like mentioned in [4], "chip designers can no longer live by silicon alone. SoC makers must also provide embedded software, which is a major competitive component. Hardware will become more and more of a commodity, and the key differentiation is in the software. The need to focus on software comes at a time when semiconductor growth is slowing."

According to Daya Nadamuni, research vice president at Gartner's semiconductor and design research group, "beyond hardware, a SoC also needs a hardware-software binding layer, an RTOS, middleware and applications. Traditional design automation tools look

only at the hardware layer. But electronic system-level (ESL) tools address hardware and software with an "integrated view" in which the SoC is treated as a system." [4]

1.2 Motivations

The MPEG4 video encoder can be found in many application domains, like home-cinema, mobile telecommunications, net-meetings, video surveillance, etc. The "inconvenience" is that each of these applications requires different functional configurations, architectural constraints, clock frequency, etc. Figure 1 shows some numbers related to the requirements specific for each application.

Application	**Resolution**	**Framerate (fps)**	**Bitrate (Kbs)**	**Allowed clock frequency**	**Power source**
Home-cinema	720x480 1920x1072	25-100	10Mbs	> 1 GHz	AC
Mobile telecom.	176x144 352x288	15-30	256-720	< 100 MHz	DC
Video conference	352x288 640x480	10-30	256-512	< 3 GHz	AC
Home Video Recording	352x288 1024x768	24-30	optional	< 100MHz < 3GHz	DC AC
Video Surveillance	640x480 1920x1072	1-25	2Mbs	< 300 MHz	AC/DC

Figure 1. Different requirements for different applications

For example, during the implementation of the MPEG4 video encoder for the Home-cinema applications, the designer should not be concerned about the allowed clock frequency and power consumption. The main target should be the high resolution and frame_rate of the movie. This is not true in case of mobile telecommunication applications. In these cases, the power consumption and maximum allowed clock frequency are restrictive aspects, whereas the targeted video resolution is much smaller.

Another aspect that has to be taken into account when implementing the MPEG4 encoder application is the Real-Time (RT) functionality. Depending on the application, the RT functionality may be required, or may NOT be required. For example, in mobile telecommunications, the RT is required. However, in case of encoding a movie for later use,

the RT is not necessary. Additionally, for all these applications, many variations of video algorithms can be used, like H264, MPEG4, MPEG2, MPEG1 etc.

In order to obtain an MPEG4 video encoder on MP-SoC, there are 3 difficult challenges that the designer has to face:

1) building the MPEG4 encoder algorithm and architecture specifications for different applications

The MPEG4 encoder algorithm and architecture specifications are highly complex, for which a significant amount of code has to be written. The final MPEG4 encoder can be targeted for different application domains (i.e. mobile phone) or configurations (i.e. video resolution), all these requiring a different algorithm functionality.

Additionally, each application domain requires different multi-processors architectures. For each of these architectures, the MPEG4 encoder algorithm has to be adapted for specific computation distributions on multiple processors, using different parallel and pipelined execution schemes. For example, for the mobile-phone applications, the MPEG4 encoder may have to execute approximately 300MIPS (Million Instructions Per Second) for QCIF (176x144) video resolution. Taking into account that for mobile phones the system cannot work at more than 100MHz (application restriction), the computations has to be distributed on multiple processors by using parallel and pipeline execution schemes.

Implementing manually different MPEG4 encoder algorithms and architecture specifications for each of the targeted applications requires a large amount of work.

2) using the correct algorithm and architecture configurations to reduce the design cost

For every application (i.e. mobile-telecom, home-cinema), the designer has to find and use the right algorithm and architecture configurations to obtain an efficient MP-SoC architecture for the MPEG4 encoder algorithm. For both, algorithm and architecture, there is a large number of parameters/configurations from which the designer has to select the right one. Using wrong algorithm and architecture configurations may lead to an inefficient chip, failure that might turn out to be very costly.

The algorithm configurations are dependent on the "client" requests, but also on the chosen architecture configurations. Architecture configurations are also dependent on the "client" requests and the chosen algorithm configurations.

For example, let us consider a mobile telecom application that requires a real-time MPEG4 encoder capable of encoding a 352x288 video resolution (algorithm parameter) using a system running at 80MHz (architecture parameter). To cope with the computation requirements, it is necessary to use a specific number and types of processors running in parallel (architecture parameter). Thus, the MPEG4 encoder algorithm has to be adapted for this parallelism level (algorithm parameter). The parallelism may increase the architecture communication requirements (architecture parameter). However, in case these communication requirements are unacceptable for this mobile telecom application, the designer may need to decrease the used parallelism level by reducing the number of processors running in parallel (architecture parameter). Such a modification imposes the need of readapting the MPEG4 encoder algorithm for the new parallelism level (algorithm parameter), and so on. Thus, it can be noticed that, it is not easy to find the correct algorithm and architecture parameters. Implementing a chip for each configuration, until finding the correct one, is a very costly approach.

3) implementing the architecture for the MPEG4 encoder application in a short time

Manually implementing the MP-SoC architecture for MPEG4 encoder, or mapping the MPEG4 encoder algorithm on already existing architecture platforms, requires a long design time. The reason is the large amount of low-level architectural details that has to be manipulated (i.e. interfaces, signals, protocols, synchronizations, addresses decoding, arbiters, etc.). In addition, this architecture has to "serve" correctly the algorithm functionality (i.e. if *Task1* needs to send data to *Task2*, the architecture has to assure that this happens correctly). To this work time, is added the time required to simulate, validate and debug the obtained results, process that (from our experiences) might double the design time.

Additionally, the designer may need to implement the MPEG4 encoder application on different types of architectures. For example, he has to implement the MPEG4 encoder algorithm plus a complete new MP-SoC architecture, or he has to implement the MPEG4 encoder on an already existing MP-SoC architecture. As a result, the designer will be

forced to become familiar with multiple design flows, specific for each of these cases. Learning all these flows requires time.

1.3 Objectives

To cope with these problems, the objectives of our work are:

1) Increase the number of potential applications, by using a common MPEG4 encoder specification. This will avoid rewriting the MPEG4 algorithm and architecture specifications for different targeted applications requiring different performances and computation distributions. The resulted MPEG4 specifications should be easy to adapt for different functionalities requiring different performances constraints. In addition, these specifications should be easy to adapt to support different levels of parallelism, required for mapping the algorithm on MP-SoC using different number of CPUs.

2) Reduce the design cost by finding the correct algorithm and architecture configurations before starting to implement the MPEG4 encoder on MP-SoC. This is done by exploring a large amount of solutions at a high-level, before the architecture implementation.

3) Reduce the design time by being capable of implementing the MPEG4 encoder on different types of architectures in a short time, using the same flow. This flow should be automated, to shorten the design time. The resulting architectures have to be either a completely new one, or an already existing architecture.

1.4 State of the art

There are multiple chips implementations containing video applications. Depending on the targeted architectures, these implementations can be categorized into 3 approaches:

1) implementations using HW approach:

The implementations using HW approach consists of designing specific HW components containing a specific algorithm. Such implementations can be found in [18] and [49

]. The implementation from [18] supports MPEG4 and MPEG2 encoders/decoders for CIF video resolution running at 25 frames/sec. The implementation from [49] supports H.264 encoder/decoder for 1024x720 video resolution at 30 frames/sec.

The target applications are the mobile telecommunication and video conferences for [18] and home-cinema for [49]. The advantages of these implementations are high performance and low-power consumption. Their main disadvantage is that they are not suitable for applications that should support different algorithm parameters. For example, in case of home video recording applications (i.e. video camera), the user may want to choose a different compression quality, video resolution, frame_rate, bitrate, etc., based on his preferences. Since in the implementations from [18] and [49], most of these parameters are hardwired into the chip, they do not provide many manipulation options. These implementations are application specific, and they cannot be adapted for other applications.

The design cost of these implementations is high, since any exploration for algorithm and architecture configurations can be done only after the implementation phase. Thus, in case the algorithm and architecture parameters prove to be wrong, the implementation phase has to be started all over again.

The design time is high since the entire algorithm and architecture has to be implemented using low-level description languages by manipulating a large amount of fine HW details. Additionally, the simulation, validation and debug phases require a fastidious and long work time.

2) implementations using SW approach

The implementations using SW approach consist of mapping the entire algorithm on architectures containing one or more CPUs. The instructions from the algorithm will be interpreted and executed by these CPUs. Such implementations can found in [17] and [50]. The implementation from [17] contains an MPEG4 encoder specially developed to encode a movie at CIF (352x288) video resolution at a frame_rate of 30 frames/sec. The implementation from [50] contains an H.264/AVC encoder (no specifications are made about the supported video resolution and frame_rate).

The targeted applications are the video conferences for [17] and mobile telecommunication for [50]. The advantage of these implementations is the easiness of designing and modifying the algorithm. Their main disadvantages are the low performances and high power consumption. Thus, the implementations from [17] and [50] cannot be targeted for applications requiring a large amount of computations, like home-cinema. Additionally, since in [17] the MPEG4 encoder algorithm was specifically implemented for a CIF video resolution at a frame_rate of 30 frames/sec., the algorithm may have to be modified if other configurations are required (i.e. other video resolution).

The design cost of these implementations is significantly lower than the implementations using a HW approach. This is due to the fact that multiple algorithm configurations can be explored without having to implement the architecture. Additionally, even after the architecture is implemented, the algorithm can be sometimes modified and remapped on the architecture, without having to modify the architecture. However, in the implementations from [17] and [50], the used algorithms do not allow to change the number of CPUs from the architecture.

The design time of these implementations is significantly smaller than the design time required for the implementations using the HW approach. The implementation consists mainly of SW mapping on the processors. Since the algorithm is implemented in SW, fewer HW details will have to be implemented. However, such implementations require the use of mixed HW and SW tools.

<u>3) implementations using mixed HW/SW approach (a.k.a. ASIP approach)</u>

The implementations using mixed HW/SW approach consist of mapping just a part of the algorithm on CPUs, and the other part is integrated in HW. Usually, on HW are integrated the algorithm's computational intensive functions. On CPUs are mapped the functions which do not require intensive computations, like control functions. Such implementations can be found in [19] and [51]. The implementation from [19] contains an MPEG4 encoder for VGA (640x480) at 25 frames/sec, on an architecture containing 4 ARM9 processors at 200MHz, and a shared HW accelerator. The implementation from [51] contains a H.264/AVC decoder for 1440x1080 interlaced video resolution running on a specific processor running at 90MHz.

The targeted applications are video surveillance and video conferences for [19] and home-cinema for [51]. The advantages of these implementations are the medium performances and medium power consumption. Additionally, since a part of the algorithm is implemented in CPUs, this part is easy to design and modify. However, their disadvantage is the need of implementing the algorithm's computational intensive functions in HW, which is reducing the configurability of these functions for different parameters. This means that is difficult to use these implementations for applications that require manipulating a large amount of parameters, like the home video recording applications.

The design cost is somewhere between the design cost required for HW approach and design cost required for SW approach. The higher the amount of functions is implemented in HW, the higher the design cost is. In addition, exploring different algorithm and architecture configurations may turn out to be very costly when many functions from the algorithm are implemented in HW.

The design time is also between the design time required for HW approach and the design time required for SW approach. The higher the amount of functions is implemented in HW, the higher the design time is. In addition, the time required for simulation, validation and debug depends on the amount of functions implemented in HW. However, such implementations require the use of mixed HW and SW tools.

In all these 3 approaches, the presented implementations have some common points:

1) The number of applications that can be targeted is limited. One of the reasons is that the MPEG4 algorithm is designed and configured for the targeted application and used architecture. When targeting another application or architecture, the algorithm will have to be modified.

2) The design cost depends on the amount of low-level architecture details that has to be implemented until measuring some performances becomes possible. These measured performances help to find the correct algorithm and architecture configurations. If these measurements are done after completely implementing the low-level architecture, any change into the algorithm and architecture configurations becomes a

costly decision. Additionally, based on the requirements of the targeted application, different algorithms and architecture configurations are needed.

3) The design time depends on the amount of low-level architecture details which has to be implemented and the amount of algorithm mapped into CPUs. In addition, this design time depends on the speed of simulation, validation and debugging.

1.5 Contributions

1.5.1 Flexible modeling style to describe the algorithm and architecture specifications for MPEG4 encoder

To be able to target multiple applications, we propose a MPEG4 encoder algorithm that can be used for multiple applications. This algorithm can be configured for different algorithm parameters required by the target application (i.e. video resolution, compression quality, bitrate, frame_rate, etc). Additionally, this algorithm can be configured for different multiprocessor architectures required by the targeted application. This is done thanks to the use of computation distribution techniques, using parallel/pipeline execution schemes.

As contribution, this document is presenting a flexible modeling style to describe the algorithm and architecture specifications for MPEG4 encoder. Starting from a unique MPEG4 algorithm/architecture representation, we can obtain automatically different algorithm/architecture models for MPEG4, for different algorithm/architecture parameters and parallel/pipeline execution schemes. This contribution will be detailed in chapter 3.

1.5.2 High-Level algorithm/architecture exploration for MPEG4 encoder with custom parameters

To reduce the design cost, we propose to find the correct algorithm and architecture before starting the architecture implementation. This is done using a High-Level algorithm/architecture exploration. The key benefits of this approach are:

a) The designer can automatically obtain the algorithm and architecture models required to achieve the exploration, thanks to the flexible modeling style used to obtain the required algorithm and architecture specifications.

b) Measuring the performances at a high-level, and not at a low-level, is done much faster, since many low-level architecture details are abstracted. This allows the designer to test many algorithm and architecture configurations, in a short time.

c) The precision of the measurements done at a high-level is assured by precisely estimating the computation times and communication times running together.

The contribution is a high-level algorithm/architecture exploration method for MPEG4 encoder with custom parameters, which is covering the previously presented advantages. Using this approach, multiple algorithm and architecture configuration solutions can be explored in a much shorter time compared to the exploration done at a low-level. This will avoid implementing a costly MP-SoC architecture for MPEG4 encoder with wrong algorithm and architecture configurations. As a result, the design cost is decreased. This contribution will be detailed in chapter 4.

1.5.3 Common flow used for the implementation of MPEG4 encoder on different targeted architectures

To reduce the design time, we propose a method to obtain in a short time the low-level MP-SoC architecture for MPEG4 encoder. This is done thanks to a component based design technology. The benefits are the possibility to obtain automatically the low-level SW and HW details for the MP-SoC architecture for MPEG4 encoder. The validation and debug speed is increased by allowing the simulations at different abstraction levels.

The contribution is a common flow used for the implementation of MPEG4 encoder on different targeted architectures, required for multiple applications. Using this flow, a complete MP-SoC architecture for MPEG4 encoder was implemented. Additionally, using this same flow, the MPEG4 encoder was implemented on existing quadric-processors architecture. The design time required to obtain different MPEG4 encoder architectures was drastically reduced.

1.6 Document outline

The rest of the document is organized as follows. Chapter 2 presents in detail the MPEG4 video encoder algorithm. Chapter 3 addresses the adaptation of MPEG4 encoder for parallelism/pipeline support along with the flexible modeling style used to describe the algorithm and architecture specifications. Chapter 4 describes the high-level algorithm/architecture exploration for MPEG4 encoder with custom parameters, and some exploration results. Chapter 5 details the flow used for implementing the MPEG4 video encoder on a completely new architecture and on the existing quadric-processors architecture, along with some implementation and performance results. Finally, some conclusions are presented in chapter 6.

2 MPEG4 video encoder algorithm

This chapter presents the MPEG4 encoder algorithm. The algorithm receives as input an uncompressed movie. The algorithm is in charge of compressing this movie, and the output is an MPEG4 bitstream containing the compressed movie with lower image quality. The MPEG4 video encoder algorithm compresses the movie frame by frame. Each frame is compressed in two possible ways: either as I frame, either as P frame. For an I frame, the algorithm compresses the entire original frame. For a P frame, the algorithm compresses only the spatio-differences between this frame and the previous frame. The MPEG4 video encoder algorithm can be configured using different parameters: video resolution, frame_rate, bitrate, quality, etc. These have a direct impact on the algorithm's computations requirements.

2.1 Introducing the MPEG4 video encoder

2.1.1 MPEG4 video encoder as new solution to MPEG2 video encoder

Video compression has become an essential component of home entertainment video and Internet streaming video. The success of these ones boosted upon the arrival of MPEG2 video encoding standard. This standard proved its effectiveness, but after almost 15 years of existence, and almost 10 years of use, it becomes obsolete. It is clear that the time is right to replace the MPEG2 video encoding with a more effective and efficient technology, that can take advantage of the recent progress in processing power [5].

In the last years, there was an assiduous debate which technology should replace the MPEG2. There is no doubt that the winner is the MPEG4 video encoder standard, and recently also the H264 video encoder standard, too. However, the H264 video encoder standard is just at its early stages of development, while the MPEG4 video encoder standard is already a mature solution [5].

The MPEG4 video encoder standard was developed by the Motion Picture Experts Group, and it was standardized around 1994 [6][7]. Only in the last approximately 6-7 years it started to be implemented. Since then, many MPEG4 implementations were done, from which the most popular are OpenDivX[8], DivX[9] and XviD[10]. Between each of these, there are some differences, mainly in the computations precisions and the compressed frame quality. This variation can exist, because the MPEG4 specifications are not restricting severely the computations inside all the steps required by the compression. The main restrictions are related to the steps that have to be followed, and the syntax of the output (compressed movie). This "non-strictness" is one of the many reasons why the MPEG4 becomes so popular.

Another reason why the MPEG4 is so popular is its compression capability. Practical experiments showed that while the MPEG2 video encoder is compressing a 2 hour movie on a 4.7Gbytes DVD, the MPEG4 is capable of compressing the same movie on a 700Mbytes CD-ROM. It is true, that using MPEG2, the resulted frame quality may be better, still with MPEG4 the resulted frame quality is still acceptably high [5].

However, a disadvantage of the MPEG4 encoder, compared with the MPEG2 encoder, is its algorithmic complexity. Comparing the algorithm used by MPEG4 with the one used by MPEG2, the algorithm of the MPEG4 is about 3 times bigger than the MPEG2. However, this disadvantage is not penalizing enough to break the success of the MPEG4 [5].

2.1.2 Usual method of using the MPEG4 video encoder

The MPEG4 video encoder receives an uncompressed movie, encodes it and obtains a compressed bitstream (Figure 2).

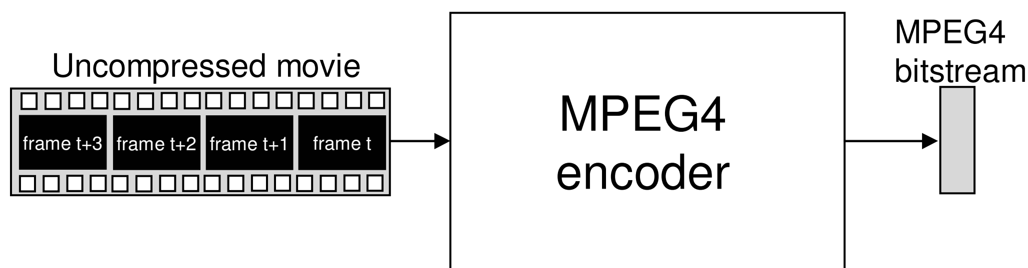


Figure 2. Using the MPEG4 video encoder

The difference between the MPEG4 video encoder and the other video encoding standards are the algorithm used to compress the movie, and the syntactic format of the resulted bitstream. These will be detailed later in this document.

Yet, the MPEG4 video encoder uses the same syntactic format for the input movie, as the other video encoders. The input movie is in uncompressed format, as a continuous stream of pixels. In this work, the format used by the uncompressed movie is called YUV411 [48] where the pixels streaming order is line by line from top to bottom, and each line is filled from left to right. The stream bandwidth depends on the number of frames per second, and frame size.

2.1.3 Describing the YUV411 video format used as input by the MPEG4 video encoder

The YUV411 is a video format used for video encoders, including the MPEG4. It was developed first for the TV video signal broadcast, in order to assure the signal compatibil-

ity between the new color TVs, and the old black&white TVs [5]. Visually, the frames using YUV411 format (Figure 3) are different from the usual frames using RGB format.

Figure 3. Color frame represented in YUV411 format

In RGB format, each pixel is represented using 3 color values: Red, Green and Blue. This means that the size required to store an RGB frame is 3 times bigger than the number of pixels from the frame.

In case of YUV411, each pixel is represented using 3 color values: Luminance, U and V. The last two are also known as Chroma Blue (Cb) respectively Chroma Red (Cr). The Luminance values are forming exactly the black&white frame. The U and V are values that are "adding" the colors. In the YUV411 format, there are 4 times less numbers of U and V values than the number of Luminance values (Figure 3). Each 4 pixels are represented using 4 Luminance values, 1 U value and 1 V value [5]. Therefore, unlike RGB format, the size required to store a YUV411 frame is only 1.5 times bigger than the number of pixels from the frame. A common method to stream a frame in YUV411 format is to transmit all the Luminance values first, then all the U values, and then the V values.

In order to assure the conversion possibility from a frame in RGB format to YUV411 format, and vice-versa, mathematical transformations are used [11]. These are presented in Figure 4, and are applied for each pixel from the frame.

$$Y = 0.299*R + 0.587*G + 0.114*B$$
$$U = Cb = 0.564*(B-Y)$$
$$V = Cr = 0.713*(R-Y)$$

$$R = Y + 1.402*V$$
$$G = Y - 0.344*U - 0.714*V$$
$$B = Y + 1.772*U$$

Figure 4. Conversion equation from RGB to YUV and vice-versa

The reason why the MPEG4 video encoder uses the frame in YUV411 format, and not RGB format, is that in YUV411 format, the objects in the frame (stored in the Luminance values) are clearly separated from the colors in the frame (stored in the chromas values). Moreover, the MPEG4 video encoder algorithm treats differently these two elements, as it will be shown later in this document. However, this is not possible using the RGB format.

2.1.4 Presenting the structure of the MPEG4 video encoder algorithm

The video compression using MPEG4 algorithm is done by saving only the spatio-temporal differences between consecutive frames: the current one, and the previous one. These differences are then compressed and stored into a well-defined (and strict) stream format, called MPEG4 bitstream. This bitstream will be used as input during the video decoding process. A simplified block diagram of the MPEG4 video encoder is presented in Figure 5.

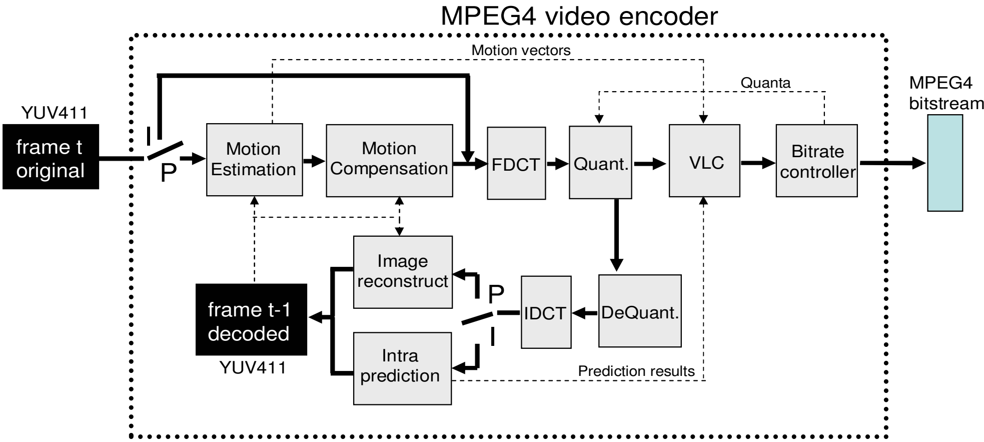


Figure 5. Simplified block diagram of the MPEG4 video encoder

It is important to mention that every time when encoding the current frame, the previous frame used by the MPEG4 video encoder to determine the spatio-temporal differences, is not the original one, but the decoded one. To obtain this frame, the algorithm first encodes the frame, and then it has to decode it, to be used later when encoding the following frame. The reason is the following: the MPEG4 decoder will never know the previous original frame, but only the previous decoded frame. When the MPEG4 decoder will de-

compress the current frame (which initially contains only differences), it will "add" it to this decoded previous frame, in order to obtain the final current decoded frame.

If the MPEG4 encoder would compute these differences relative to the previous original frame (perfect quality), and not the previous decoded frame (lower quality), these differences are not "capturing" the quality degradation of the decoded previous frame. If the MPEG4 decoder would "add' these incomplete differences to its previous decoded frame, it will result in a continuous increase of quality degradation after decoding each frame.

Nevertheless, if the MPEG4 encoder computes these differences relative to the previous decoded frame, and not the previous original frame, these differences will "capture" also the quality degradation of the previous decoded frame. By "adding" these complete differences to the previous decoded frame, the MPEG4 decoder will arrive also at restoring the quality of the previous decoded frame, and at obtaining the new decoded frame.

In conclusion, by using the previous decoded frame during the MPEG4 encoding, and not the previous original frame, the MPEG4 encoder will compute the differences in such a way that the MPEG4 decoder will be able to keep the movie‘s quality degradation to a constant level. Otherwise, this level would increase exponentially. This is why, the MPEG4 encoder has to encode the current frame, but also to decode it. This way, the MPEG4 encoder knows what the decoded frame will look like when decoding the movie by the MPEG4 decoder.

In order to compress the movie, the MPEG4 video encoder algorithm distributes the frames into two categories. For each category, the MPEG4 video encoder will compress the frames using different encoding principles.

2.1.5 Different types of encoding principles used for the video's frames

The MPEG4 video encoder treats the frames as I (Integral frame, a.k.a Intra) or as P (Partial frame, a.k.a Inter). The frames encoded as I, are encoded as "new" frames, without taking into the consideration the previous frame. The frames encoded as P will be treated taking into account the information existing in the previous frame.

The first frame from the movie is encoded all the time as I (because there is no previous frame to be taken into account). During the movie, several other frames can be encoded as I. The frames existing between two I frames will be encoded as P (Figure 6). The interval between two I frames is called key_frames. In case the key_frames value is 100, that means that an I frame will be used every 100 frames, the remaining 99 frames between these two I frames will be treated as P.

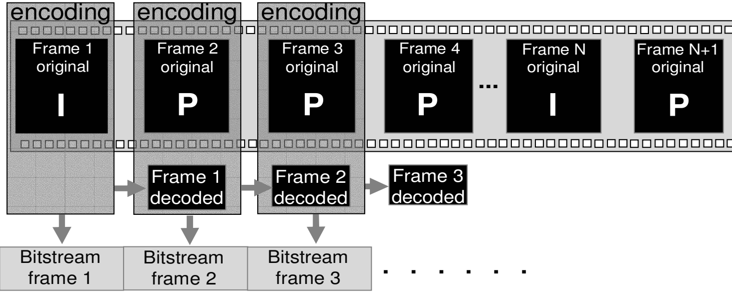


Figure 6. Representing the different types of encoding principles for the video frames

Taking into account that the compressed size of the I frames is generally bigger than the compressed size of the P frames (exceptions may occur), by using too many I frames during a movie will lead to decreased compression efficiency, and worse compressed frame quality. However, using many I frames will improve the speed of seeking into the compressed movie (by the decoder). The advantage of using I frames, is the increased tolerance to possible errors that might appear in the bitstream, because of multiple factors: error in the bitstream construction, transmission, storage, etc.

Theoretically, it is possible to use during a movie only one I frame at the beginning, and the rest of the movie's frames to be encoded as P. This is a risky solution, because it eliminates the seek feature. Additionally, in case the bitstream is deteriorated during the movie, the remaining movie sequence will be completely compromised.

The specifications of the MPEG4 video encoder mention nothing regarding a 'recommended" value for the key_frame. This aspect is completely left to the decision of the human factor. However, the algorithm is capable of forcing a frame to be compressed as I, even if it was supposed to be compressed as P. This happens when the current frame is too different from the previous frame, in which case compressing the frame as P will lead to worse compression ratio compared with the case of treating this frame as I. This feature is

called “scene change detection” [5]. Additionally, this can also be applied for a region of the frame only, as will be shown later in the document.

2.2 Describing the encoding principle for an I frame

2.2.1 Introducing the concept of encoding an I frame

In order to encode an I frame, the MPEG4 video encoder partitions the frame into small areas of 16x16 pixels, called MacroBlocks. In addition, each of these MacroBlocks is partitioned into 4 small areas of 8x8 pixels called MicroBlocks (Figure 7).

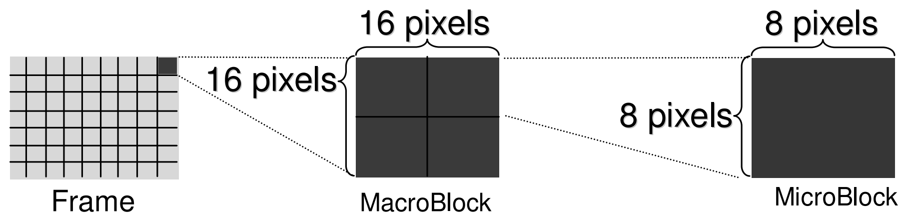


Figure 7. Frame partitioning into MacroBlocks and MicroBlocks

Since, as previously mentioned, the frame is formed by three colors (Y, U and V), each of the frame’s MacroBlocks is formed from 6 MicroBlocks: 4 for Y, 1 for U and 1 for V. This is because the U and V frames are not partitioned into MacroBlocks, but only in MicroBlocks (Figure 8).

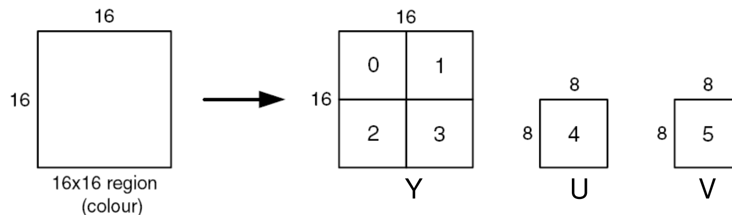


Figure 8. MicroBlock partitioning for Y, U and V colours

2.2.2 Representation of the algorithm and functions used for encoding an I frame

In case of the I frame, the MPEG4 video encoder will treat each of these MacroBlocks, and its MicroBlocks, in a progressive order, starting with the MacroBlocks on the first line,

from left to right, then the second line, and so on. Figure 9 shows the functions used to encode a MacroBlock for an I frame.

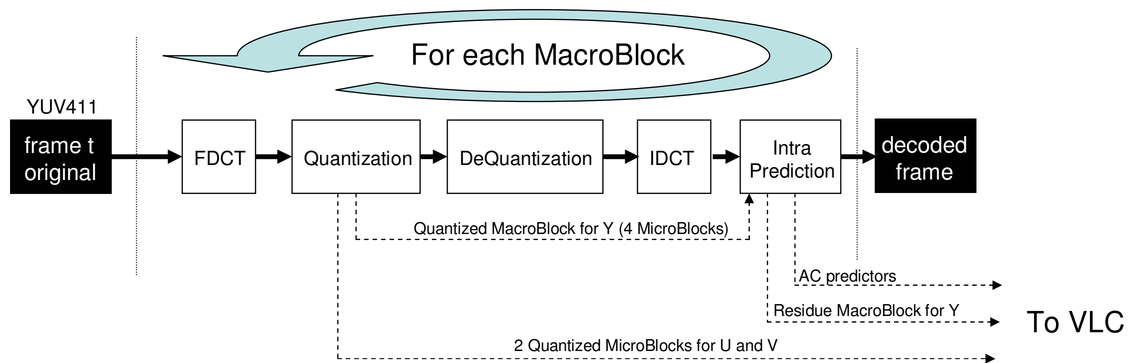


Figure 9. Algorithm and functions responsible for encoding an I frame

For each of the MacroBlocks, the MPEG4 video encoder will first transform frame values into frequency values using FDCT (Fast Discrete Cosine Transformation) transformation. After that, it will Quantize these values in order to loose an amount of quality. The following steps are used to decode the MacroBlock. The quantized values are DeQuantized in order to restore them close to the original values (before Quantization), and then retransform them into frame values using IDCT function. In the end, the quantized values are executed by the IntraPrediction function, in order to determine “fading” (AC coefficients) in the MacroBlock. The result will be this “fading” and the quantized MacroBlock from which the “fading” was subtracted (Predicted MacroBlock). These two elements will be sent to the VLC (Variable Length Compression), which will b compress them.

2.2.3 Describing the Fast Discrete Cosine Transformation (FDCT)

By using the FDCT, the frame values stored into a MacroBlock will be transformed into frequency values. The FDCT is applied separately for each of the 6 MicroBlocks forming the MacroBlock. Since the size of the MicroBlock is 8x8 pixels, the FDCT will be an 8x8 transformation. Figure 10 presents an example of DCT values obtained from an 8x8 MicroBlock, and its actual purpose is described in detail in [5].

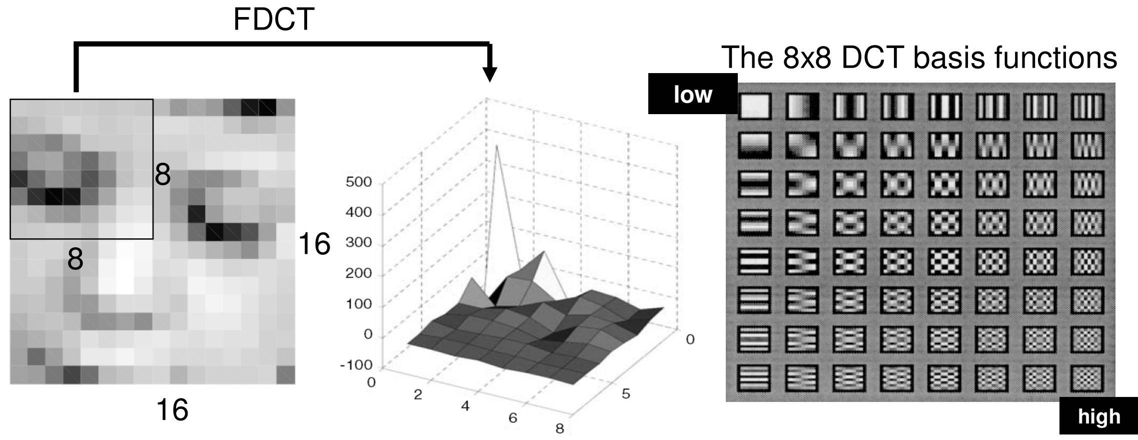


Figure 10. Obtaining from the original 8x8 MicroBlock the DCT values

The obtained DCT value from the [0,0] position belongs to the lowest frequency of the frame, while the value from the [7,7] belongs to the highest frequency. Just like sounds, every frame can be composed from a multitude of frequencies with different amplitudes. The values obtained after the FDCT are these amplitudes, associated to each frequency.

In order to achieve these transformations, the equations presented in Figure 11 are used. The X_{ij} are the pixels belonging to the original frame, and the Y_{xy} are the obtained DCT values. The value of the N is 8.

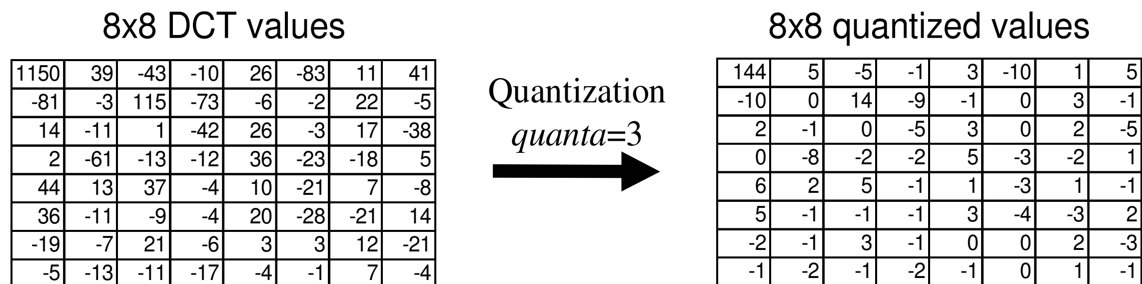
$$Y_{xy} = C_x \sum_{i=0}^{N-1} \left[C_y \sum_{j=0}^{N-1} X_{ij} \cos \frac{(2j+1)y\pi}{2N} \right] \cos \frac{(2i+1)x\pi}{2N} \quad , \quad C_{x,y} = \sqrt{\frac{1}{N}}$$

Figure 11. FDCT transformation equation

As it can be noticed, this equation requires cosine operations, which leads to floating point operations. However, because the parameters used by them are well known, there are existing methods using fixed point operations, from which one of the most popular is Lee’s algorithm [12].

2.2.4 Describing the Quantization function

Using the transformed MacroBlock after the FDCT function, the next step is to quantize these values. Just like the FDCT function, the quantization is done for each MicroBlock from the current MacroBlock, and it is the main source of the quality degradation caused by the MPEG4 video encoder. The quantization is done by removing a specific number of bits from all the 8x8 DCT values. The number of bits that has to be removed is called *quanta*. This *quanta* can vary from frame to frame, but it is unique during the quantization of the MacroBlocks from the same frame.



8x8 DCT values

1150	39	-43	-10	26	-83	11	41
-81	-3	115	-73	-6	-2	22	-5
14	-11	1	-42	26	-3	17	-38
2	-61	-13	-12	36	-23	-18	5
44	13	37	-4	10	-21	7	-8
36	-11	-9	-4	20	-28	-21	14
-19	-7	21	-6	3	3	12	-21
-5	-13	-11	-17	-4	-1	7	-4

8x8 quantized values

144	5	-5	-1	3	-10	1	5
-10	0	14	-9	-1	0	3	-1
2	-1	0	-5	3	0	2	-5
0	-8	-2	-2	5	-3	-2	1
6	2	5	-1	1	-3	1	-1
5	-1	-1	-1	3	-4	-3	2
-2	-1	3	-1	0	0	2	-3
-1	-2	-1	-2	-1	0	1	-1

Figure 12. Example of obtaining the quantized valued from the DCT values

In the example presented in Figure 12, by quantizing with 3 it is equivalent with a division of 8 (2^3). In case the DCT value is 1150 (in binary is 10001111110), after the quantization using a *quanta*=3, the resulted value is 143.75. Depending on the designer's choice, in order to avoid using floating point values, this value can be either truncated (ignore the decimals) to obtain the value 143, or it can be rounded up to obtain the value 144. Using the rounding feature provides a slightly better image quality, compared with the case of truncating the value [5].

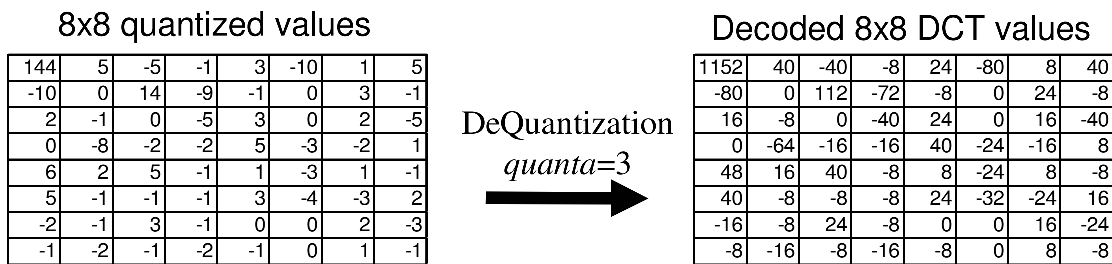
The purpose of the quantization is to "sacrifice" frame information in order to obtain values that can be better compressed. By quantizing the DCT values, the resulted quantized values will contain many zero values. As will be explained later in this document, the more zeros there are, the better the compression ratio obtained by the VLC function will be. However, the quantization process is irreversible, so using too big *quanta* value will lead to significant quality loss, but a good compression. The minimum possible *quanta* value mentioned in the MPEG4 specifications is 2, and the maximum possible *quanta* value is 31.

2.2.5 Describing the DeQuantization function

If until this moment the MacroBlock was encoded, several steps are needed to decode it. After encoding and decoding all the MacroBlocks, the entire decoded frame can be obtained. This will be used as “previous frame” when encoding the next frame.

The first decoding step is to DeQuantize the values of the MacroBlock, which were Quantized in the previous function. The DeQuantization function works at MacroBlock level, and it treats separately the 6 MicroBlocks contained in the current MacroBlock.

Overall, the operations executed during the DeQuantization are exactly the opposite of the operations executed during the Quantization. It takes all 64 values of the MicroBlock, and concatenates a number of less significant bits. The number of added bits is the same as the one used by the Quantization for removing the less significant bits, defined by the *quanta* parameter (Figure 13).



8x8 quantized values

144	5	-5	-1	3	-10	1	5
-10	0	14	-9	-1	0	3	-1
2	-1	0	-5	3	0	2	-5
0	-8	-2	-2	5	-3	-2	1
6	2	5	-1	1	-3	1	-1
5	-1	-1	-1	3	-4	-3	2
-2	-1	3	-1	0	0	2	-3
-1	-2	-1	-2	-1	0	1	-1

Decoded 8x8 DCT values

1152	40	-40	-8	24	-80	8	40
-80	0	112	-72	-8	0	24	-8
16	-8	0	-40	24	0	16	-40
0	-64	-16	-16	40	-24	-16	8
48	16	40	-8	8	-24	8	-8
40	-8	-8	-8	24	-32	-24	16
-16	-8	24	-8	0	0	16	-24
-8	-16	-8	-16	-8	0	8	-8

Figure 13. Example of obtaining the decoded DCT values after the DeQuantization

Comparing the obtained Decoded 8x8 DCT values with the original DCT values before the Quantization (Figure 12), it can be noticed that the values are different, even if they are close. This is caused by the irreversible quality loss resulted after the Quantization. Therefore, the higher is the value of *quanta*, the higher will be the difference between the Decoded 8x8 DCT values and the original DCT values.

2.2.6 Describing the Inverse Discrete Cosine Transformation (IDCT)

After the Decoded 8x8 DCT values are obtained for the current MacroBlock, the next step is to retransform these frequency values into actual frame values. Just like the initial

FDCT function applied it computations for each MicroBlock from the current MacroBlock, so is the IDCT function (Figure 14).

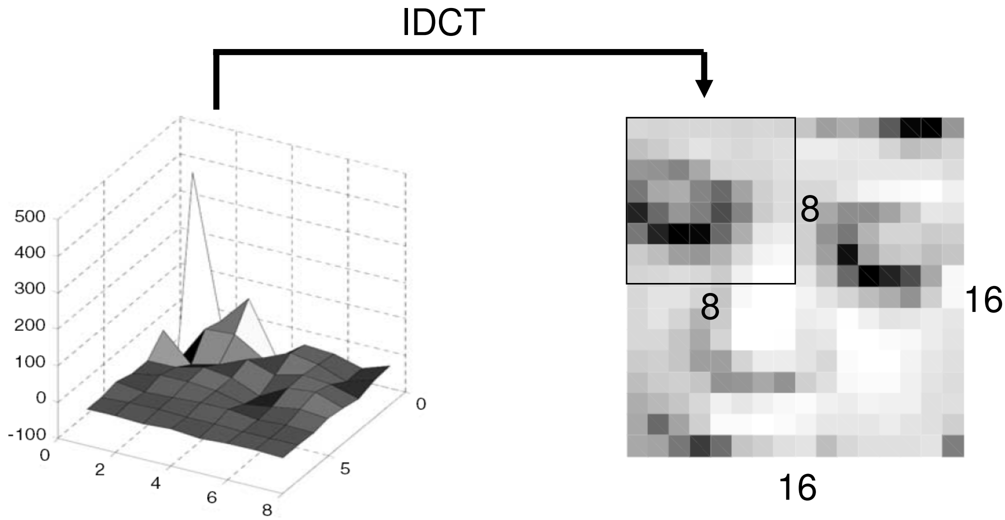


Figure 14. Executing the Inverse Discrete Cosine Transformation over a MicroBlock

In order to achieve these transformations, the equations presented in Figure 15 are used. The Y_{xy} are the values belonging to the Decoded DCT values, and the X_{ij} are the frame pixels belonging to the resulted decoded frame. The value of the N is 8.

$$X_{ij} = \sum_{x=0}^{N-1} C_x \left[\sum_{y=0}^{N-1} C_y Y_{xy} \cos \frac{(2j+1)y\pi}{2N} \right] \cos \frac{(2i+1)x\pi}{2N} \quad , C_{x,y} = \sqrt{\frac{1}{N}}$$

Figure 15. IDCT transformation equation

The IDCT transformation equation also contains cosine operations, which leads to floating point operations. However, as the FDCT, there are algorithms using fixed point operations [12].

The important thing is that after this phase, was obtained completely the decoded MacroBlock, which will form a part of the decoded frame. The other parts of the decoded frame are obtained similarly, by encoding and decoding all the other MacroBlocks. The resulted decoded frame will be used as “previous frame” when encoding the next frame. However, the process of treating the current MacroBlock is not yet finished. There is still one step before sending the results to the VLC for compression.

2.2.7 Describing the Intra Prediction function

Before sending the Quantized MacroBlock to the VLC for compression, one more transformation can be achieved over this MacroBlock. Even if the values of this MacroBlock contain many zeros after the quantization, there is still some information that can be eliminated, but this time without any more quality degradation.

This step is applied only for the Y color from the current MacroBlock, and it is achieved at MacroBlock level, contrary to the previous steps that worked at MicroBlock level. Often, the 16x16 quantized values from the current MacroBlock represent a frame that contains a well-defined pattern, also known as "fading". This "fading" has to be determined, and eliminated.

Usually, only by knowing the frame values stored into the neighboring MacroBlocks of this current MacroBlock, it is possible to predict what this current MacroBlock might look like (Figure 15). It is true, that the predicted result might be different with what the current MacroBlock contains. In this case, the prediction will be aborted.

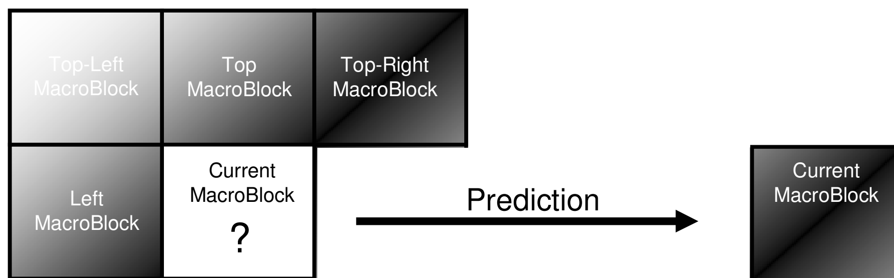


Figure 16. Predicting a MacroBlock based on the neighbor MacroBlocks

In order to predict the current MacroBlock, a set of values from the neighboring MacroBlocks (Figure 17) are used. These neighboring MacroBlocks are already decoded, from a previous step of the current loop of encoding the MacroBlocks of the current frame as I. From these values and the prediction algorithm described in [5], are determined a set of 6 values, called AC predictors. Using also the quantized values of the current MacroBlock, the algorithm can adjust these AC predictors, or in the worse case abort the prediction if the actual current MacroBlock is different from the predicted one.

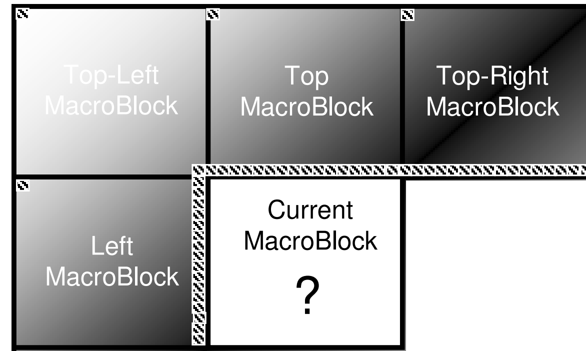


Figure 17. Values used from the neighboring MacroBlocks to predict the current MacroBlock

After the prediction is finished, this "fading" will be eliminated from the Quantized current MacroBlock, obtaining a Residue MacroBlock (Figure 18). As a result will be obtained a MacroBlock from which were replaced a great amount of values with only 6 values (the AC predictors).

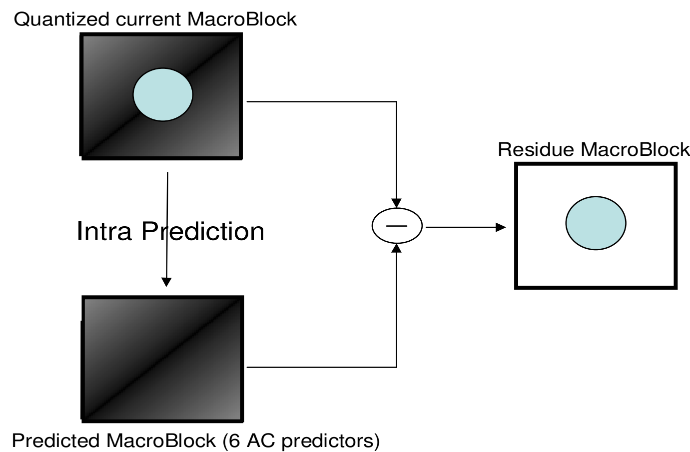


Figure 18. Obtaining the final Residue MacroBlock after prediction

So, instead of sending the Quantized current MacroBlock to the VLC for compression, after the Prediction, only the Residue MacroBlock (which contains a great amount of zero values) and 6 AC predictors values are sent to the VLC. The MPEG4 video decoder will receive the Residue MacroBlock with the 6 AC predictors. From these 6 AC predictors and the neighboring MacroBlocks it will reconstruct the Predicted MacroBlock, which will be added to the Residue MacroBlock to obtain the actual Quantized current MacroBlock.

2.3 Describing the encoding principle for a P frame

2.3.1 Introducing the concept of encoding the P frame

When encoding a P frame, just like in the case of encoding an I frame, the algorithm partitions the frame into 16x16 pixels MacroBlocks, each of these MacroBlocks being partitioned into 4 MicroBlocks with the size of 8x8 pixels (Figure 7). Because the frame is formed by 3 elements (Y, U and V), for each of the frame's MacroBlocks it will be associated 6 MicroBlocks: 4 for Y, 1 for U and 1 for V. In this case, the U and V frames are not partitioned into MacroBlocks, but only MicroBlocks (Figure 8).

2.3.2 Representation of the algorithm and functions used for encoding a P frame

For encoding a P frame, the algorithm takes into account the current frame, but also the previously decoded frame. It is irrelevant if previously, this decoded frame was obtained after encoding an I or P frame. In Figure 19 is presented the algorithm and functions responsible for encoding a P frame.

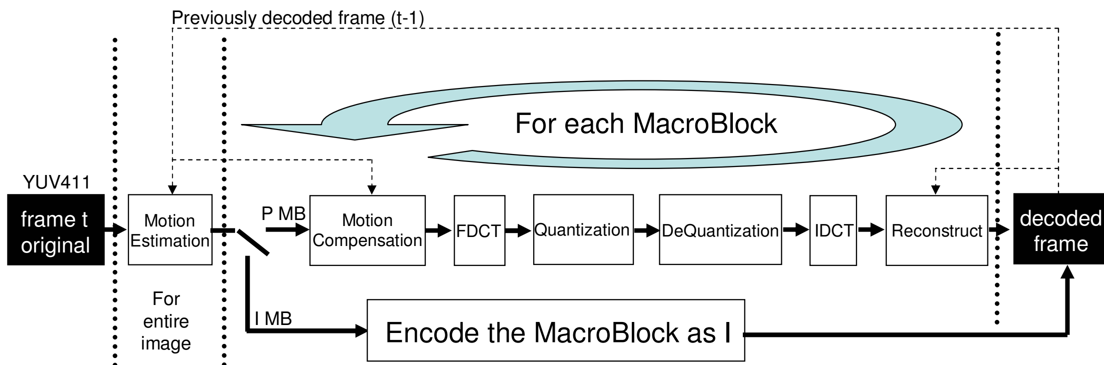


Figure 19. Algorithm and functions responsible for encoding a P frame

Using these 2 frames, the Motion Estimation phase determines if the current frame includes information which can be already found in the previous decoded frame, even if not at the same coordinates. For each MacroBlock from the current frame, the Motion Estimation can obtain two possible results: either this MacroBlock was found in the previous frame (even if it is not 100% similar), either this MacroBlock contains 16x16 pixels which are completely new, and it cannot be found in the previous frame.

In case the MacroBlock was not found, the algorithm will compress this MacroBlock as I. The algorithm used for compressing a MacroBlock as I was presented in the previous subchapter.

In case the MacroBlock was found somewhere in the previous decoded frame, the MotionEstimation will provide a motion vector for this MacroBlock. This motion vector is indicating the origin of this current MacroBlock, from the previous decoded frame. Of course, this current MacroBlock might contain some small differences compared to the frame from the previous decoded frame. Therefore, the MotionCompensation will compute this difference, obtaining a MacroBlock that contains only residues (extra data). As a result, the current MacroBlock, which initially contained a 16x16 original pixels, after the MotionEstimation and MotionCompensation, it will contain only some few data. The rest can already be found somewhere in the previous decoded frame.

This resulted MacroBlock containing residues will be encoded and decoded using the FDCT, Quantization, DeQuantization and IDCT functions. Finally, these encoded/decoded residues will be "added", MacroBlock after MacroBlock, to the previously decoded frame by the Reconstruction function, obtaining in the end the new decoded current frame. This will be used as previous frame when encoding the next frame.

2.3.3 Describing the Motion Estimation function

The goal of the MotionEstimation function is to find for each of the MacroBlocks from the current original image, if there exists a similar one in the previous decoded one. If it does, the MotionEstimation computes the motion vector. In addition, the MotionEstimation is executed using the Y image. The U and V are not used by the MotionEstimation.

The MotionEstimation estimation can use the decoded previous image in two ways: either zoomed by 2 on both axes (interpolated), or as it is (unchanged). In the first case, the MotionEstimation will provide precise results, but it will require more computations [5]. In the second case, the MotionEstimation will provide less precise results, but it will require approximately 4 times less computations. Choosing which of these cases to use depends on the application.

The interpolated frame is computed using an interpolation algorithm. By zooming a frame twice on both axes, for each of the frame's pixels will have to be computed 3 extra new pixels. As it can be seen in Figure 20, if the original frame (left) is formed only by P pixels, the interpolated frame (right) contains these P pixels, along with new 3 types of pixels associated to each P pixels: Horizontal pixels, Vertical pixels, and HV pixels.

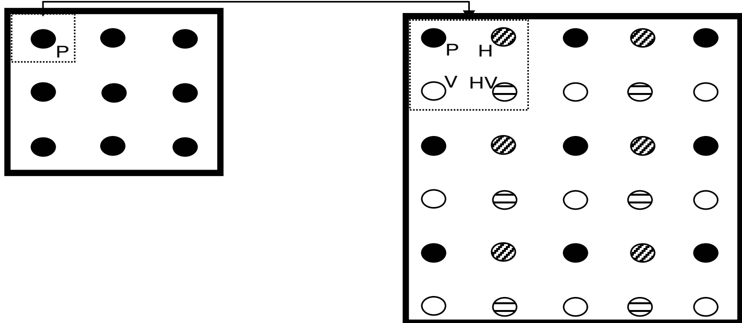


Figure 20. Representing the interpolation of a frame

The value of the H, V and HV pixels are computed using a six tap Finite Impulse Response (FIR) filter algorithm [5], based on a set of surrounding P pixels.

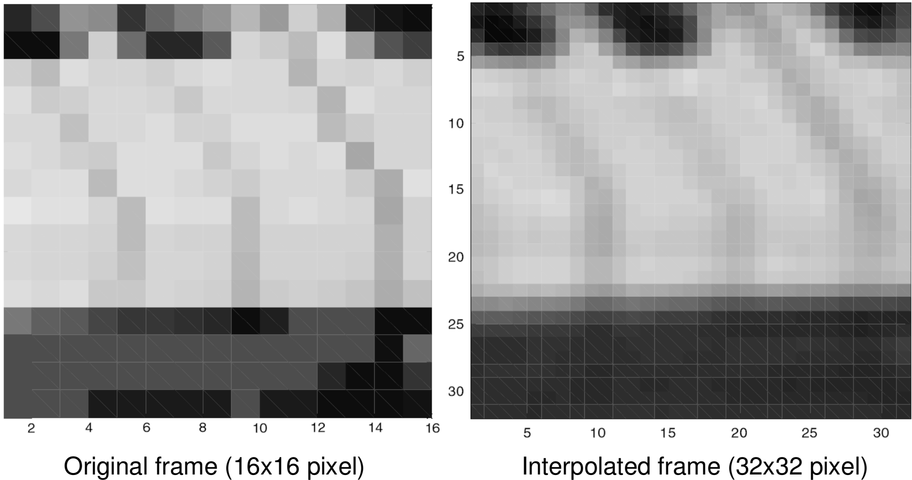


Figure 21. Close look at an original frame, and the resulted interpolated frame

After this, the MotionEstimation takes each MacroBlock from the current original frame, and searches for an equivalent in the previous decoded frame. Using this current MacroBlock, the Motion Estimation compares it with multiple 16x16 regions into a specific search area from the previous decoded frame, until it finds an acceptable similar one.

In the example presented in Figure 22, the current MacroBlock with coordinates (32,48) in the current frame, is compared with a multitude of 16x16 regions in the previous decoded frame, into the search area between the coordinates (16,32) and (64,80). In the current example, the search area is (±16, ±16). However, this aspect depends on the will of the user. In the end, the Motion Estimation will find that the 16x16 region with the coordinates (24,38) in the decoded previous image is similar with the current MacroBlock. The motion vector is obtained by computing the difference between the coordinate of the current MacroBlock (32,48) and the coordinate of the similar 16x16 region (24,38) from the previous decoded image. The resulted motion vector is (8,10).

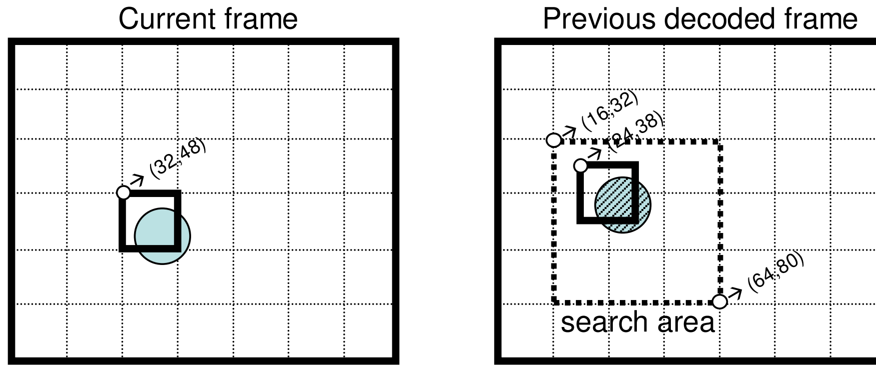


Figure 22. Representing the Motion Estimation for a MacroBlock

The similarity factor between two 16x16 regions is computed using the Sum of Absolute Differences (SAD) formula (Figure 23). The bigger is the SAD value, the bigger is the difference between these two regions.

$$SAD = \sum_{i=0,\, j=0}^{i<16,\, j<16} ABS(C_{ij} - P_{ij})$$

Figure 23. SAD formula used to compute the similarity between two 16x16 regions

The MotionEstimation computes the SAD value for multiple coordinate's positions, until it finds the smaller one. When the resulted SAD value is smaller than a given threshold, the Motion Estimation considers that it found the most similar 16x16 region with the one contained in the current MacroBlock. By default, the MPEG4 encoder specifications

propose the value of this threshold to be 512. However, this can also be adjusted depending on the will of the user.

There are many methods of choosing which 16x16 regions from the previous decoded frame to be compared with the current MacroBlock. FullSearch [5] method is the simplest one, and it compares the current MacroBlock with all the possible 16x16 regions from the search area. This method is slow, even if it provides the best results. However, different other methods were developed. The most popular ones are the DiamondSearch [13] and SquareSearch [14]. These are based on comparing only some of the possible 16x16 regions from the search area, based on algorithmic decisions. They are significantly faster than the FullSearch, and provide appropriated results with the FullSearch method.

Identically, the Motion Estimation function will compute the motion vectors for all the MacroBlocks from the current frame. The MacroBlocks to which was found a similar 16x16 region in the previous decoded image will be encoded in the following steps as P. The MacroBlocks for which was not found any similar 16x16 region in the previous decoded image will be encoded in the following steps as I, using the principle described earlier in this document.

In addition, in the moment the Motion Estimation finds that a significant number of MacroBlocks has to be encoded as I, the Motion Estimation will stop, and it decides that all the MacroBlocks will be encoded as I. This feature is called “scene change detection”.

2.3.4 Describing the Motion Compensation function

After the MotionEstimation had computed the motion vectors for each of the frame’s MacroBlock, the next step is to find for each of the MacroBlocks from the current frame, the remaining differences (“residues”) compared with its corresponding best match 16x16 region from the previous frame. For the example presented in Figure 22, the resulted difference for the MacroBlock with coordinates (32,48) is presented in Figure 24. The values stored on the Difference MacroBlock are “negative” values of the actual differences.

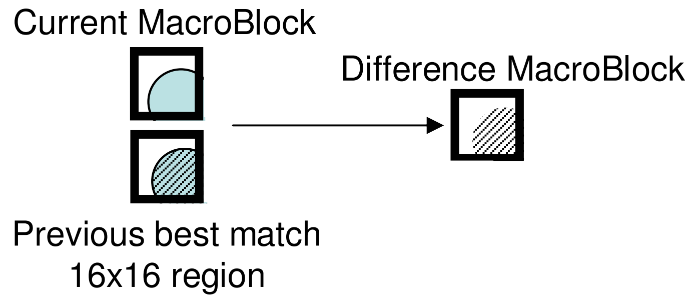


Figure 24. Determining the Difference MacroBlock by the Motion Compensation function

From this moment, the MPEG4 video encoder considers that this difference is in fact the original current image received.

2.3.5 Describing the FDCT, Quantization, DeQuantization and IDCT functions for the P MacroBlock

The algorithm takes each Difference MacroBlock, and transforms it from image into frequencies, using the FDCT function (presented previously in the document). The resulted DCT values are quantized, and sent to the VLC to be stored into the MPEG4 bitstream.

An important difference of encoding a MacroBlock as P, compared with the case of encoding as I, is that it may be declared as *skipped* MacroBlock if the values resulting after the quantization are only zeros. In this case, the MPEG4 video encoder will inform the MPEG4 video decoder (through the bitstream) that this MacroBlock does not have to be decoded, because it has not changed since the previous frame. This is often the case when the movie contains static sequences.

In addition, just like the I MacroBlock, the MPEG4 algorithm is decoding these quantized values, in order to build the decoded current frame to be used later when encoding the next frame. For this purpose, the quantized values are dequantized, obtaining the decoded DCT values. These DCT values are decoded using the IDCT function, in order to obtain the decoded MacroBlock.

2.3.6 Describing the Reconstruction function

At this moment, the current MacroBlock, which contains only differences ("residues") was encoded and decoded. Therefore, the last step is to reconstruct the current MacroBlock, using the decoded MacroBlocks (obtained after the IDCT) and the previous best match (Figure 25). This phase is doing the inverse of the MotionCompensation function.

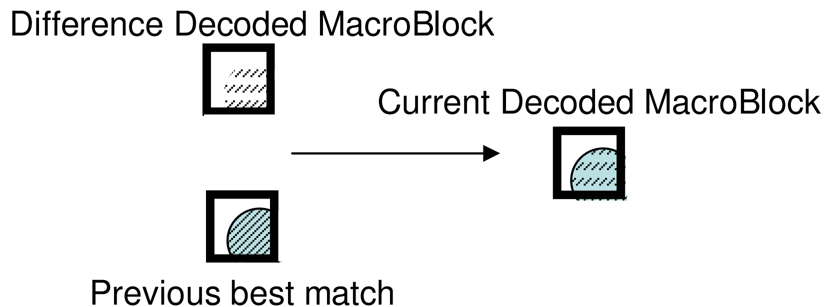


Figure 25. Obtaining the Current Decoded MacroBlock by the Reconstruction function

It can be noticed by comparing the Difference MacroBlock from the Figure 24 with the Difference Decoded MacroBlock from the Figure 25, that this decoded MacroBlock is a little bit different from the initial difference MacroBlock. This is due to the quality degradation resulting after encoding (FDCT,Quantization) and decoding (DeQuantization, IDCT). By using this Difference Decoded MacroBlock and its previous best match MacroBlock (used also by the MotionCompensation), the Current Decoded MacroBlock is reconstructed. This MacroBlock is also a little bit different from the Original Current MacroBlock (see Figure 24).

In conclusion, after processing all the MacroBlocks of the frame, the resulting decoded current image contains some differences compared with the original image, caused by the quality degradation of all the Difference MacroBlocks. However, this altered image is exactly the one that will be obtained by the MPEG4 video decoders, and displayed on the screen to the spectator.

2.4 Detailed description of the VLC and Bitrate Controller

2.4.1 Description of the compression phase done by the MPEG4 video encoder

After encoding an I or P frame, the size of all the results are more than twice bigger than the original input image. One of the reasons is the fact that the original input image is represented on 8 bits/pixel, while the encoded results for each of the image pixels are on 16 bits/pixel. Even more, there are extra motion vectors for each MacroBlock, plus 6 AC predictors on 16bits/predictor for each MacroBlock, skipped MacroBlock flags, MacroBlock encoding mode (I or P) flag, etc.

However, the great majority of these values are containing many 0 values. This is why, this kind of results is ideal for compression. The MPEG4 video encoder is compressing these results using the VLC function [5]. The final output of the VLC function is a well-defined (and strict) result, called MPEG4 bitstream. This MPEG4 bitstream will be used later by an MPEG4 video decoder, in order to decode the movie. The block diagram of the VLC function is presented in Figure 26.

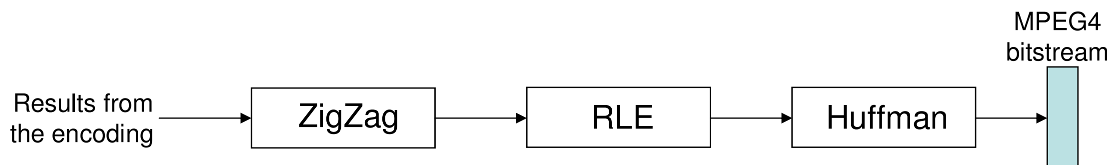


Figure 26. Block diagram of the VLC function

For each of the encoded MacroBlocks, is applied the ZigZag function (a.k.a. Reordering) to reorder the values of this encoded MacroBlock into a well-defined order. The results are then compressed using the RLE (Run-Length Encoding) algorithm, and finally compressed even more using the Huffman algorithm. This phase is executed identically for all the other encoded MacroBlocks, and the result will be the MPEG4 video encoder.

2.4.2 Applying ZigZag reordering on a MacroBlock

Currently, the encoded MacroBlocks are in two dimensional matrix forms. Because the following RLE and Huffman compressions are using as input the data in a vector form, it is imperative to reorder the values of the encoded MacroBlock under the form of a vector.

The ZigZag function is in charge of doing this reordering. It is applying the reordering for each of the 6 MicroBlocks associated for the current MacroBlock (Figure 27), resulting 6 vectors each of them with 64 values.

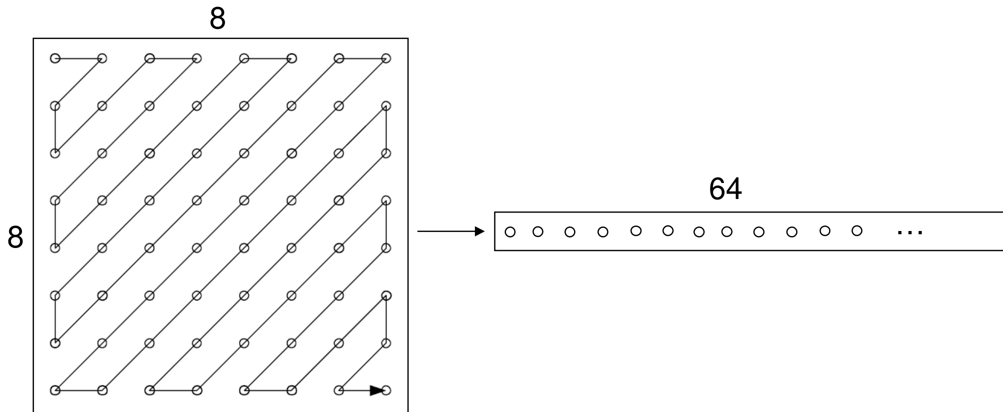


Figure 27. Reordering a MicroBlock using the ZigZag function

The ZigZag reordering is used, because the values stored in an encoded MicroBlock contains the highest values at the top-left corner, and the lowest at the bottom-right corner. This is caused by the FDCT function, as was already presented earlier in this document. Therefore, by doing the ZigZag reordering, the resulted 64 values vector will contain the highest values at the beginning, and lowest values (many zero values) at the end. This way, it is assured to have in the resulted vector, consecutive regions containing the same values. This aspect is highly exploitable by the following RLE and Huffman compression phase.

2.4.3 Compressing the ZigZag-ed MacroBlock using RLE and Huffman compression

Using the resulting 6 vectors, the next step is to compress them using the RLE compression algorithm [15]. The basic idea of this algorithm is to use sequences of consecutive identical values, and to code them as two values: number of repetition, and value.

The resulting compressed results are compressed even more, using the Huffman compression algorithm [16]. The idea behind this algorithm is to assign to each of the values from this vector, a symbol (on bits) based on the probability of occurrence of each value. The problem with using the original Huffman algorithm is that all these coding symbols

(called dictionary) have to be calculated, and stored into the resulted compressed result, to be used by the decoder. This leads to extra computations, and decreased compression ratio. To avoid this, the MPEG4 uses a standard dictionary, called VLC tables [5]. This way, only the compressed results have to be stored. The MPEG4 video decoder will use the same VLC tables when decoding the bitstream.

2.4.4 Structure of the MPEG4 bitstream

In order to ensure that the compressed results will be decodable by any MPEG4 video decoder, the MPEG4 specifications impose a strict result format, called MPEG4 bitstream. This is structured on multiple levels: movie, frame and MacroBlock (Figure 28).

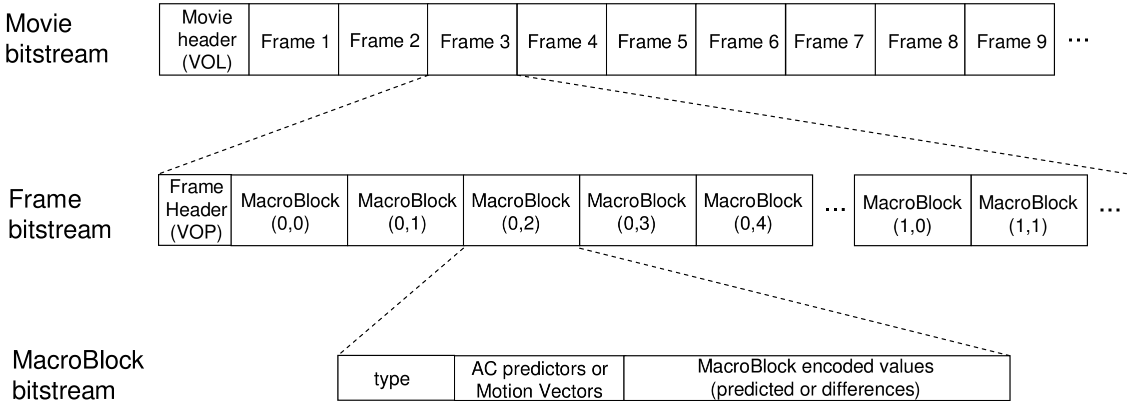


Figure 28. MPEG4 bitstream structure for movie, frame and MacroBlock

At the beginning of every movie bitstream, there is a header called VOL (Visual Object Layer) header. This is storing basic information regarding the movie: resolution, frame_rate, type of motion estimation used (full pixel or half-pixel), etc.

This header is followed by the bitstream associated to each of the movie's frames. This bitstream is initiated using a VOP (Visual Object Picture) header. This header is storing basic information regarding the encoding phase of the current frame: *quanta*, frame type (I,P), etc. This header is followed by the bitstream associated to each of the frame's MacroBlocks, in a sequential order.

Each MacroBlock bitstream is initiated by the type of this MacroBlock (I, P, or *skipped*). Then, depending on their type, the next pieces of information are the motion vec-

tors (P MacroBlock) or AC predictors (I MacroBlock). In the end, the compressed encoded MacroBlock values are stored. It is obvious that, if the MacroBlock type is *skipped*, there will be no informations stored for the current MacroBlock bitstream, except the type of it.

An important aspect is that all these components forming the MPEG4 bitstream are concatenated at bit level, and not byte level. This aspect will be an important issue that had to be taken into account later in this document.

2.4.5 Quantization adjustment by Bitrate Controller for the next frame encoding

After each frame encoding and compression, the result is the MPEG4 bitstream for that frame. The size of this bitstream is variable, depending on the nature of the movie.

In case the size of this bitstream is bigger than a desired value (specified by the user), the Bitrate Controller will increase the *quanta* value which will be used when encoding the next frame. This way, more quality degradation will occur during the next frame encoding, but the size of the bitstream for that next frame will be lower.

In case the size of this bitstream is lower than a desired value (specified by the user), the Bitrate Controller will decrease the *quanta* value which will be used when encoding the next frame. This way, less quality degradation will occur during the next frame encoding, but higher will be the size of the bitstream for the next frame.

As a result, when encoding an entire movie, the user has to specify its targeted average bitstream size for a frame. The Bitrate Controller will adjust the *quanta* value “on the fly’ during the encoding, in order to keep the bitstream for each frame as close as possible to the user’s demand.

2.5 MPEG4 algorithm parameters

To compress a movie, the MPEG4 video encoder requires multiple parameters.

2.5.1 Image resolution parameter

This parameter specifies the *height* and *width* of the video's image. Since the MPEG4 algorithm is processing the image at *MacroBlock* level (16x16), the *height* and *width* of the image has to be a multiple of 16. Therefore, the MPEG4 algorithm is able to compress any video with an image resolution multiple of 16. However, some commonly used resolutions are presented in Figure 29.

Resolution name	Width	Height
QCIF	176	144
QVGA	320	240
CIF	352	288
VGA	640	480
4CIF	704	576
HDTV 720	1280	720
HDTV 1072	1440	1072

Figure 29. List of different standardized resolutions

In case the image resolution is not a multiple of 16, the algorithm is doing some pixels stuffing, by copying the border pixels, to obtain an image resolution multiple of 16.

2.5.2 Frame rate parameter

This parameter specifies the number of frames existing for one second of movie. The higher the *frame rate*, the smother (and more visually comfortable) will be the moving sequences in the movie, but also the image quality will decrease (because more frames have to be stored into 1 second of bitstream). Figure 30 shows some common *frame_rates*.

Frame rate	Example of application domain
1	Long term video surveillance
15 or 16	Video conferences, Mobiles
24 (PAL) 25 (SECAM)	TV (Europe)
29.996 (NTSC)	TV (USA & Japan)
50	Cinema

Figure 30. List of frame rates for different examples of application domains

2.5.3 Keyframe rate parameter

This parameter is specifying the number of frames between two I frames. In case the *keyframe rate* is 100, the compressed movie will contain 99 P frames between each two I frames. The higher is the value of the *keyframe rate*, the higher will be the resulting compression factor, fewer computations will be required, but lower will be the "tolerance" of the decoder to errors inside the bitstream will be lower, and it will take more time to seek into the movie. In practice, the most common values for the *keyframe rate* are between 5 and 300. For example, by default, the OpenDivX uses the *keyframe* 5, the DivX uses 100, and the XviD uses 300. However, in each of these cases, the value for the *keyframe* can be changed by the user.

2.5.4 Compression rate (bitrate) parameter

By adjusting the compression rate parameter, it is possible to demand from the encoder a specific compression factor. The higher the compression rate will be, the lower the image quality of the resulted encoded movie will be, but fewer computations are required. There are 2 methods of demanding a desired *compression rate*:

1) *variable compression*: in this case, the user specifies to the encoder that the compression rate is not important for him, and it can vary in time. The only thing important is to obtain a specific image quality for the encoded movie. This aspect is controlled by adjusting the interval between which the *quantization* factor can change. Normally, the *quantization* factor is varying between the values 2 (highest quality) and 31 (lowest quality). By specifying that the *quantization* factor has to be 2 all the time, the encoder will compress the movie obtaining the higher quality, but will also obtain the lowest compression factor which will vary (depending on the movie images).

2) *targeted compression*: in this case the user will specify the targeted size of the resulted bitstream, also known as *bitrate* factor. Depending on the actual resulted bitstream size, the encoder will adjust automatically the *quantization* factor in order to obtain the desired bitstream size. For example, in case of a QCIF movie at 25 frames/sec, the uncompressed size of 1 second in RGB format is 1.900.800 bytes. By adjusting the *bitrate* to 700kbits/sec, the resulting compression factor will be approximately 1:21. The resulted

image quality will hugely depend on the image complexity of the movie. This method is usually used when a specific storage support is targeted for the encoded movie (i.e. CD-ROM, DVD).

2.5.5 Quantization range parameter

This parameter represents the interval through which the *quanta* value can be adjusted by the encoder. By default, in any encoders (OpenDivX, DivX, XviD), this interval is between 2 and 31. It is important to mention that the use of value 1 is not allowed, since the standard imposes to save into the bitstream the value of the *quanta* minus 2. The decoders will read this value, and increment it with 2, in order to obtain the real *quanta* value.

Thus, based on the value of the targeted bitrate parameter (see previous sub-chapter) and the resulting bitstream size for the current frame, the MPEG4 encoder (most exactly the BitrateController function) will readapt the *quanta* value for the next frame. This value will be forced to stay in the interval specified using this quantization range parameter.

The smaller this interval is, the more difficult it will be to keep the size of the compressed movie into the required compression rate. However, this will avoid significant quality variations during the movie, since the *quanta* value is the one that directly affects the quality of the compressed movie.

By setting this interval to a single value, the quanta value is kept constantly to this single value. This feature practically eliminates the use of compression rate parameter, since the *quanta* will not change, in order to obtain the required compression rate.

2.5.6 Rate control delay parameter

This parameter adjusts the number of frames between which the *quanta* value will be readjusted. Choosing the value of the rate delay to 1 will force the algorithm to adjust the *quanta* value after each frame. The disadvantage is that the quality of the movie might change after each frame, which might lead to a visually unpleasant effect. The advantage is that, the size of the compressed movie is correctly kept in the targeted size.

By default, the OpenDivX uses for the rate_delay a value of 10, DivX is using 30, and the XviD is using a value equal with the frame_rate (i.e. 25).

2.5.7 Computations precision parameter

This aspect is related to the precisions of the computations done during the encoding phase. The higher the computation precisions are, the higher the quality and compression rate of the encoded movie will be, but more computations and memory will be required. For example, in case of the *MotionEstimation* phase, different computation precisions can be set by choosing between different motion estimation algorithms to be used (full search, halfpixel, earlystop, diamond, square), each of them with its own complexity/precision. In case of *Quantization* and *DeQuantization* phases, it can be chosen between different quantization methods (h263 quantization, mpeg4 quantization).

2.5.8 Scene change detection parameter

This parameter is a Boolean parameter with which can be activated or deactivated the use of the scene change detection feature.

It is common during a movie that the current frame contains a different scene, compared with the previous frame. This can be determined during the *MotionEstimation* function. Normally, when a scene is changed, a large number of MacroBlocks from the current frame will be different from the ones existing in the previous frame. Thus, the *MotionEstimation* will "mark" these MacroBlocks to be encoded as I (new MacroBlocks), even if the current frame is encoded as P frame. When the number of I MacroBlocks bypasses a specified threshold (the standard proposes half the number of total MacroBlocks from the image), the *MotionEstimation* will stop its computations, and the MPEG4 encoder will do a forced compression of the entire frame in I mode.

In other words, when using this feature, the algorithm will decide to use I frames when scene changes are detected, along with the I frames imposed by the key_frame parameter.

2.5.9 Compression mode parameter

There are 2 ways an image can be treated: *progressive* or *interlaced*.

1) *progressive*: the entire image will be compressed. The advantage is the high quality of the resulting encoded image (Figure 31, left). The disadvantages are the high computations and memory required to compress it.

2) *interlaced*: only every 2 lines of the image is compressed. For example, for the 1st image are compressed only the Odd lines, and for the 2nd image are compressed only the Even lines. The advantages are the low computations and memory required (twice fewer than the *progressive* mode). The disadvantage is the low quality of the resulted encoded image, which can be unacceptable in case of high movements (Figure 31, right).

Figure 31. Image encoded in progressive mode VS. Image encoded in interlaced mode

2.5.10 Algorithm type parameter

The MPEG4 encoder algorithm is not the only video encoder algorithm that compresses an image using the concept of MacroBlocks. Several other algorithms are using similar approaches. Historically, the first algorithm that used this concept was the MPEG1 encoder algorithm (Figure 32). After that was developed the MPEG2 algorithm, which contains more features compared with MPEG1. The H263 algorithm followed the MPEG2, bringing additional new features. The MPEG4 encoder algorithm appeared after the H263, and its complete name is MPEG4 Part 2. Additional other algorithms preceded the MPEG4.

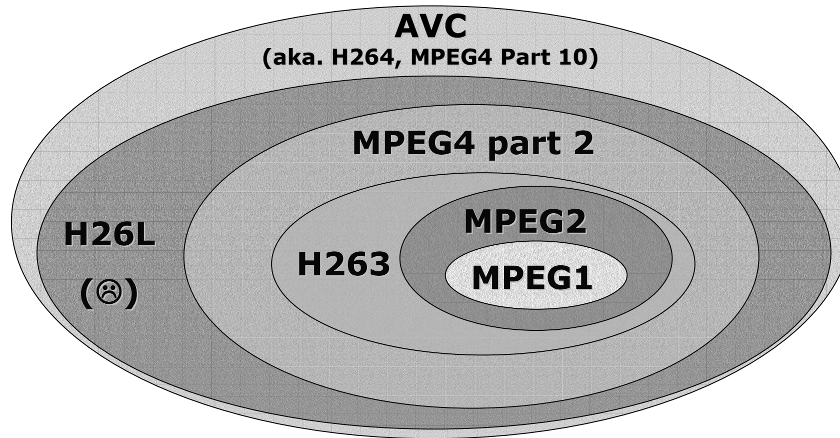


Figure 32. Representing the algorithms inclusion based on their capabilities (features)

Every new algorithm was in fact an "extension" of an already existing algorithm. For example, the MPEG4 part 2 encoder algorithm already contains the features provided by the MPEG1, MPEG2, H263 algorithms, plus some new specific features. In Figure 33 are presented the most important features used for each algorithm.

In our work, we use the MPEG4 part 2 algorithm (presented previously in the document). With this algorithm it is possible to encode the movie also in MPEG2 and MPEG1 modes. This is possible by "deactivating" in the algorithm some features specific for MPEG4, and use only the ones specific for MPEG2 and MPEG1. For example, one of them is related to the IntraPrediction. If the MPEG4 uses the left, left-top, top and right-top MacroBlocks to compute the "fading" for the current MacroBlock, the MPEG2 encoder uses only the left MacroBlock to compute the "fading" for the current MacroBlock. However, some other functions are identical. For example, the MPEG4, MPEG2 and MPEG1 use the same FDCT and IDCT functions.

With the *Algorithm type parameter*, it is possible to mention which algorithm to be used. Based on this value, some features will be used, or will be "skipped". Currently, our work supports the MPEG4, MPEG2 and MPEG1 algorithms.

We have left as future work the support for the H263 and H264 algorithms. It is important to mention that the H26L algorithm was never completely developed by the JVT (Joined Video Team). During its developing process, the JVT group and the MPEG group merged, in order to develop the AVC algorithm (a.k.a. H264, MPEG4 part 10).

	MPEG1	MPEG2	H.263	MPEG4 (part 2)	AVC (a.k.a H.264, MPEG4 part 10)
Targeted Bitrate (Mbs)	**8**	**4.5**	**1.8**	**0.7**	**0.3**
Motion estimation	**0-16** (fullpel) **SAD** (macro 16x16)	**0-8-128** (fullpel,halfpel) **SAD** (macro 16x16)	**0-8-128** (fullpel,halfpel) **SAD** (macro 16x16)	**0-16-128** (fullpixel,halfpel, quarterpel, inter4v) **SAD** (micro 8x8, macro16x16)	**0-16-128** (fullpixel, halfpel, quarterpel, inter4v) **SAD** (micro 4x4, macro16x16)
B frames	**-**	**X**	**X**	**X**	**X**
Motion prediction &compensat.	No prediction, only compensation	**6 DC** (left MB)	**6 DC** (left,top-left,top,top-right MB)	**9 DC,AC** (left,top-left,top,top-right MB)	**6 DC,AC** (left,top-left,top,top-right MB)
FDCT / IDCT	**8x8**	**8x8**	**8x8**	**8x8**	**4x4,4x8,8x4,8x8,16x8, 16x16**
Quantization	**MPEG**	**MPEG**	**H263**	**MPEG,H263 adaptative**	**MPEG,H263,H264 adaptative**
ZigZag	**D**	**D**	**D**	**D,V,H**	**D,V,H**
Compression	**VLC**	**VLC**	**VLC**	**VLC**	**VLC/CABAC**
Perform. (Gops)	**20**	**40**	**78**	**133**	**191**

Figure 33. Listing the features supported by each video compression algorithm

2.6 Conclusions

The objective of this chapter is to introduce the MPEG4 video encoder algorithm. The algorithm receives as input an uncompressed movie, and obtains an MPEG4 bitstream containing the compressed movie with some quality degradations. The different steps used to compress the movie were detailed. The study of this algorithm allowed us to understand its high complexity. Additionally, the algorithm behavior is dynamic, because it depends on the input video and the used algorithm parameters. The amount of required computations depends on the targeted application domain, since each of these applications require the use of different configurations for the MPEG4 encoder algorithm. To obtain efficient results, the algorithm has to be correctly configured and optimized for the targeted application and the used architecture. This requires a significant amount of work time. However, the average time to market for the video applications should not be more than some months (half year). To cope with this problem, the MPEG4 video encoder algorithm has to be eas-

ily adaptable for different architectures containing different number of processors. This can be done using an MPEG4 video encoder algorithm that supports different computations distributions on multiprocessor architectures, using parallel and pipeline execution schemes.

3 Flexible modeling style to represent the Combined Algorithm/ Architecture Model for MPEG4

This chapter presents a flexible modeling style to represent the algorithm/architecture model for a parallelized and pipelined MPEG4 encoder. First, this chapter presents the techniques used to parallelize and pipeline the MPEG4 encoder. Then, the chapter presents the targeted Flexible Architecture with 2 SMPs for MPEG4 encoder. Then, the chapter focuses on the Combined Algorithm/Architecture Executable Model, used to describe at high-level of abstraction this Flexible Architecture with 2 SMPs. Since different architectures are required, for different algorithm and architecture parameters, this chapter presents a flexible modeling style to obtain automatically different Combined Algorithm/Architecture Executable Models starting from a unique Flexible Algorithm/Architecture Model for MPEG4.

3.1 Parallelism/pipeline support for the MPEG4 video encoder algorithm

3.1.1 Amount of computations required by the MPEG4 encoder

The original MPEG4 video encoder is a sequential algorithm, computing sequentially each of the frame's MacroBlocks, either during a single function (i.e. Motion Estimation), or during multiple functions (i.e. FDCT, Quantization, DeQuantization, IDCT). The MPEG4 video encoder is a single task application. Because of the high complexity of the MPEG4 video encoder, the amount of computations required by this task is high.

There are multiple factors which are affecting this amount of computations, like: image resolution, spatio-temporal complexity of the movie, motion estimation search area, desired computation precisions, etc. Figure 34 shows an approximate amount of computations required to encode different video resolutions, at 25 frames/sec, using 16x16 motion search area and maximum computation precision.

Resolution	MIPS
QCIF	280
QVGA	800
CIF	1120
VGA	3200
4CIF	4500
HDTV 720	10200
HDTV 1072	17000

Figure 34. Computations required by MPEG4 video encoder for different video resolutions

These values may become even bigger, in case of increasing the motion search area. For example, for HDTV 1072 video resolution at 100 frames/sec and using full motion search area (128x128), the amount of required computations is close to 32TIPS (Tera Instructions Per Second). In case of targeting this algorithm to be implemented on a chip, there is no processor today that can provide such an amount of computation power. This computation power may be a limitation even in case of simple QCIF resolution. For example, inserting a processor running at 300MHz into a mobile phone is unacceptable in term of power consumption.

3.1.2 Objectives

Our objective is to implement an MPEG4 encoder algorithm that can easily be adapted for different configurations of computations distribution. This allows the implementation for reasonable frequencies. Moreover, we want the same approach to be applicable for different other video encoding algorithms (i.e. MPEG1, MPEG2).

We intend to do this by using 2 computations distribution techniques: computations distribution for parallel execution, and computations distribution for pipeline execution. Additionally, the resulting algorithm should be as flexible (modifiable) as possible, in order to easily adapt it for multiple computations distribution configurations (i.e. different levels of parallelism).

3.1.3 State of the art

There are several existing works for MPEG4 video encoders that are using the concept of computations distribution over multiple processors architectures. In the following sections, we will present the examples that are the most important for our work.

a) In [55], the MPEG4 algorithm is implemented using a SW approach. The complete MPEG4 algorithm is mapped on a single processor, more exactly a TM1300 CPU. This means that, in this case, the algorithm computations are not distributed over multiple processors (since there is only one processor). Thus, in case of increasing the computation requirements of the MPEG4 algorithm (i.e. by increasing the video resolution, or frame rate), the processor may become saturated. The only solution would be to boost the speed of this processor (i.e. increase its clock frequency, HW accelerators).

   As a result, the MPEG4 algorithm cannot be used for applications that require a large amount of computations, because that would require over-clocking this single processor to unacceptable clock frequencies. These applications are home-cinema, video recording, and mobile telecommunications requiring low clock frequencies.

b) In [19], the MPEG4 video encoder is implemented using a mixed HW/SW approach, where the MPEG4 algorithm is distributed over multiple processors, and

some computational intensive functions are implemented into a shared HW accelerator. The algorithm behavior is pipelined, by distributing the computations over multiple processors (4 ARM9 CPUs running at 200 MHz). This distribution is achieved by mapping one or more functions from the MPEG4 video encoder on different processors that communicate with each other (i.e. the MotionEstimation is mapped on a processor, the FDCT in another processor, etc). Thus, pipeline approach is used, even if overall the MPEG4 encoder still has a sequential behavior.

The problem of this approach is that, this pipeline can become saturated when increasing the computation requirements of the MPEG4 video encoder, which leads to the need of HW accelerators. Additionally, this pipeline is not "perfect", because generally speaking, the MPEG4 encoder is an output dependent application. For example, the MotionEstimation (which is the first function executed during a frame encoding process) has to wait for the entire pipeline to be "consumed". This is due to the fact that the MotionEstimation requires the current frame, but also the previous decoded frame. Moreover, this previous decoded frame is obtained only at the end of the entire encoding and decoding process, after the entire pipeline is "consumed".

Any parallelism approach is done at function level, mainly using HW instructions. For example, a SIMD [52] approach can be used for the FDCT function. This will execute in parallel the same FDCT instructions for each of the 6 MicroBlocks contained in the current MacroBlock.

The problem with this kind of parallelization approach (based on HW instructions) is that it drastically reduces the portability of the algorithm on other types of architectures. Additionally, the parallelized functions are too small (fine grain parallelism). When targeting a HW implementation, this is ok. However, in case of targeting this application for CPUs, the result will be a large number of required context switches, which might drastically reduce the performance of the application.

As a result, this MPEG4 algorithm cannot be used for applications requiring a large amount of computations, or using an architecture with more than 4 processors. Such

applications are home-cinema, video surveillance, mobile telecommunication (since the 200MHz required clock frequency is unacceptable in such applications).

c) In [20] and [21], the MPEG4 video encoder algorithm, and the H.26L video encoder algorithm are distributed over multiple processors, using a SMP [53] approach. The idea of these approaches is to instantiate multiple tasks of MPEG4 encoders (or H26L encoder), each of them operating over a different small area of the image. For example, if the algorithm will be parallelized into 4 tasks, the original image will be split into 4 parts (areas), and each task will operate over one of these areas. Thus, the application has a parallelized behavior with a single SMP.

   The advantage of their approach is its scalability. It is quite easy to adapt the algorithm for different other levels of parallelization (i.e. for 9 tasks running in parallel, each of them operating on a ninth of the image).

   The disadvantage of their approach is the fact that the compression phase (VLC) is executed during the encoding phase (MotionEstimation, FDCT, Quantization, Dequantization, IDCT, etc). For example, when the MotionEstimation determines the motion vector for a MacroBlock, this motion vector is immediately compressed into the bitstream using the corresponding VLC instructions. The same thing happens practically after each function. As a function finishes its computations, some of its results are immediately stored into the bitstream. In other words, the encoding phase and the compression phase are merged into a single phase. Since the encoding phase contains an algorithm whose input is dependent on its output (previous encoded and decoded image), no pipeline is allowed. To conclude, the approaches presented in [20][21] are targeted only for parallel execution scheme (single SMP), and pipelining is not possible without changing the algorithm.

   Another disadvantage of their approach is the memory requirements. The compression phase (VLC) requires a large amount of memory to store its standard Huffman tables (compression dictionary), tables needed to compress the data into the bitstream. In case of MPEG4 encoder, the size of these tables is approximately 100Kbytes. Since the compression phase (VLC) is not separated by the encoding phase, after the algorithm distribution, they will have the same number of phases for

compression and encoding. Thus, these standard Huffman tables might need to be multiplied, to have 1 instance for each VLC (unless shared memory architecture is used). Additionally, the compression phase represents maybe 10% of the entire algorithm computation requirements. This means that, the compression phase is in “idle” mode for approximately 90% of the algorithm’s execution time. Therefore, this algorithm requires a significant amount of data and code memory.

Additionally, in their approach, the image is split into completely different areas, areas that have no common image regions between them. This might impose some detection problems during the MotionEstimation in case of MacroBlocks which are moving during a movie sequence, from an area (belonging to one task) to another area (belonging to another task).This might decrease the quality of the resulting compressed movie.

Additionally, in [20] and [21], their approaches are demonstrated to work on only one video encoding algorithm. This raises the following question: “Is it possible to apply this approach for different other encoder algorithms?”

As a result, this MPEG4 algorithm cannot be used for applications requiring low memory, or high quality compressed movie. Such applications are mobile telecommunication (low memory), respectively home-cinema and video surveillance (high quality movie).

3.1.4 Contributions

The contributions presented in this chapter, is an MPEG4 video encoder algorithm that can be easily adapted for multiple applications. This algorithm supports different computations distributions over multiple processors, using different levels of parallelism and pipeline behavior. Additionally, the resulted algorithm consumes less memory than the approaches presented in [20][21] by using a smaller number of VLCs. In order to represent this algorithm on an MP-SoC architecture, we propose a flexible modeling style used to describe the *Combined Algorithm/Architecture Executable Model* for MPEG4, starting from a unique *Flexible Algorithm/Architecture Model* for MPEG4. The same approach can

be extended to other video encoder algorithms (i.e. MPEG1 and MPEG2) only by setting the *Algorithm Type parameter*.

3.2 Proposed MPEG4 video encoder supporting different parallel/pipeline configurations

3.2.1 Presenting the approach used for inserting parallelism and pipeline support into the MPEG video encoder

In our approach, we use computations distribution using SMP approach, for the execution in parallel of heavy computations. Compared with the cases presented in [20] and [21], which used only 1 SMP, in our approach we use 2 separated SMPs: one SMP for the encoding phase, and one SMP for the compression phase. Additionally, these two SMPs will be executed in a pipeline. This section presents in detail how this is achieved.

All the functions forming the MPEG4 video encoder algorithm were grouped into 2 big tasks: *MainDivX task* and the *VLC task* (Figure 35). The *MainDivX task* represents the encoding process of an image, while the *VLC task* is in charge of compressing the results obtained by the *MainDivX task*.

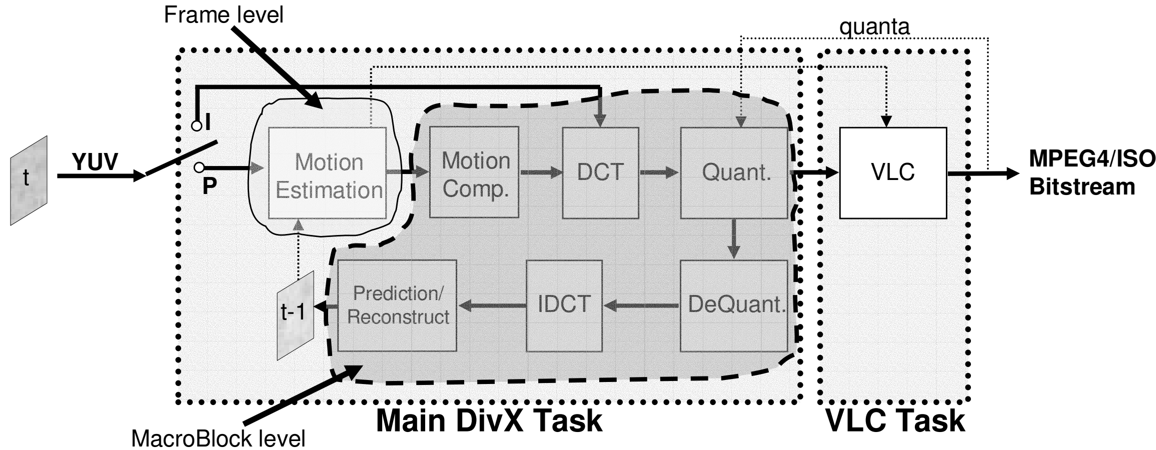


Figure 35. Partitioning the MPEG4 video encoder in multiple tasks

Compared with the approaches presented in [20][21], where the *MainDivX task* and the *VLC task* are merged into a single task, in our approach, these two tasks were sepa-

rated. This was possible in the following way: every time a function from the *MainDivX task* obtained some results which should be inserted in the bitstream by the *VLC task*, we are not inserting them immediately, but we are memorizing (buffering) them. After finishing the *MainDivX task* to encode a MacroBlock, the *VLC task* will start to compress all the buffered results. While the *VLC task* is compressing all these results, the *MainDivX task* can continue to encode the next MacroBlock.

Thus, the *MainDivX task* and the *VLC task* have a pipelined behavior, and the communication between them is done using 2 buffers. While the *MainDivX task* is writing the results in buffer 1, the *VLC task* is reading from buffer 2 the previously computed result of *MainDivX*. In the next step, the *MainDivX task* will write the results in buffer 2, while the *VLC task* will read from buffer 1. This technique is called “flip-flop” [56]. As a result, compared with the approaches from [20][21], we have succeeded to separate the compression phase by the encoding phase.

The reason why the *MainDivX task* was not distributed into more pipelined tasks (like in [19]), is due to the fact that the input of the entire *MainDivX task* is output dependent. As it can be seen in Figure 35, the MotionEstimation requires along with the current frame(t) also the previous encoded and decoded frame(t-1). However, in the current status, the entire MPEG4 encoder algorithm has a pipelined behavior with two stages: *MainDivX* and *VLC*.

In order to insert parallel behavior into the algorithm, we have distributed the computations of each of these tasks using a SMP approach:

a) The SMP for the MainDivX task is obtained just like in [20][21], by dividing the original image into multiple smaller areas, and instantiating the same number of *MainDivX tasks*, each of them in charge of processing the corresponding area.

b) The SMP for the VLC task is obtained by the instantiation of multiple identical VLC tasks, each of them compressing the results from the *MainDivX tasks*, more exactly the encoded MacroBlocks obtained by the *MainDivX tasks*. Each of the VLC task will be in charge of compressing some of the encoded MacroBlocks. For example, if we use 2 *VLC tasks*, each of them will be in charge of compressing only half the

number of total MacroBlocks encoded by all the *MainDivX tasks*. To do this, we had to adapt the code of the *VLC task* to execute at MacroBlock level. Additionally, we have exploited the fact that the *VLC tasks* do not have to wait for the *MainDivX tasks* to finish processing the entire image. By adapting the *VLC* tasks to work at *Macro-Block* level, once a *MainDivX task* finished encoding a MacroBlock, the corresponding *VLC task* can start to compress it, while that *MainDivX task* continues to process the next MacroBlock.

This way, the resulting MPEG4 video encoder algorithm contains 2 SMPs which are executing as a pipeline at MacroBlock level. The level of parallelism into each of these SMPs can easily be adapted for different configurations. For example, in case of the SMP for the *MainDivX*, changing from 4 *MainDivX tasks* running in parallel to 9 *MainDivX tasks* running in parallel can be made by dividing even more (9 areas) the original image, and instantiating 9 *MainDivX tasks* for each of the resulting image areas. In case of the SMP for the *VLC*, changing from 2 *VLC tasks* running in parallel to 5 *VLC tasks* running in parallel can be made by associating the number of encoded MacroBlock to be compressed by each of these *VLC tasks*.

Since the two SMPs require different kinds of computation power (our experiments showed an average rate of 6 to 1 between the MainDivX and the VLC), the structure of both SMPs may be different in term of number of tasks. This way, compared with the approaches from [20][21] where the number of VLC tasks were identical to the number of MainDivX tasks, in our approach we can use a number of *VLC tasks* much smaller than the number of *MainDivX tasks*, but just enough to not become a computational bottleneck. When one of the tasks becomes a bottleneck, the amount of data associated to that task is reduced [22]. As a result, in our approach, the application's required memory is reduced.

However, in our approach, the use of 2 additional smaller tasks is required: *Splitter* (for image splitting) and *Combiner* (for final reordering). For test-bench purposes, we have also used 2 test-bench tasks: *Video* and *Storage* (Figure 36).

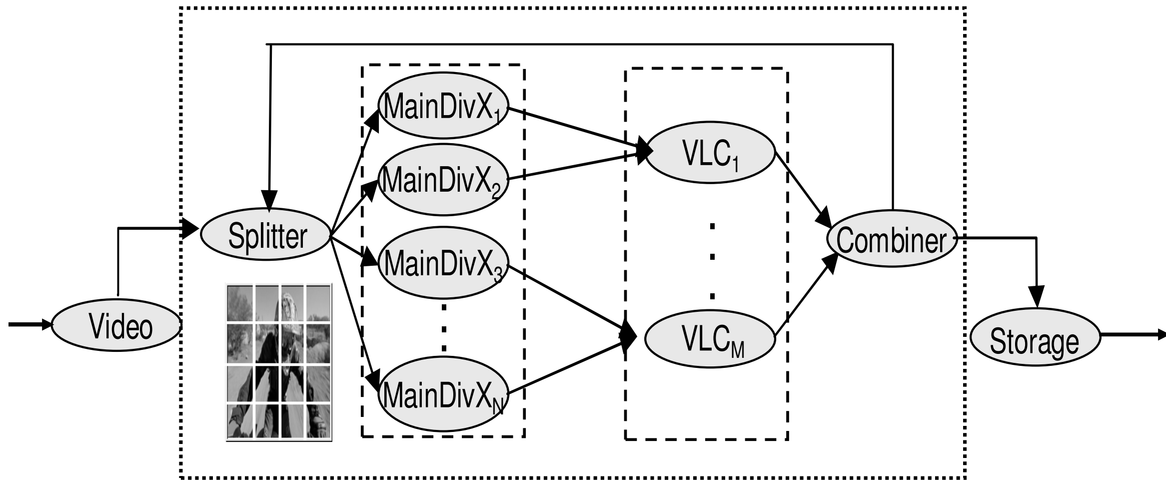


Figure 36. General data flow for the parallel/pipelined MPEG4 video encoder application

The *Video* task does not belong to the final design. It is just a test-bench task, needed to simulate the behavior of a video source. It sends the video under the form of a pixels stream, compatible with YUV411 standard, to the *Splitter*. The *Splitter* divides the image into multiple areas (1 area for each of the MainDivX tasks) by rutting its pixels to the corresponding *MainDivX*, which will encode this area. Each time the *MainDivX tasks* finished encoding a MacroBlock, its encoded results are sent to a corresponding *VLC task*, which will compress this MacroBlocks one by one. The compressed MacroBlocks are then sent to the *Combiner*, which reorders all the *VLC* results, in order to obtain an MPEG4/ISO bitstream. Additionally, the *Combiner* also contains the Bitrate Controller function of the MPEG4 video encoder. After that, this bitstream is sent to the output *Storage* task, which is another test-bench task that simulates a storage support.

As a result, the final algorithm behavior is composed of 2 pipelines. One pipeline at frame level between the *Splitter* and the rest of the algorithm, and a second local pipeline at MacroBlock level between the *MainDivXs*, *VLCs* and the *Combiner.* All the *MainDivX tasks* are executed in parallel, and all the *VLC tasks* are executed in parallel.

It can be noticed that the *Combiner task* is returning a data to be used as input for the *Splitter task*. This data represents the *quanta* value. Since these two tasks belong to a pipeline, it may give the impression that the *Splitter* has to wait for the *Combiner* to give feedback to the *quanta*. However, this is not true, since the *Splitter* will use the previous quanta

value if the *Combiner* did not arrive at sending the new quanta value. Overall, this does not affects the good functionality of the algorithm, especially when the rate_control_delay parameter is high, in which case the *quanta* value will remain constant for a large number of frames. Similar cases can be found in the DivX and XviD implementations, when using multi-pass encoding strategy (this encoding strategy is out of the scope of this document, so we will not detail it further).

The coarse grain partitioning was chosen, instead of fine grain partitioning (i.e. every basic function of the *MainDivX task*, like ME, SAD, DCT, IDCT, Quant, DeQuant be a different task) for simplicity and efficiency. In addition, fine-grained partitioning for highly called tasks (like in case of SAD) may induce serious performance degradation, because of the required context switches between them.

3.2.2 Describing the method used for dividing the input image into multiple smaller areas required for the parallelized MainDivX tasks

As the processing unit of the MPEG4 video encoder is the MacroBlock, the frame (image) division is done at MacroBlock level. After dividing the frame, multiple smaller areas will be obtained. For all the resulting areas, the same number of the *MainDivX* tasks will encode them in parallel. Depending on the required level of parallelism, the frame will be divided into a specific number of areas. This division can be done in multiple ways. Figure 37 shows 3 examples, when a QCIF resolution frame is divided into different areas.

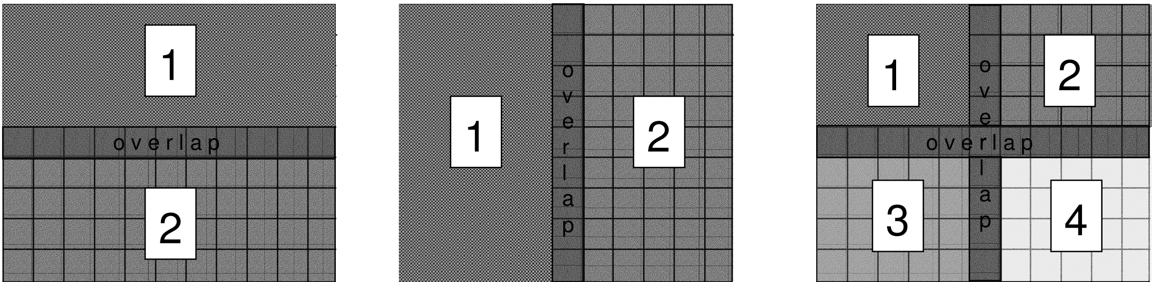


Figure 37. Three examples of frame division methods for the QCIF resolution

It can be noticed that between all neighboring areas, an overlap line/column region is used. The goal of this overlap is to assure the continuity of the global MPEG4 video encoder processing, caused by the progressive nature of the algorithm. For example, during

the Intra Prediction, the computations for a MacroBlock depend on the nature of the neighbor MacroBlocks from the left, top-left, top, and top-right. By using the overlap, we provide to the current Macro-Block (for example, from the first line in Figure 37 left) all the required neighbor Macro-Blocks.

For instance, for the second case from Figure 37, when the *MainDivX2* task (which is encoding the 2nd area) starts processing the first line, it is aware of the final results of the *MainDivX1* task (which is encoding the 1st area) for that line. Thus, the *MainDivX2* can anticipate the results of the *MainDivX1* for that MacroBlock, so that it does not have to wait for the *MainDivX1* to finish that first line. This way, perfect parallelism is assured. Another reason of the overlap is to assure the correct functionality of both *MainDivX* tasks in case objects in the image are moving from an area to another. This feature was not possible in the approaches presented in [20][21].

There are multiple other methods of dividing a frame. For example, a QCIF image can be divided into 8 areas: 4 horizontal x 2 vertical, or 2 horizontal x 4 vertical, etc. The number of these methods becomes significantly bigger when increasing the video resolution.

The only restriction is that, a divided area cannot be smaller than 1 MacroBlock. Using divided areas of exactly 1 MacroBlock (16x16 pixels) leads to a significant reduce of the number of MacroBlocks treated in P mode, because the MotionEstimation cannot be used. The smallest recommended divided area should be of 3x3 MacroBlocks, which ensures the good functionality of the MotionEstimation and the encoding of a MacroBlock as P.

3.3 Flexible Architecture with 2 SMPs

3.3.1 Global view of the Flexible Architecture with 2 SMPs

The targeted architecture model (Figure 38) is a flexible architecture containing 2 SMPs (1 SMP for *MainDivX* and 1 SMP for *VLC*) and an interconnect structure in charge of the inter-processors communications. This architecture is not yet implemented. It just gives an idea about the targeted architecture for the MPEG4 encoder. Additionally, it can impose some architecture parameters. In the case from Figure 38, it was freely chosen to map all the time the *Splitter* and *Combiner* on HW, to avoid becoming I/O bottlenecks.

This architecture is flexible, since it can be configured for different parameters. For example, it can be configured for different number of CPUs for the MainDivX-SMP and the VLC-SMP, depending on the chosen level of parallelism. Each of these CPUs contains only one task. Additionally, the Communication Network can be configured to manage the communication between different number of CPUs, as it will be described later in this document.

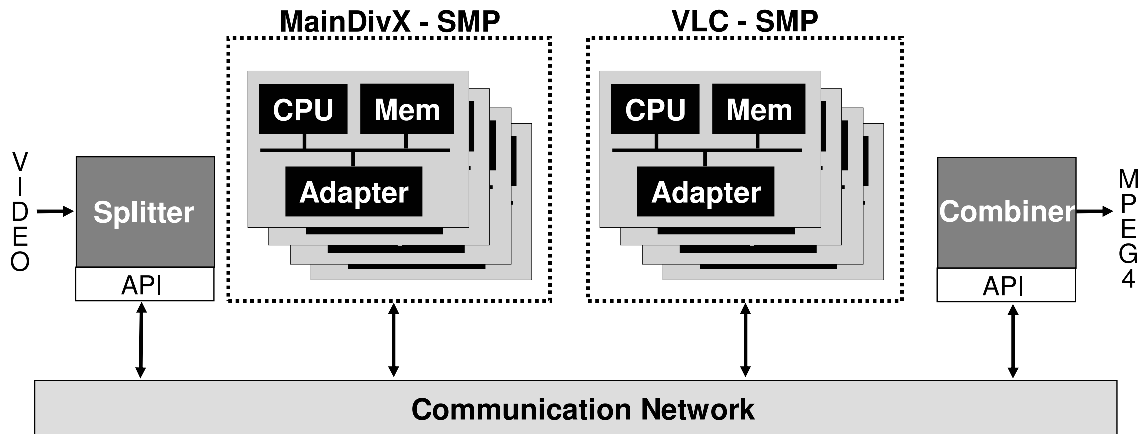


Figure 38. Flexible architecture with 2 SMP

3.3.2 Describing the functionality of the Splitter

The *Splitter* is in charge of receiving a stream of pixels, and of deciding the corresponding *MainDivX* for each pixel. Since the stream of pixels does not contain any information regarding the coordinate of each pixel in the image, is the *Splitter's* job to count the coordinates for each pixel. Based on this coordinate, and a list of coordinates tables associated to it for all divided images of all the *MainDivX*, the *Splitter* can easily determine to which *MainDivX* belongs this pixel. This pixel is stored into a buffer of pixels targeted for this *MainDivX*, along with the relative address (position) of this buffer in the divided area associated to this *MainDivX* (Figure 39).

The sizes of each of the buffers are customizable. The *quanta* value provided to the *Splitter* by the *Combiner* will be the last value sent to the all *MainDivX*. If this *quanta* did not arrive from the *Combiner* until that moment, the Splitter will send the previous *quanta* value.

The division coordinates table depends on the method used to divide the image. For example, for the first case from Figure 37, the division coordinates will contain (x1_start=0, y1_start=0, x1_end=176, y1_end=80) for the 1st area associated to *MainDivX1*, and (x2_start=0, y2_start=64, x2_end=176, y2_end=144) for the 2nd area associated to *MainDivX2*. Since some pixels can belong to multiple regions (overlap), this pixel will be stored into each pixels buffer corresponding to those areas.

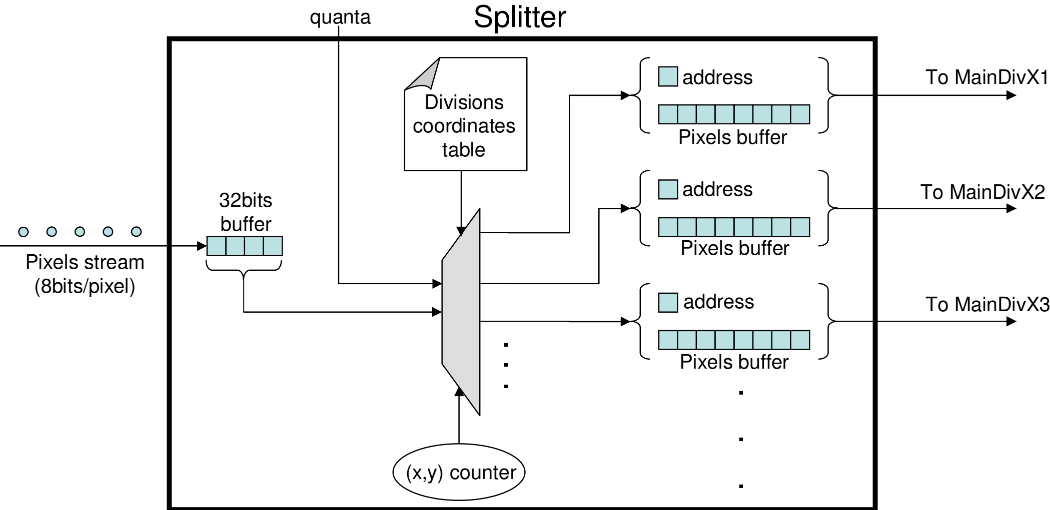


Figure 39. Simplified representation of the Splitter

3.3.3 Describing the functionality of the MainDivX

The *MainDivX* receives its current frame, which is the corresponding part of the frame prepared by the *Splitter*. The *MainDivX* executes the MPEG4 video encoder algorithm on this image. Each time it has finished encoding a MacroBlock, its results are stored into an output buffer to be sent to the corresponding *VLC* (Figure 40). The size of the structure storing the encoded results is always 820 bytes for a MacroBlock, irrelevant if this MacroBlock was encoded as I or P.

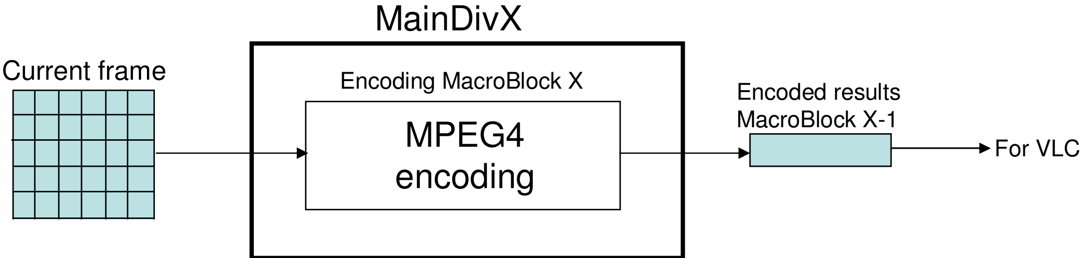


Figure 40. Simplified representation of the MainDivX

Meanwhile, the *MainDivX* continues to encode the next MacroBlock. This loop is repeated until the *MainDivX* has finished processing all the MacroBlocks.

It is important to mention that in the encoded results for a MacroBlock are stored also the coordinates of this MacroBlock in the image. This will be used later by the *Combiner* in order to be able to reorder correctly the MacroBlocks for constructing the MPEG4 bitstream.

Regarding the overlap region, only one of the *MainDivX* will send to the *VLC* the encoded results for a MacroBlock belonging to that overlap. For example, in the first case from the Figure 37, the *MainDivX1* will send the result for the overlap, while the *MainDivX2* will not. In the third case from the Figure 37, for the overlap between the 2nd and 4th area, only the *MainDivX2* will send the results, and not the *MainDivX4*. Generally, the *MainDivX* that sends the results of an overlap is the task with the lower index counter. As a result, all the *VLCs* will receive only one encoded result for each MacroBlock.

3.3.4 Description of the functionality of the VLC

The *VLC* receives the encoded results for a MacroBlock, and compresses it (Figure 41). In case this MacroBlock belongs to the (0,0) coordinates in the image, the *VLC* is attaching in front of this compressed MacroBlock the VOL and VOP headers [5].

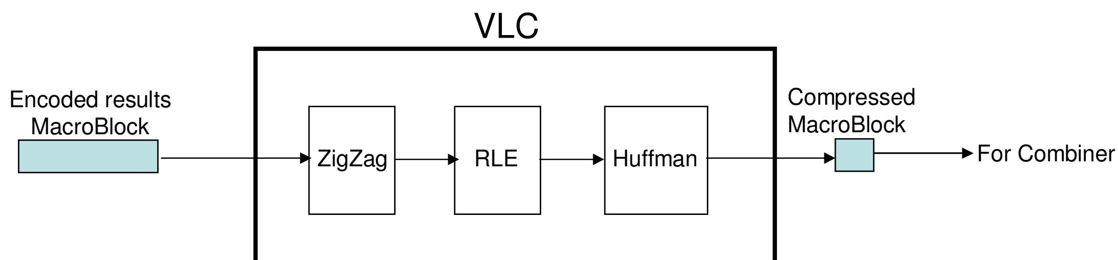


Figure 41. Simplified representation of the VLC

If the size of the input for the *VLC* is known, then the size of the compressed output MacroBlock is variable. The maximum possible size for a compressed MacroBlock is 800 bytes [5]. This maximum is achieved for extreme cases (e.g. white noise in the image).

Therefore, even if the average size is approximately 100 bytes, the maximum allocated buffer size for the compressed MacroBlock has to be able to encapsulate these 800 bytes.

The compressed buffer contains the compressed bitstream for that MacroBlock, the size of the bitstream for that MacroBlock, and the coordinates in the image of that MacroBlock.

It is important to mention that the size of a bitstream for a MacroBlock is represented in bits, and usually this size is not a multiple of 8. Thus, any future concatenation of multiple bitstreams for MacroBlocks has to be done at bit level, and it cannot be done at byte (octet) level. The concatenations are achieved by the *Combiner*.

3.3.5 Describing the functionality of the Combiner

The *Combiner* receives the Bitstreams of MacroBlocks from all the *VLC*s. Based on the coordinate in the image for the received MacroBlock bitstream, the Combiner stores this bitstream into a Bitstreams Buffer (array) used to store all the Bitstreams for each MacroBlock (Figure 42). When each position in this Bitstream Buffer is occupied, then the current frame is completely encoded and compressed. The last phase is to concatenate at bit level all these MacroBlock Bitstreams, obtaining the final MPEG4 bitstream for the current frame.

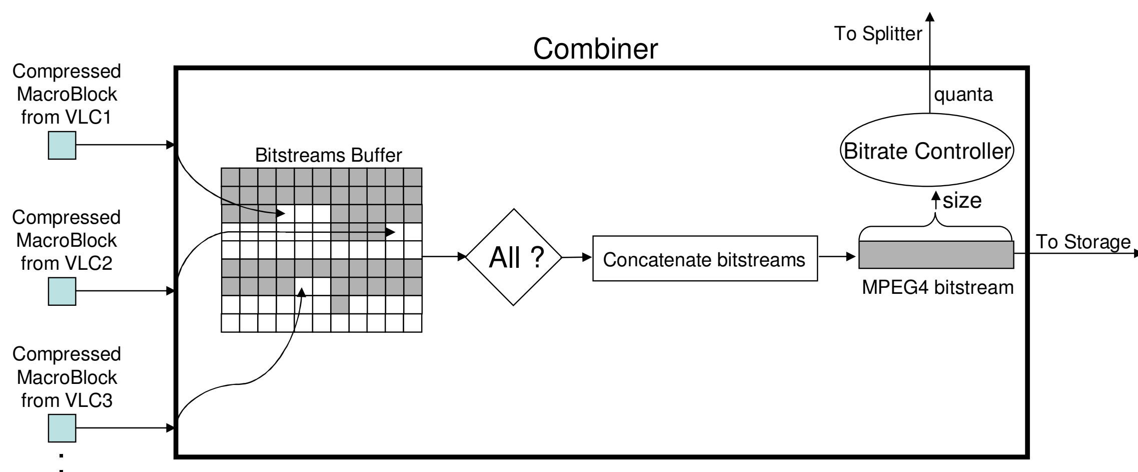


Figure 42. Simplified representation of the Combiner

Once the MPEG4 bitstream is obtained for the current frame, the *Combiner* uses the size of this bitstream along with the Bitrate Controller function, in order to compute (adjust) the *quanta* value to be used when encoding the next frame. This *quanta* will be sent to the *Splitter*, which will send it to all the *MainDivX*.

The resulted MPEG4 bitstream is sent under the form of a data stream. In practice, this MPEG4 bitstream will have to be sent either to a Real-Time Transmition Protocol Interface (RTTPI), for example in case of a mobile-mobile communication, either to a Cinema File System Interface (CFSI), in case this MPEG4 bitstream will have to be stored into a cinema file format (i.e. AVI, WMV). However, these aspects are out of the purpose of the MPEG4 video encoder. In our experiments, the resulted MPEG4 bitstream is sent to a testbench module, called *Storage*, which writes the MPEG4 bitstream on a HDD.

In order to describe the *Flexible Architecture with 2 SMPs*, we use the concept *Combined Algorithm/Architecture Executable Model*. Since multiple architecture configurations may exist (i.e. for different number of CPUs), different configurations of *Combined Algorithm/Architecture Executable Models* are needed. To obtain them automatically, we use a flexible modeling style using a unique *Flexible Algorithm/Architecture Model for MPEG4*, from which multiple configurations of *Combined Algorithm/Architecture Executable Models* can be generated.

3.4 Combined Algorithm/Architecture Executable Model

3.4.1 Concept of Combined Algorithm/Architecture Executable Model

The *Combined Algorithm/Architecture Executable Model* represents a high-level simulation model used to describe a system. This system contains an architecture and an algorithm. The algorithm will be executed on this architecture. Such a model allows to create accurate executable specifications for a complex system, from the beginning of the design flow. This model provides significant advantages [27]:

a) It avoids inconsistency, errors and helps to ensure the completeness of the specifications. Creating an executable specification for a system represents in fact the realization of a program that behaves in the same way as the system.

b) It captures the behavior of the architecture and the algorithm, and the interaction between them. This allows us to build a correct system, which ensures the good functionality of the algorithm and the architecture "running together".

c) It allows to validate, simulate and debug the system functionality, even before the actual chip implementation begins. These are achieved by the executing the specification, and monitor its behavior.

d) It helps to create early performance models of the system and validate system performances.

e) It allows the use of test-bench data as input for the system. This allows the possibility to test different functionality scenarios, even before implementing the chip.

To describe a system as a simulation model, there are 3 important aspects which have to be taken into account: the concept of *components, links* and *organization* (Figure 43).

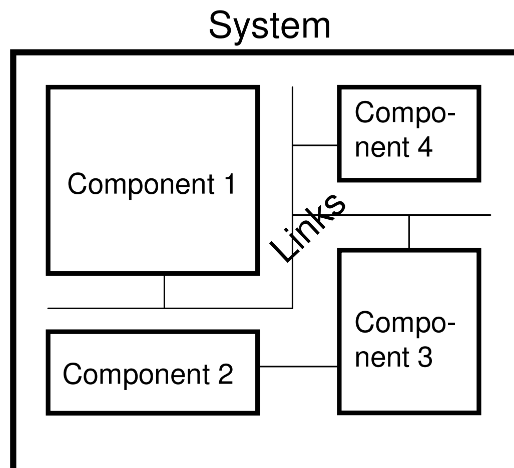


Figure 43. Describing a system using *components* and *links*

a) A *component* is a basic building block used to describe a part of the system. A complete system is composed by one or multiple *components*. These *components* allow the designers to break complex systems into smaller more manageable pieces. Additionally, a *component* allows the hiding of internal data representation and algorithm from other *components*. Thus, the change of a *component* can be done without modifying other *components*. This allows the designer to optimize the design locally, at *component* level. Each *component* contains a behavior description (named *contain*)

and an *interface* [57]. The *contain* represents an algorithm describing the functionality of the *component*, or a system. In other words, it is possible to describe a *component* as another system containing multiple *sub-components*. This is called hierarchical system description. The *interfaces* contain a behavior that allows the *component* to interact with the rest of the system, through the system's *links*. More details about *components* can be found in [57].

b) The *links* represent the interconnection infrastructure between the *components*. The *links* allow the interaction between the *components*, by ensuring the transport of the data between the *components.* The *links* between the components can be described at different abstraction levels, as a network (Philips AEthereal, STMicroelectronics Octagon, etc), abstract interconnections (FIFO channel, MPI channel, etc) or physical interconnections (physical wires).

   Depending on their abstraction levels, the *links* can hide some communication details from the *components*. If the *links* are described at a high-level (abstract interconnections), the *components* do not have to manage explicitly the protocol, but only the "services" provided by such *links*. For example, in case the links are MPI channels, the low-level communication details are completely hidden from the *components*. As will be detailed later in this document, the *components* are communicating by calling specific MPI functions, like *MPI_Send* or MPI_*Recv*. However, if the *links* are described at low-level (physical interconnections), the *components* will have to manage explicitly the protocols (i.e. handshake) imposed by such *links*. More details about the links can be found in [57].

   By using the concept of *components* and *links*, the computation aspects are separated from the communication aspects.

c) To obtain a correct functional simulation model, all the *components* and *links* have to be assembled together using specific *organization* semantics ("rules") [57]. In other words, it is not only sufficient to build *components* and *links* to obtain a simulation model, they also have to be assembled correctly. For example, every *link* has to be connected with two *components*. Additionally, the obtained simulation model has to be executed, in order to validate its behavior. The execution of a simulation

model can be done in two ways: execution using a simulation environment, execution as a standalone executable program. Examples of simulation models that are executed using a simulation environment are the models described using Matlab, VHDL, Simulink, etc. [57]. To execute these models, it is required to use an additional tool containing the execution environment needed to "interpret" the functionality of this model. An example of simulation model that is executed as standalone executable program is the model described in SystemC. Since this model is based on C/C++ language, it is sufficient to compile the model using a typical C/C++ compilation approach, to obtain the standalone executable program containing the functionality of the model [27]. In this case, the execution environment is integrated into the executable program.

3.4.2 SystemC used for the description of the Combined Algorithm/Architecture Executable Model

SystemC is a C++ class library and a language to design system description models [27]. These models are described using executable specifications. An executable specification is essentially a C/C++ program that exhibits the same behavior as the system when executed. Syntactically, designing a model in SystemC is identical to classical Object Oriented Programming. All the aspects from a model, either *components* or *links,* are in fact C++ objects. This makes the entire model compilable, and thus executable.

A model described using SystemC uses the following concepts:

a) The *components* in SystemC are called Modules. These modules are declared using the SC_MODULE class. The *contain* of these modules can be algorithmic tasks described as threads (SC_CTHREAD, SC_THREAD, SC_METHOD). The complete behavior of these tasks can be described using C/C++ language. Additionally, the *contain* of a module can be another module (SC_MODULE). This allows a hierarchical description of the system. The interface of a Module is declared using SC_INTERFACE class. This is used to ensure the interaction between the *contain* of the module with the exterior of the module.

b) The *links* in SystemC are defined as *ports* (SC_PORT), *signals* (SC_SIGNAL) and *channels* (SC_CHANNEL). The *ports* of a module are in fact external interfaces of that module, which pass information to and from the module, and trigger actions into the module. The *signals* create elementary connections between the modules *ports* allowing the modules to communicate [27]. In fact, *signals* can be seen as physical wires. The *channels* represent an abstract interconnect method between the modules *ports*, which can abstract multiple signals and protocols. By using *channels*, like MPI channels, it is possible to model abstract interconnections.

c) The SystemC execution environment is in fact an executable simulation engine, called SystemC scheduler, used to execute the behavior of the *modules*, *ports*, *signals* and *channels*. The execution is divided into two phases: initialization phase and execution phase [57]. During the initialization phase, the scheduler is loaded, after which are instantiated all the *modules*, *ports*, *signals* and *channels* from the current simulation model. During the execution phase, SystemC executes all the tasks until they arrive at a “blocking point”, for example at wait instructions [57]. In this moment, the SystemC updates all the values from the ports and signals, and the list of tasks that has to be executed in the next simulation time (clock cycle). If there is no task to be executed, the SystemC keeps advancing the simulation time until there is a task to be executed. So, the SystemC simulation is cycle-based [27], and the synchronizations between the tasks can be of two types: synchronization using physical clock, or synchronization using logical clock (when there is no existing physical clock). More details about SystemC execution environment can be found in [57] and [27].

The main advantage of describing a model in SystemC is that it provides an understandable language for both software and hardware designers. For the same model, the software designers can interpret it as an object oriented program, while the hardware designers can interpret it as an ensemble of ports, signals, clocks, modules, etc. Additionally, when describing models in SystemC, it allows the software designers to use typical C/C++ programming techniques for the implementation of complex algorithm tasks that will be integrated into modules (i.e. *MainDivX* task).

3.4.3 Combined Algorithm/Architecture Executable Model using MPI-SystemC HLPPM

The *Combined Algorithm/Architecture Executable Model using MPI-SystemC abstract interconnect execution model* is a system described completely in SystemC. This system is formed by multiple SystemC modules. Each module contains one or more algorithmic tasks. The communications between the tasks are done using message passing. The interconnections between the tasks are done through an *abstract interconnect execution model*, called *MPI_SystemC* (Figure 44). In [30], such an interconnect execution model is called *High-Level Parallel Programming Model* (HLPPM). A HLPPM hides completely the low-level architecture details: Communication Network, HW/SW & HW/HW Interfaces.

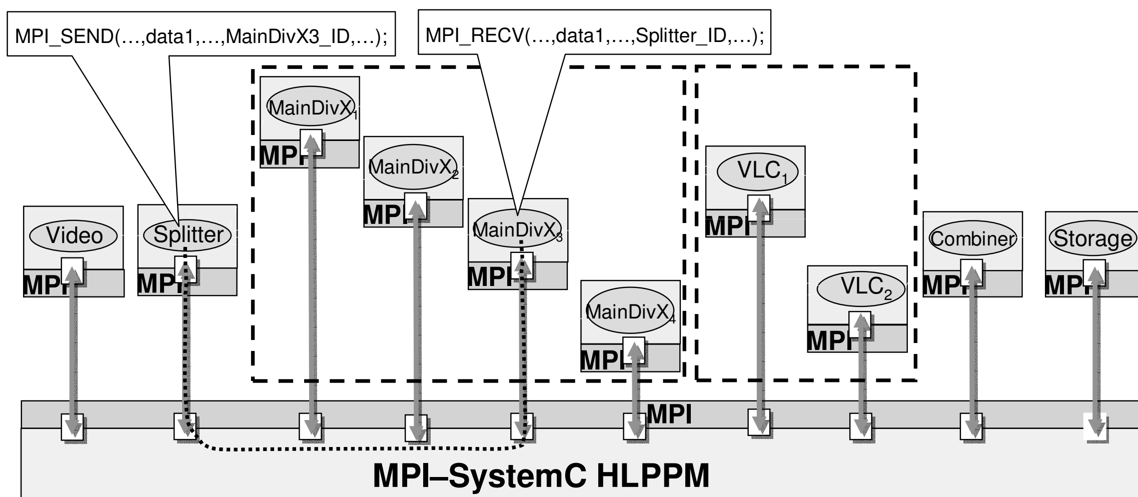


Figure 44. Combined Algorithm/Architecture Executable Model using MPI-SystemC

The example in Figure 44 shows an example of *Combined Algorithm/Architecture Executable Model for MPEG4*, using 4 MainDivXs and 2 VLCs. The model is formed by multiple SystemC modules, each of them containing one task. These tasks contain the algorithm behavior, described in C/C++. It can be noticed that this model also contains the *Video* and *Storage* modules. However, these 2 modules are only for test-bench purpose, and they will not be used in the final architecture. The tasks from the modules are communicating through message passing by calling a set of MPI primitives Figure 45. The syntax and purpose of each MPI primitive are detailed in [31] and [58].

```
MP_Init( *this,argc,argv);
MP_Finalize( *this);

MP_[I]Send( *this,buf,count,datatype,dest,tag,comm);
MP_[I]Recv( *this,buf,count,datatype,source,tag,comm,status);

MP_[I]BSend( *this,buf,count,datatype,dest,tag,comm);
MP_[I]BRecv( *this,buf,count,datatype,source,tag,comm,status);

MP_[I]SSend( *this,buf,count,datatype,dest,tag,comm);
MP_[I]SRecv( *this,buf,count,datatype,source,tag,comm,status);

MPI_Wait( *this,request,status);
MPI_Test( *this,request,flag,status);
```

Figure 45. Subset of MPI communication primitives

The *Splitter* task sends a message *data1* to *MainDivX3* task, by mentioning in the MPI_SEND primitive the unique IDs of *MainDivX3* task. On the other sides, the *MainDivX3* task will call an MPI_RECV primitive to receive the data from the *Splitter* task.

When using communication with MPI primitives, the low-level communication protocols and details, HW/SW and HW/HW interfaces are completely abstracted, thanks to the *MPI_SystemC HLPPM*.

The *MPI_SystemC HLPPM* [58] is a runtime execution environment for message passing communication using the subset of MPI primitives presented in Figure 45. It is similar to MPICH [31] (supports the same MPI primitives) but with the possibility of including configurable timing annotations for the communication, using SystemC libraries. Figure 46 shows that the communication between two tasks is done using Communication Units (CU) (one CU for each task), which manages the MPI requests from the tasks, the communication with other CUs, and inserts the timing annotations. Since a CU can be connected to many other CUs, the *MPI-SystemC HLPPM* can support point to point and bus topologies.

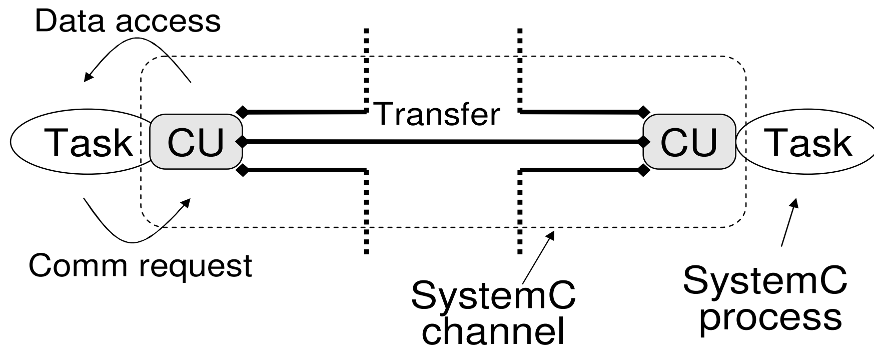


Figure 46. Task-to-Task communication using MPI-SystemC

3.5 Flexible Algorithm/Architecture Model for MPEG4

In the followings section, the unique *Flexible Algorithm/Architecture Model for MPEG4* is presented. It is used to obtain automatically different configurations of *Combined Algorithm/Architecture Executable Models*. This is possible by using the concepts of tasks with *Flexible Computations*, *Flexible Outputs* and *Flexible Inputs*. The resulted *Combined Algorithm/Architecture Executable Models* is used to represent a specific configuration of the *Flexible Architecture with 2 SMPs* for MPEG4.

3.5.1 Concept of tasks with Flexible Computations

We call a task with flexible computations, the task from which can be obtained multiple tasks forming a SMP. This is obtained by instantiating this task into multiple identical tasks that will be executed in parallel. Such tasks can be the *MainDivX* task, *VLC* task. For each example, the code of all the instantiated tasks is identical. The only difference is the working parameters. However, in the task with flexible computations, these parameters are not yet decided. An example of obtaining multiple customized tasks forming a SMP from a task with flexible computations is presented in Figure 47.

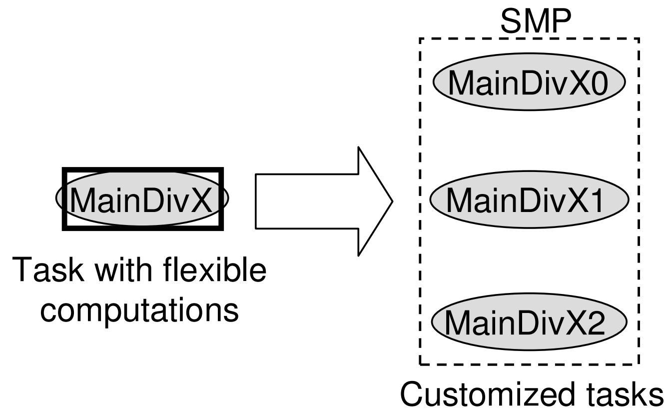


Figure 47. Task with flexible computations used to obtain multiple customized tasks in a SMP

To describe a task with flexible computations we use the concept of macro-code. A task described into macro-code can be expanded into multiple identical tasks using a macro-expander tool. Example of such macro-expender tools is the RIVE. The RIVE macro-expander tool was developed by Lovic Gauthier as part of his PhD work [41]. The purpose of this language is similar to M4 [32], but it is easier to use. This tool receives as input a file described in macro-code, and a file containing a set of parameters. Using these two files, the RIVE tool generates one or more text files, by customizing the macro-code with the parameters contained in the configuration file. The advantage of using this approach is to be able to describe generic elements, from which can be generated specific elements.

The process of macro-expending the *MainDivX* task is presented in Figure 48. This task is described into MainDivX.cpp.riv file (in this example, we have decided to present only the code lines significant to this sub-chapter). This file contains the macro-code for the main function of the *MainDivX* task. This function is called *MainDivX"N"_MAIN*. The value *N* means that it was not yet decided how many *MainDivX* tasks will be used later. Inside this function is called the function *MainDivX_COMPUTE*, which contains the computations needed to encode an image. The values for the parameters used by this function are not yet decided. These parameters are the height and length of the image that will be processed by the *MainDivX*, some border related parameters required for the overlaps and MotionEstimation, and some internal specific data structures.

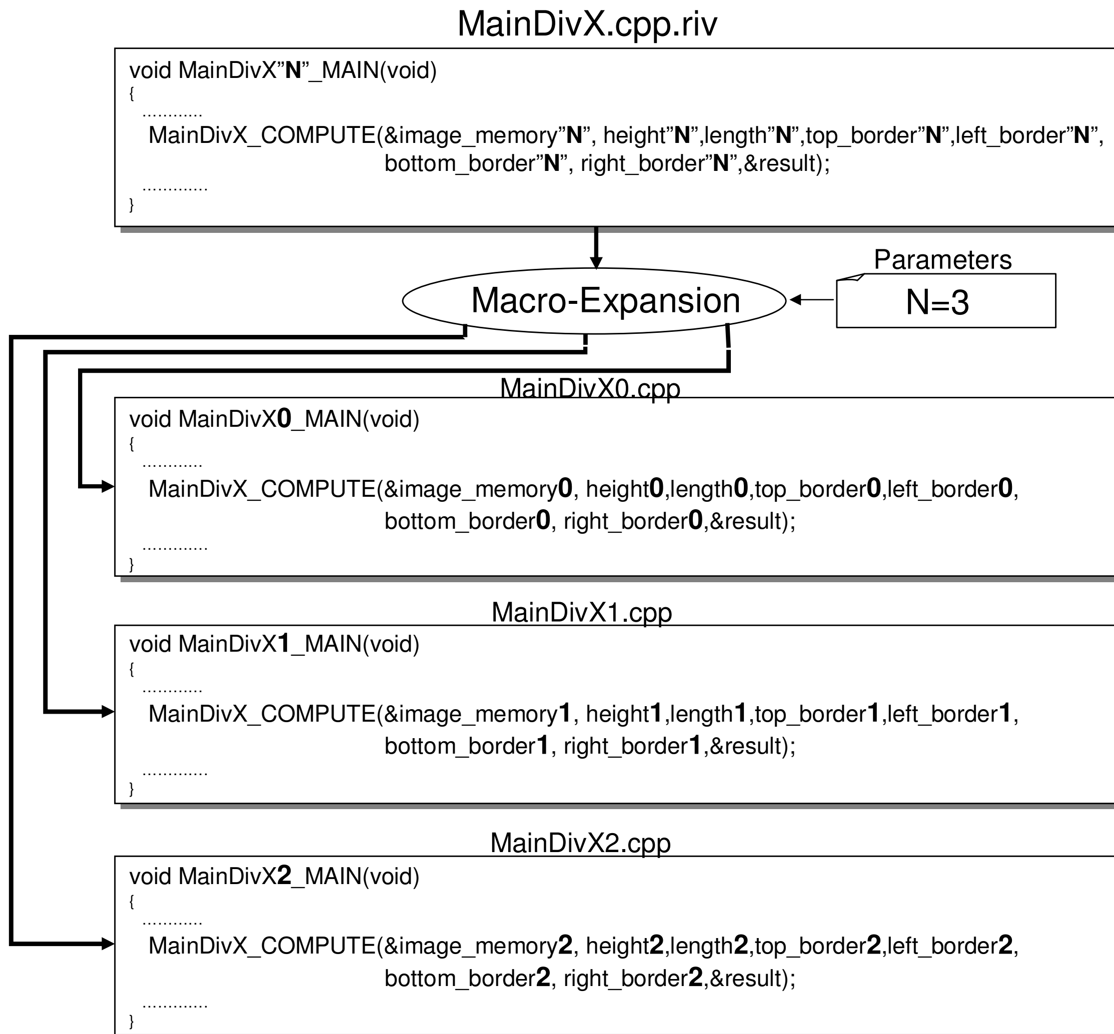


Figure 48. Macro-generation from a task with flexible computations

From the MainDivX.cpp.riv file, multiple instances of MainDivX tasks can be macro-generated. The macro-generation is done calling the RIVE macro-expander using the following syntax: *RIVE MainDivX.cpp.riv –i Parameters –o MainDivX.cpp.* The macro-generator will use as input file the MainDivX.cpp.riv which will be expanded using the parameters from the file Parameters. In this example, the used parameter is N=3, which means that we want to obtain 3 *MainDivX* tasks. The results of the macro-generation are 3 compilable C/C++ files belonging to the 3 *MainDivX* tasks.

For each of the *MainDivX* task, the parameters are now fixed. The actual values for some of these parameters are stored into a header file that is generated using the same macro-generation technique. The important thing is that all 3 *MainDivX* tasks will use the

same function to encode their corresponding images. The only differences are the used parameters.

As a result, a task with flexible computations represents a task from which multiple customized tasks can be macro-generated. These customized tasks are forming a SMP. The code of these tasks is practically identical, the only difference are the used parameters.

3.5.2 Concept of tasks with Flexible Output

We call a task with flexible output, a task that can be adapted to send data to a different number of other tasks. The destination tasks are usually belonging to a SMP. An example of such task is the *Splitter* task (Figure 36 and Figure 39). The *Splitter* task has to send data to multiple *MainDivX* tasks. The problem is that, the number of *MainDivX* tasks may vary. This means that it should be easy to adapt the *Splitter* task to support a customizable number of outputs (Figure 49).

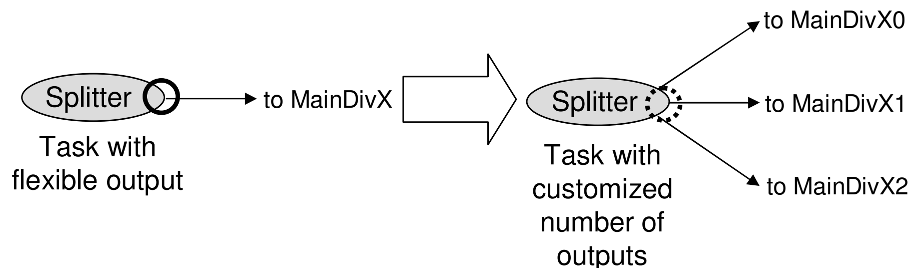


Figure 49. Task with flexible output used to obtain task with customized number of outputs

To achieve this, we use the same macro-expansion technique, like the cases used for the tasks with flexible computations. The process of macro-expending the *Splitter* task is presented in Figure 50.

The code of the *Splitter* task is in the Splitter_MAIN function from the Splitter.cpp.riv file. In this function, the *Splitter* task will send data to N targeted *MainDivX* tasks. This is done by using a loop approach, in which the target_divx is from 0 to N-1. In each loop iteration, the *Splitter* will check first if the buffer corresponding to the current target_divx is full. If it is full, the Splitter will send it to the target_divx *MainDivX* task, using an MPI_SEND call. None of the parameters of the MPI_SEND are fixed yet. In addition, the number of *MainDivX* tasks is also not yet fixed.

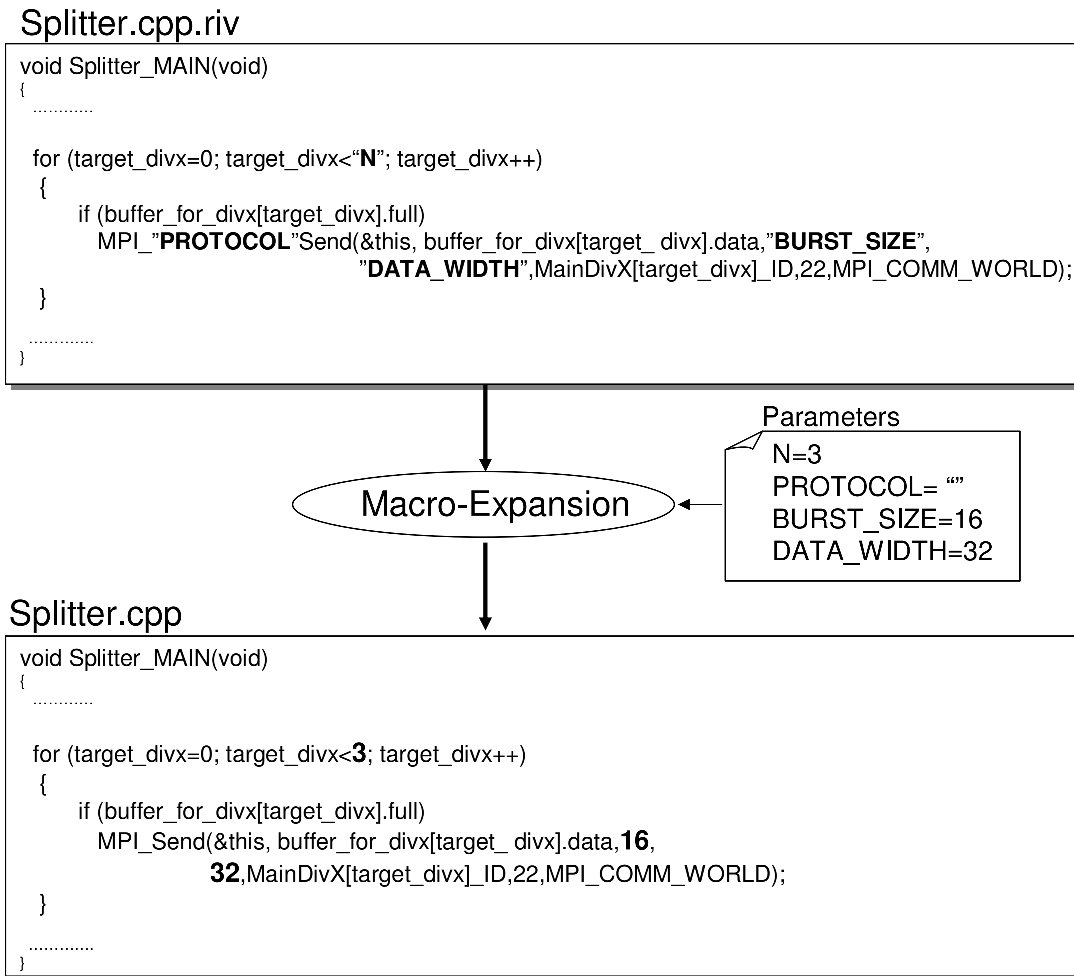


Figure 50. Macro-generating a task with flexible outputs

From Splitter.cpp.riv can be macro-generated the Splitter.cpp file, containing the *Splitter* task with customized outputs. In this file, the number of *MainDivX* tasks is already fixed to 3. Thus, the target_divx counter will advance from 0 to 2. For each target_divx value, the Splitter tasks will check if the buffer for *MainDivX* target_divx is full. If it is full, it will send the data from this buffer to the MainDivX[target_divx]. Additionally, the parameters for the MPI_SEND communication primitive are fixed: the protocol is blocking (because the PROTOCOL="" resulting in blocking MPI_SEND), the burst size is set to 16, and the data width is set to 32 bits. As a result, the *Splitter* task was customized to be able to send data to 3 *MainDivX* tasks and using customized communication parameters.

As a result, a task with flexible output is used to macro-generate a task that can send data to a customized number of destinations using customized communication parameters.

3.5.3 Concept of tasks with Flexible Input

We call a task with flexible input, a task that can be adapted to receive data from different number of tasks. The source tasks are usually belonging to a SMP. An example of such task is the *Combiner* task (Figure 36 and Figure 42). The *Combiner* task receives data from multiple *VLC* tasks. However, the number of *VLC* tasks can vary. This means that the *Combiner* task will have to be easily adaptable to support a customizable number of inputs.

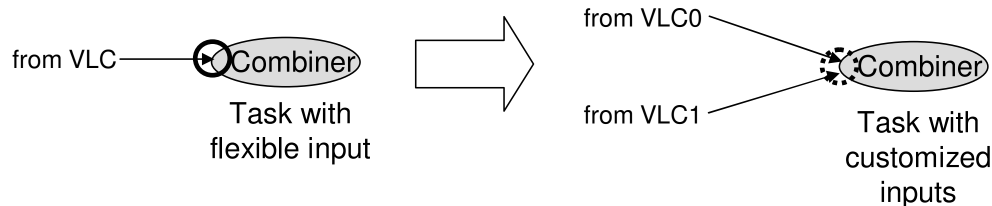


Figure 51. Task with flexible input used to obtain task with customized number of inputs

In this case also, a macro-expansion technique is used, similar to the one used for the tasks with flexible computations and tasks with flexible outputs. The process of macro-expending the *Combiner* task is presented in Figure 50. The code of the *Combiner* task is in the Combiner_MAIN function from the Combiner.cpp.riv file. In this function, the *Combiner* task will receive data from any VLC that is sending data to the *Combiner* task. The *Combiner* tasks will receive the data in the same order as the *VLCs* are sending the data. This is possible by using in the MPI_RECV (from the *Combiner* task) the parameter MPI_ANY_SOURCE. Of course, in each *VLC* task is used an MPI_SEND(…., Combiner_ID, ….). Thus, compared with the flexible outputs, for the flexible inputs it is not necessary to mention the number of sources. The MPI_SEND primitive will be in charge of arbitrating the data coming to the *Combiner* task from the VLC tasks.

The Combiner.cpp file, containing the *Combiner* task with customized inputs can be macro-generated from the Combiner.cpp.riv file. The communication parameters for the MPI_Recv are also customized. As a result, the *Combiner* task was customized to be able to receive data from all the VLC tasks sending data to the *Combiner*.

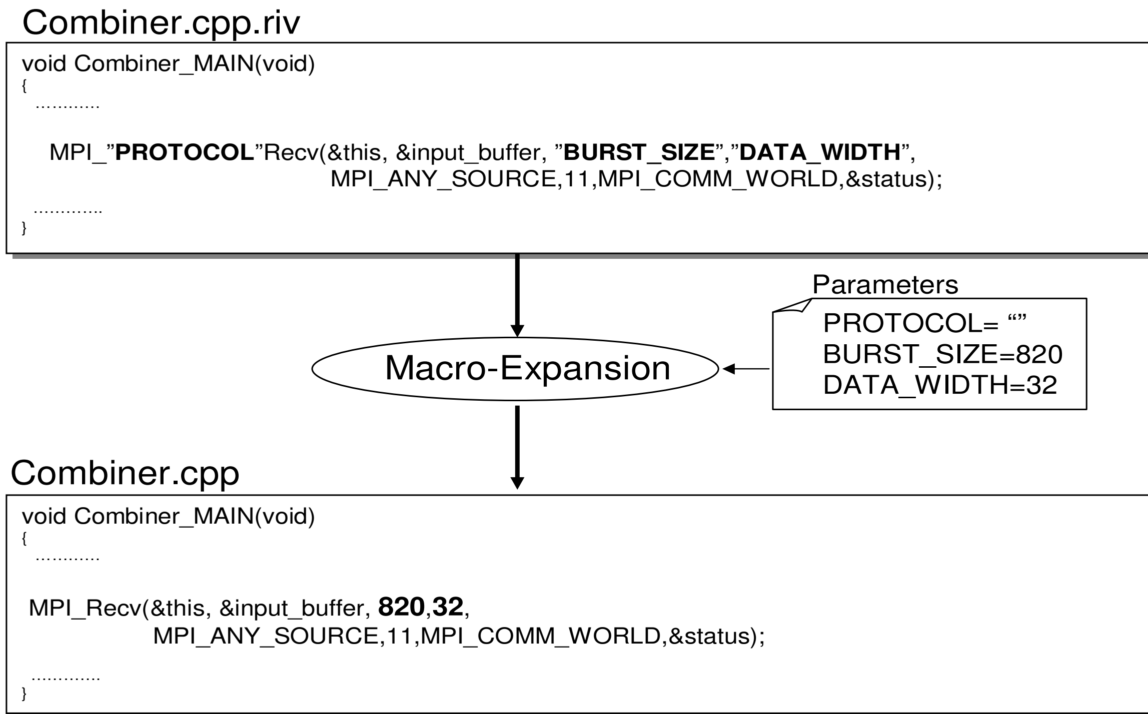


Figure 52. Macro-generating a task with flexible input

3.5.4 Flexible Algorithm/Architecture Model for MPEG4

The *Flexible Algorithm/Architecture Model for MPEG4* (Figure 53) is composed of Modules and an *Abstract interconnect execution model (MPI-SystemC HLPPM)*. Each Module contains one task. Each task can be of 2 types:

a) flexible task, which has at least one of following characteristics:

- flexible computations – task belonging to a SMP : *MainDivX*, *VLC*

- flexible input – task which is receiving data from a SMP: *VLC*, *Combiner*

- flexible output – task which is sending data to a SMP: *Splitter*, *MainDivX*

b) fixed task – none of the above: *Video*, *Storage*

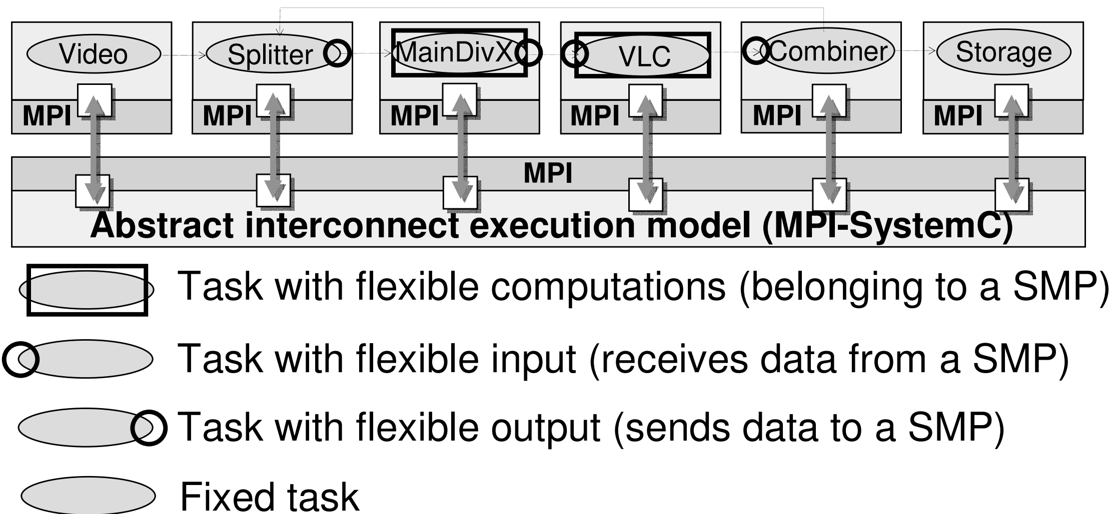


Figure 53. Flexible Algorithm/Architecture Model for MPEG4

Each element of this model is written as macro-code: tasks, SystemC modules. There are approximately 30 files written as macro-code, representing around 2000 lines of code. The main files which are not written as macro-code are the files containing the computations belonging to the *MainDivX* and *VLC* tasks, and the MPI-SystemC. These are approximately 50 files, representing around 15000 lines of code for the MPEG4 encoder algorithm and approximately 4000 lines of code for the MPI-SystemC.

The tasks are communicating through message passing by using MPI primitives. A simplified macro-code for the *MainDivX task* with MPI primitives is shown in Figure 54. *MPI_**"PROTOCOL"**Recv(this,&image_memory**"N"**,sizeof(image_memory**"N"**),**"DATA_WIDTH"**,SPLITTER_ID,22,MPI_COMM_WORLD,&status)* receives data from the *Splitter* task. This primitive specifies the pointer were data will be stored, the amount of data, the communication data width and the unique ID assigned to the source task (*Splitter*). It can be noticed that most parameters are not yet fixed.

Using the *Flexible Algorithm/Architecture Model for MPEG4*, many customized models can be macro-generated. This is done by expanding every file containing macro-code with the desired algorithm/architecture configuration parameters. The result represents the *Combined Algorithm/Architecture Executable Model* in which the entire algorithm and architecture are customized.

```
//------------------ MainDivX task ---------------------------------------------
EXTERN *image_memory"N",height"N", length"N", top_border"N", left_border"N",
            bottom_border"N", right_border"N",*result"N";

void MainDivX"N"_MAIN (void)
{
  //initialization of computations
  MainDivX_INIT (&image_memory"N", height"N", length"N");

  //infinite loop for every frame
  while (1)
  {
        //data_receive_communication from the Splitter
        MPI_"PROTOCOL"Recv(this,&image_memory"N",sizeof(image_memory"N"),
                                    "DATA_WIDTH",SPLITTER_ID,22,MPI_COMM_WORLD,&status);

        //calls the function with flexible computations
       MainDivX_COMPUTE (&image_memory"N",height"N", length"N", top_border"N",
                              left_border"N", bottom_border"N", right_border"N",&result"N");

       //send_results_communication to the VLC
       MPI_"PROTOCOL"Send(this,&result"N",sizeof(result"N"),"DATA_WIDTH",
                                    VLC["target_vlc"]_ID,22, MPI_COMM_WORLD);
}
```

Figure 54. Example of task description using MPI primitives

3.5.5 Algorithm and Architecture configurations

During the macro-expansion process, we use 2 categories of configuration parameters, shown in Figure 55.

Algorithm parameters	**Architecture parameters**
Level of Parallelism/Pipeline	Number of CPUs
Video resolution	Type of CPUs
Frame rate	HW-SW partitioning
Bitrate	Communication topology
Key frame	Blocking/Non-blocking comm.
MotionEstimation precision	Arbitration type
MotionEstimation search area	Message size
Progressive/Interlaced mode	Data width
Scene change detection	Data transfer latency
Quantization range	Transfer initialization latency
Quantization type (H263,MPEG4)	Transfer close latency

Figure 55. Algorithm and Architecture parameters

Using these configurations, the designer is able to choose the configuration of the generated *Combined Algorithm/Architecture Executable Model* for MPEG4 into multi-processor architecture. Additionally, these parameters allow targeting different applications

(i.e. mobile telecommunication), each of them requiring different algorithm and architecture configurations (i.e. video resolution, frame rate, number of CPUs, type of CPUs, etc).

3.5.6 Obtaining the Combined Algorithm/Architecture Executable Model

Figure 56 shows an example of macro-generated SystemC Model with 2 SMP subsystems (*MainDivX* and *VLC*), and the data dependencies between the tasks (the dotted arrows). This model is called *Combined Algorithm/Architecture Executable Model*. It is an un-timed model, and it captures both the architecture and the algorithm.

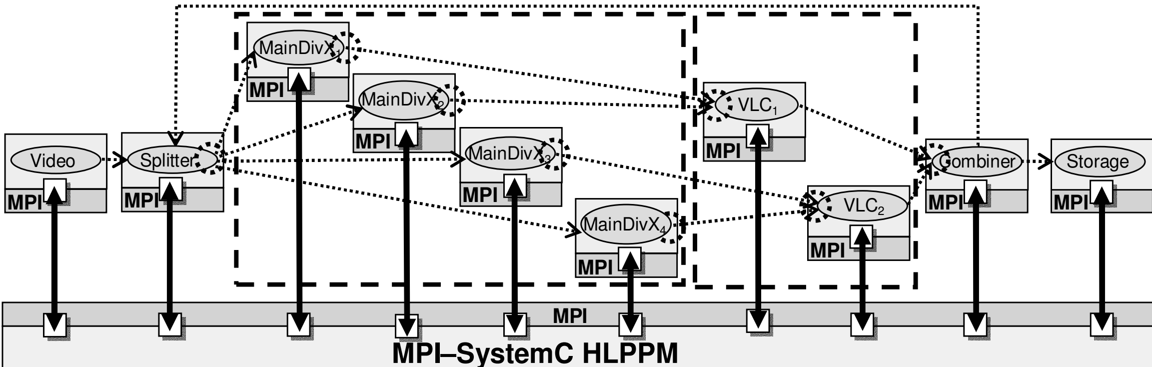


Figure 56. Example of obtained Combined Algorithm/Architecture Executable Model

This model is obtained by macro-generating each file containing macro-codes from the *Flexible Algorithm/Architecture Model for MPEG4*. Figure 57 shows the C/C++ code of the resulted *MainDivX1* task after the macro-expansion.

```
//------------------ MainDivX task ---------------------------------------------
EXTERN *image_memory1,height1, length1, top_border1, left_border1,
            bottom_border1, right_border1,*result1;

void MainDivX1_MAIN (void)
{
  //initialization of computations
  MainDivX_INIT (&image_memory1, height1, length1);

  //infinite loop for every frame
  while (1)
  {
        //data_receive_communication from the Splitter
        MPI_Recv(this,&image_memory1,sizeof(image_memory1),
                    32,SPLITTER_ID,22,MPI_COMM_WORLD,&status);

        //calls the function with flexible computations
        MainDivX_COMPUTE (&image_memory1,height1, length1, top_border1,
                            left_border1, bottom_border1, right_border1,&result1);

        //send_results_communication to the VLC
        MPI_BSend(this,&result1,sizeof(result1),32,
                    VLC[0]_ID,22, MPI_COMM_WORLD);
}
```

Figure 57. Code of MainDivX1 task obtained after the macro-expansion

It can be seen that all the algorithm/architecture parameters are now fixed. For different algorithm/architecture configuration parameters, different Combined Algorithm/ Architecture Executable Models are obtained. The key advantage of such a model is its suitability for performances analysis, algorithm debug, synchronization debug, etc.

3.6 Conclusions

The objective of this chapter was to present our proposed flexible modeling style used to obtain automatically different *Combined Algorithm/Architecture Executable Models* for MPEG4 starting from a unique *Flexible Algorithm/Architecture Model* for MPEG4. This is used to represent at high-level of abstraction different configurations of the targeted Flexible Architecture for MPEG4 with 2 SMPs, containing a highly parallelizable/pipelined MPEG4 video encoder algorithm.

This algorithm can be used for multiple applications, like home-cinema, video recording, video surveillance, mobile telecommunications, etc. For each application, the algorithm can be configured with the required algorithm parameters, and adapted to the used multiprocessors architecture. This algorithm can support different computations distribu-

tion configurations on multiprocessors architectures, using parallel and pipeline execution schemes. The parallelism support is done using 2 SMP, each SMP containing multiple identical tasks running in parallel. The parallelism level is configurable, based on the configuration of the required multiprocessor architecture. The pipeline support is done by separating the MPEG4 encoder algorithm in two phases: encoding phase, and compression phase. These two phases can execute in pipeline, at MacroBlock level.

The generation of different *Combined Algorithm/Architecture Executable Model* for MPEG4 is possible thanks to the unique *Flexible Algorithm/Architecture Model* for MPEG4, containing flexible tasks with flexible computations (MainDivX-SMP and VLC-SMP), flexible inputs and flexible outputs. Using a macro-expansion approach, different executable models are automatically obtained for different algorithm and architecture parameters.

This approach allows to obtain automatically different configurations of algorithm/architecture executable models for MPEG4, for different algorithm and architecture parameters, even for a large number of processors (i.e. >100 CPUs). The resulted models can be used for performances analysis, algorithm debug, synchronization debug, etc.

4 High-Level Algorithm/Architecture Exploration

This chapter presents the proposed High-Level Algorithm/Architecture Exploration flow. This flow allows us to explore different algorithm and architecture configurations, even before starting the implementation of the MPEG4 encoder on MP-SoC. This approach leads to a significant reduction of the design cost. This flow is using the unique *Flexible Algorithm/Architecture Model for MPEG4* from which different customized models can be automatically obtained. These customized models are used for performance estimations, using time annotation technique. Using this flow, multiple configurations of MPEG4 encoder on MP-SoC were successfully explored, for different algorithm and architecture configurations.

4.1 Principle of the High-Level Algorithm/Architecture Exploration

4.1.1 Introduction

Video encoding is widely included in most of consumer, multimedia, mobile and telecommunication applications [17]. This is becoming a key technology for many future applications. These different applications impose different constraints on the encoding parameters (video resolution, frame rate) and on the resulting design (cost, speed and power). Even if MPEG4 seems to be a nearly accepted common standard for most embedded systems domains, a plethora of MPEG4 architectures exists today to comply with different applications [3][17] [18][19].

Implementing an MP-SoC architecture until the RTL level, starting from a wrong set of ad-hoc parameters (i.e. CPUs number/type, or communication topology) implies a costly design. Each modification of parameters will lead to the need of complex modifications (which may also lead to a deteriorated result, because of the bugs appearing after these "forced" modifications), or in the worst case the need to redesign completely the architecture. In case of simple architectures, these problems appear rarely, but in case of complex applications, like MPEG4 video encoder on MP-SoC, this represents an important blockage for the project. Additionally, this effort needs to be repeated for each application configuration requiring an MPEG4 video encoder (i.e. for a different resolution). Few products may justify such a design budget, and the only working solution to get video encoding for low cost products (such as consumers) is to reduce the design cost of the product. One solution to reduce the design cost is to do the architecture exploration at a high-level, before implementing the low-level architecture.

4.1.2 Solution space for MPEG4 encoder on MP-SoC

Implementations of MPEG4 video encoder on MP-SoC can be applied in multiple domains: video surveillance, camera recorders, mobile telecommunications, home cinema, etc. Each of them requires specific architecture configurations, and imposes their own constraints in term of speed, power and chip surface. Finding the final implementation solution

requires adjusting a large number of parameters. These parameters can be split into two categories:

a) *Standard MPEG4 Algorithm parameters* are related specifically to the algorithm functionality: video resolution, frame rate, bitrate, quantization range, quantization type, motion estimation precision, motion search area, progressive/interlaced encoding, key frame rate, scene change detection, etc [5]. To be able to implement the MPEG4 video encoder on a parallel architecture, the algorithm is able to be parallelized/pipelined, by adding parameters for parallelism/pipelining support.

b) *Architecture parameters* are related to the targeted MP-SoC architecture: number of CPUs to be used, type of CPUs, HW-SW partitioning, communication topology, blocking/non-blocking protocol, arbitration type, message sizes, data width, maximum allowed data transfer latency, transfer initialization latency, etc.

4.1.3 State of the art - classical exploration flow

Several work in the literature tried to use the concept of high-level architecture exploration in order to reduce the design cost [24][25] [26][28][29]. Even if the concept is very powerful in some application domains, this is still not widely used in the case of complex design requiring MP-SoC. When estimating the performances of an entire system, it is imperative to estimate the performances of both, communication and computations.

In [24], the performance estimations are covering only the communication parts, depending on the usage of the tokens. However, from the authors' knowledge, it is impossible to obtain different communication performances for the same architecture configurations, by testing different communication times (i.e. communication initialization latency, data communication latency). Because video encoding applications are also extremely intensive in term of computations, the method presented in [24] is not sufficient. Also, their programming model based on tokens, requires already to manage explicitly the tokens locations and allocations, using very specific APIs provided by the TTL model. However, in the case of our MPEG4 encoder, these details might be irrelevant because the communication is predictable and follows a well known pattern.

In [25], the performance estimations are covering only the communication times. Also, the computation part is completely ignored. During experiments, the execution time is determined using the standard *clock* instruction in C. The problem of using the *clock* instruction is that it might return different results between simulations, depending on the workstation status (cache status, background applications, etc).

In [26], the performance estimations are covering both computations and communications, with high precision, using an ISS (Instruction Set Simulator) linked with SystemC [27]. However, this model is already at low-level, and it requires many low-level parts of the architecture to be already build (i.e. HW/SW wrappers). Computation times are determined only for the processors supported by that ISS, and changing the type of the ISS is a long design process.

In [28], the communication and computation estimation is done using a set of specific primitives which are explicitly annotated inside the code of the application (algorithm and architecture). Each of these primitive is "linked" with a high-level model of an architecture component. For example, the algorithm's memory accesses are captured by manually annotating each variable access, using specific memory *read* and *write* primitives. Behind these primitives can be found complex high-level memory models (caches, local memories, shared memories, etc). Depending on the configuration used for these memory models, these *read* and *write* primitives will capture the performances of the memory accesses. Similarly is done to capture other aspects: interconnect network, computations, etc. This approach captures almost all the aspects of a system, and it provides a high degree of generality and flexibility. However, this approach, even if it looks attractive, requires to "fine" annotations. For video applications, this approach has two significant weaknesses:

1) it requires a very long time to annotate manually the code of a video application, since such code is big (i.e. for MPEG4 encoder, around 100.000 annotations are required)

2) annotating the code leads to the impossibility to capture the compiler optimizations, for example the register vs. memory accesses optimizations (i.e. for our experiments, the compiler optimizations provided approximately 40% performance boost).

Commercial tools are already offering some early architecture explorations. One of them is the Xtensa Xplorer [29]. It captures the computation requirements very precisely, but the inter-processor communication is done via "an always available" shared memory. This tool is not sufficient in case different communication topologies or different processor models had to be used.

In all the presentedclassical exploration flows [24][25][26][28][29] (Figure 58a) , the designer implements manually the *Algorithm Specifications* starting from a set of already chosen *Algorithm Configurations*. Then, the *Architecture Specifications* is manually implemented, which should match with the *Algorithm Specifications*. In the end, the *Algorithm Specifications* and *Architecture Specifications* are combined manually, to obtain a *Combined Algorithm/Architecture Executable Model*. This model simulates the algorithm and architecture running together, and it is used for *Performance Estimations*. If these estimations are not satisfying the requirements, the designer has to modify/redesign the algorithm and/or architecture specifications. This flow has some weak points:

a) The exploration space is reduced. When having to change the algorithm and/or architecture specifications, the only things that can be changed are related to the parallelism/pipelining execution scheme of the algorithm on the architecture. Additionally, some mapping decisions [29], data organizations and communications configurations [24][25] may be changed. However, any change leads to complex modifications of the algorithm specifications and/or architecture specifications. This is increasing the time required to obtain the final product.

b) Building the *Combined Algorithm/Architecture Executable Model* has to be done manually, which is a difficult, time consuming and error prone [26]. In addition, this model has to be re-designed every time the algorithm and/or architecture specifications are changed. The simulation speed of this model depends on the used abstraction level. If the abstraction level is too low [26], the simulation speed becomes unacceptable long.

c) The performance estimation precision depends on the used abstraction level. At whatever level of abstraction, the estimation precision is a key issue. In [24][25], the performance estimations are covering only the communication. In [29], the per-

formance estimations are covering precisely the computations, but the communication performances are estimated using an "always available" shared memory. This is insufficient if another communication topology is required. In [26], the estimations are precise, but the lack of abstractions makes the simulations long.

4.1.4 Contribution of the proposed High-Level Algorithm/Architecture Exploration

The key contribution is the use of a unique *Flexible Algorithm/Architecture Model for MPEG4*, from which multiple configurations of *Combined Algorithm/Architecture Executable Models* can be obtained automatically. Our proposed exploration flow (Figure 58b) is able to cover multiple requirements:

a) The need to explore a large solution space is solved by automatically generating the *Combined Algorithm/Architecture Executable Model* from a unique *Flexible Algorithm/Architecture Model for MPEG4*, thanks to the flexible modeling style presented in the previous chapter. This approach provides the possibility to customize automatically the algorithm and build the abstract architecture, based on a set of *Algorithm/Architecture Configurations*.

b) The need of obtaining a fast simulation is covered by doing the architecture exploration at a high-level. As a result, by ignoring many low-level architecture details in the *Combined Algorithm/Architecture Executable Model*, the simulation becomes fast.

c) The need of precise simulation results is solved by using a High-Level Architecture Exploration that provides estimation results with high precision. This is done by using precise estimations for the computations and communication times, by annotating the computations time and communications time. In addition, the exploration captures the computations and communications running together, to estimate the performances of the entire system. If these performances are not satisfying the requirements, the designer will have to change the algorithm and/or architecture configurations, or rarely, to modify *the Flexible Algorithm/Architecture Model for MPEG4*.

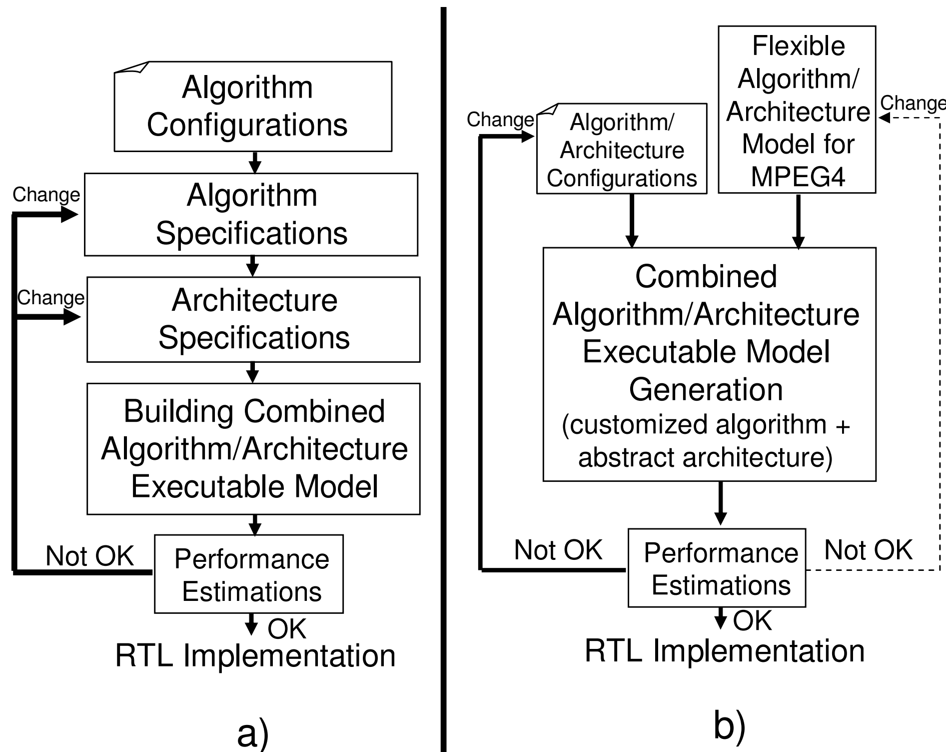


Figure 58. Classical exploration(a) vs. Proposed exploration(b)

This architecture exploration is achieved at a high-level using the *Flexible Algorithm / Architecture Model for MPEG4*, from which multiple configurations of *Combined Algorithm/Architecture Executable Models* are automatically obtained. This decreases the time needed to obtain and test multiple architecture configurations. This helps to find in a much shorter time the correct algorithm and architecture configurations for the MPEG4 encoder into MP-SoC. Since the entire exploration is done at a high-level, there is no need to implement and test multiple MP-SoC architectures at a low-level, which is drastically reducing the design cost. The proposed approach was applied successfully for the generation of several configurations of MPEG4 encoders.

4.2 High-level Algorithm/Architecture Exploration flow for MPEG4

This section describes in detail the High-Level Algorithm/Architecture Exploration flow (Figure 59) used for the MPEG4 video encoder.

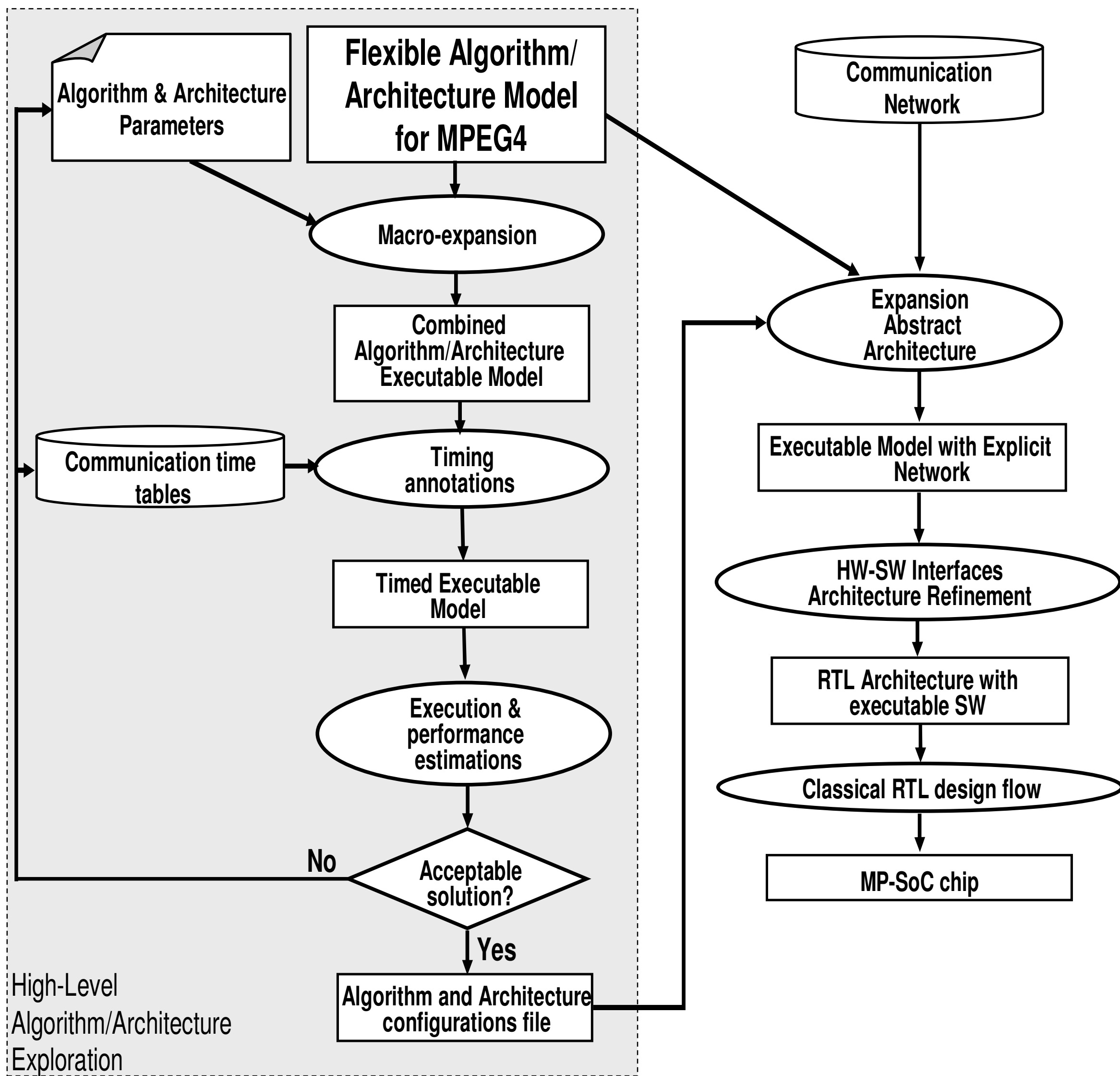


Figure 59. Detailed representation of the design flow

This flow is composed of 4 major phases: (1) generating the *Combined Algorithm/Architecture Executable Model*; (2) obtaining the *Timed Executable SystemC Model*; (3) performance estimation and reconfiguration; (4) building the final RTL architecture. Only the first 3 phases are part of the high-level architecture exploration and will be detailed. Presenting the phase of building the final RTL [47] architecture will be explained later in this document.

4.2.1 Obtaining the Timed Executable Model required for performance estimations

For performance estimations, a *Timed Executable Model* is used. This model is obtained in two steps:

a) The *Flexible Algorithm/Architecture Model for MPEG4* is macro-expanded to obtain the *Combined Algorithm/Architecture Executable Model*. This initial model is used to compute the delays for the computations. Delays are obtained using a classical approach consisting of executing the C/C++ code of the tasks on an Instruction Set Simulator (ISS) of the targeted CPU. This gives an approximate number of clock cycles required by the different tasks, independently of the communications. The obtained times are not 100% accurate, because the scheduling effect is not captured with this approach. The experiments show that the performance estimation is precise enough for our architecture exploration, as will be shown later in this paper.

b) The obtained computational delays are inserted in the *Combined Algorithm/Architecture Executable Model*, by inserting time annotations for computations and communications into the tasks code. The time annotations for computations are done by inserting WAIT calls to simulate the computations delays [33]. The time annotations for communications are embedded within the MPI-SystemC. Communication times are given for different communication configurations (message size, data width, protocol, transfer latencies). In this work, these times are given as parameterized delay functions associated to each MPI primitive. The execution of each primitive is divided into 3 steps: initialization (initial synchronizations), transfer (for each data) and closure (communication release). An execution time is associated to each of these steps, allowing a detailed viewing/analyzing of the communication behavior.

By annotating time indications on the *Combined Algorithm/Architecture Executable Model,* the *Timed Executable Model* is obtained. Figure 60 shows the code of the resulting time annotated C/C++ code for the *MainDivX1* task, which contains the time annotations for the computations, and the time annotations for the communications (integrated into MPI-SystemC HLPPM). The values for these delays are captured in tables and depend on

the configurations chosen for the computations (i.e. CPU model, CPU clock frequencies) and communication primitives (i.e. data width, message sizes, latencies). The resulted *Timed Executable Model* allows performance estimations at a high-level of abstraction for different algorithm/architecture configurations.

```
//------------------ MainDivX task ---------------------------------------------
EXTERN *image_memory1,height1, length1, top_border1, left_border1,
            bottom_border1, right_border1,*result1;

void MainDivX1_MAIN (void)
{
  //initialization of computations
  MainDivX_INIT (&image_memory1, height1, length1);
  WAIT(13.224);

  //infinite loop for every frame
  while (1)
  {
        //data_receive_communication from the Splitter
        MPI_Recv(this,&image_memory1,sizeof(image_memory1),
                    32,SPLITTER_ID,22,MPI_COMM_WORLD,&status);

        //calls the function with flexible computations
       MainDivX_COMPUTE (&image_memory1,height1, length1, top_border1,
                              left_border1, bottom_border1, right_border1,&result1);
       WAIT(2.312.564);

       //send_results_communication to the VLC
       MPI_BSend(this,&result1,sizeof(result1),32,
                      VLC[0]_ID,22, MPI_COMM_WORLD);
}
```

Figure 60. Obtained time annotated code for MainDivX1 task

4.2.2 Performance estimations and architecture exploration

By compiling and executing the *Timed Executable Model*, performances can be measured using the function *sc_simulation_time()* after encoding every frame. The execution of this timed model gives an estimation of performances. The obtained performances can be represented using performances diagrams (graphic tables), and they include the time annotated computations and communications running together. In addition, in the same graphic can be displayed for comparison the performances measured for multiple different algorithm/architecture configurations, to help the designer to take the next decisions. Figure 61 gives the estimated performance for the execution of MPEG4 for 25 frames of QCIF

(176x144) video resolution movie, using 1,2,4,8,16 and 32 CPUs ARM7 [36] at 60MHz for the *MainDivX* tasks and 1 CPU for *VLC*.

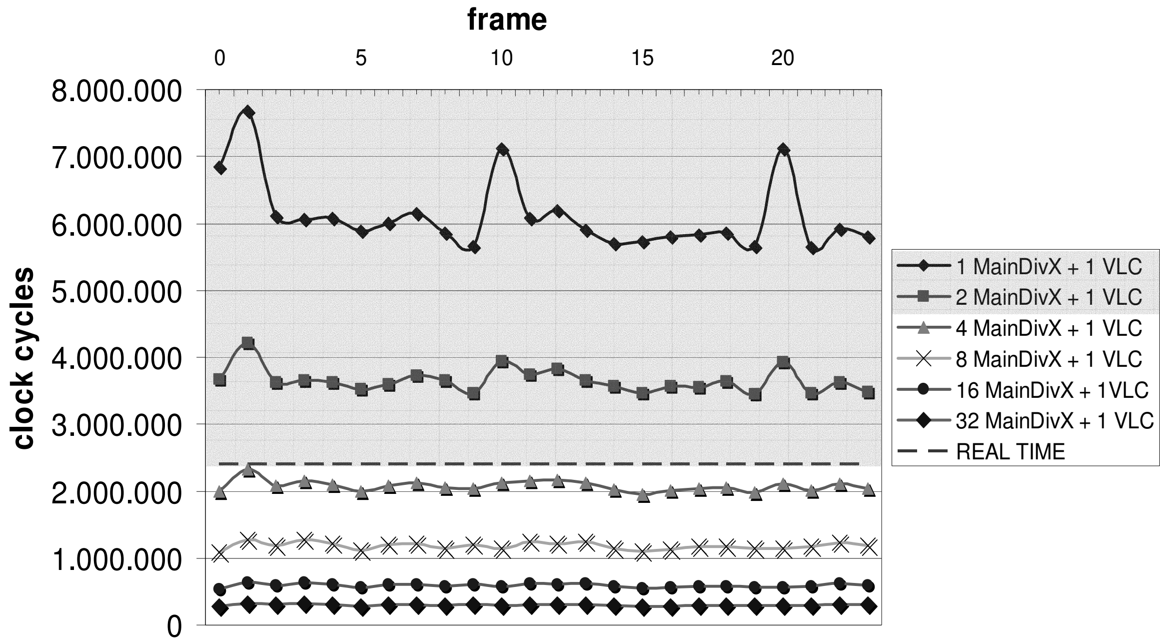


Figure 61. Performance estimated for QCIF, using ARM7, 60MHz

As benchmark movie we used one second (25 frames) of „snow-show" movie (similar to what the TV receivers show when there is no signal on the antenna). This represents the worst-case scenario for the MPEG4 application. Consequently, the real-time encoding for any other input case is assured. In addition, the used search area for the Motion Estimation is 16x16. The reason is that previous research experiments showed that for QCIF (176x144) and CIF (352x288) resolutions, the full search area can be discarded, because the compression gain does not pay for the performance loss. However, this is not true for higher video resolutions.

Figure 62 shows the estimated performances using ARM946E-S, 4kI$, 4kD$ CPUs at 60MHhz [36]. In order to achieve real-time, maximum 2.400.000 cycles are allowed for the compression of 1 frame. From Figure 61 and Figure 62, we can determine that minimum 5 ARM7 or 2 ARM946E-S processors are required to achieve real-time.

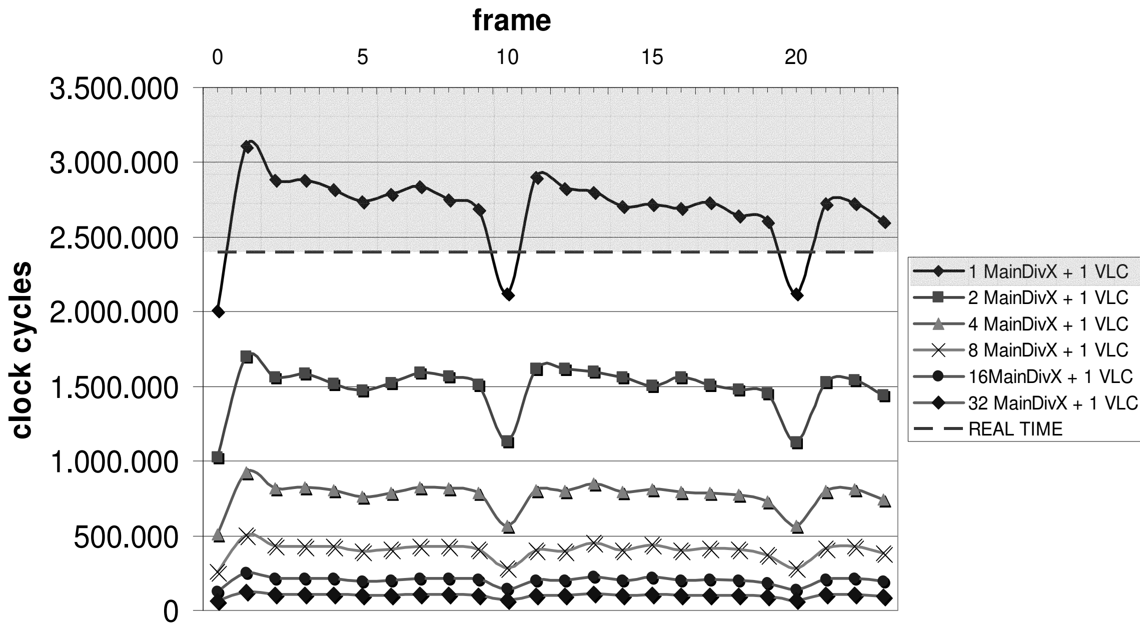


Figure 62. Performance estimated for QCIF, with ARM946E-S CPUs, 4kI$,4kD$, 60 MHz

Different curves of these simulations are obtained doing a new macro-expansion of the *Flexible Algorithm/Architecture Model for MPEG4* with different parameters. Besides number and types of CPU, several other parameters may be explored. For example, the communication may be explored via message size, data width, protocols and latencies.

4.2.3 Validation of the high-level simulation results

The architecture exploration allows us to fix a set of parameters that will define the number of required CPUs, models of CPUs, communication protocols, message sizes, maximum latencies, etc. These parameters will be followed during the architecture implementation. Figure 63 shows an example of obtained configurations.

CPUs Number → 5 (4 MainDivX + 1 VLC)
IPs Number → 2
Splitter → IP
$MainDivX_1$ → CPU (ARM7)
$MainDivX_1$ → 60MHz
$MainDivX_1$ → no cache
… the same for the other 3 MainDivX
VLC_1 → CPU (ARM7)
VLC_1 → 60MHz
VLC_1 → no cache
Combiner → IP
Splitter-$MainDivX_1$ send protocol → Blocking
$MainDivX_1$-Splitter recv. protocol → Non-Blocking
Splitter-$MainDivX_1$ burst size → 128 bytes
Splitter-$MainDivX_1$ data_width → 32 bits
Splitter-$MainDivX_1$ init_latency → 2 cycles
Splitter-$MainDivX_1$ data_latency → 3 cycles
$MainDivX_1$-VLC_1 send protocol → Blocking (FIFO 810bytes)
VLC_1-$MainDivX_1$ recv. protocol → Blocking
VLC_1 recv. arbitration → AnySource
$MainDivX_1$-VLC_1 burst size → 810 bytes
$MainDivX_1$-VLC_1 data_width → 32 bits
.... similar for the other modules

Figure 63. Example of architecture configuration file

The communication time tables are following the performances/requirements of a DMS communication network [40] which we've targeted to be used during the architecture implementation at RTL level. More details regarding the DMS will are presented in the fifth chapter. However, different other communication time tables can also be used for other communication networks. In [61], were estimated the encoding performances at high-level when using different communication networks: DMS, AMBA and Octagon.

4.3 Experiments and results analysis

This section presents the experimental results obtained for the architecture exploration of the MPEG4 application for QCIF (176x144) and CIF (352x288) video resolution at 25 frames/sec, using ARM7 and ARM946E-S processors running at 60 MHz.

4.3.1 Performance estimated for QCIF, using ARM7, 60MHz

For QCIF video resolution using only ARM7 processors running at 60MHz, the resulting architecture required 5 processors: 4 processors for 4 *MainDivX* tasks, and 1 processor for 1 *VLC* task. Simulation results for 1, 2, 4, 8, 16 and 32 CPUs for *MainDivX* and 1 for *VLC* was shown in Figure 61. The architecture configurations obtained after the High-Level Algorithm/Architecture Exploration was presented in Figure 63.

4.3.2 Performance estimated for QCIF, using ARM946E-S, 4kI$, 4kD$, 60MHz

For QCIF video resolution using only ARM946E-S processors running at 60MHz, using 4kbytes cache for instruction and 4kbytes cache for data, the resulted architecture required only 3 processors: 2 processors for 2 *MainDivX* tasks, and 1 processor for 1 *VLC* task. Simulation results for 1, 2, 4, 8, 16 and 32 CPUs for *MainDivX* and 1 for *VLC* was shown in Figure 62.

4.3.3 Performance estimated for CIF, using ARM7, 60MHz

For CIF (352x288) video resolution, the same experiments were conducted. In case of using only ARM7 processors, the architecture required 23 processors: 20 for *MainDivX* task and 3 for *VLC* task. Initially, 16 processors were sufficient for the *MainDivX* tasks, but the communication degradation made this impossible. Therefore, we opted for more processors, instead of choosing a "super" communication. Figure 64 shows the performance diagram using ARM7 CPUs at 60 MHz.

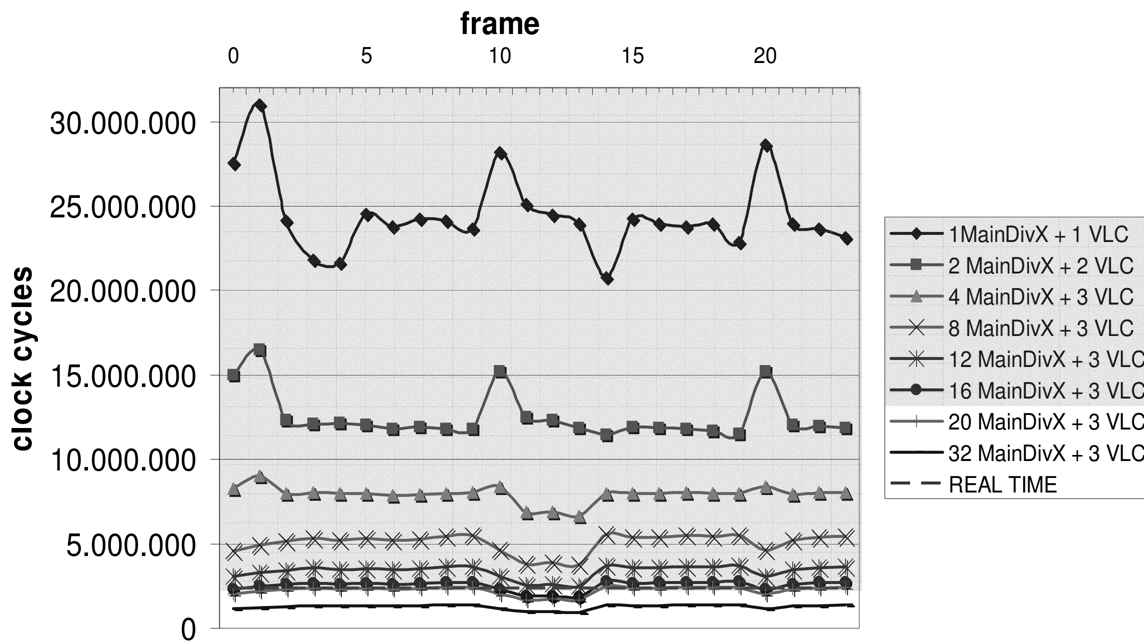


Figure 64. Performance estimated for CIF, using ARM7, 60MHz

4.3.4 Performance estimated for CIF, using ARM946E-S, 4kI$, 4kD$, 60MHz

Figure 65 shows the obtained performance diagram for CIF video resolution, using ARM946E-S processors. In order to achieve real-time functionality, 10 ARM946E-S processors at 60 MHz were required: 8 for *MainDivX* tasks and 2 for *VLC* tasks.

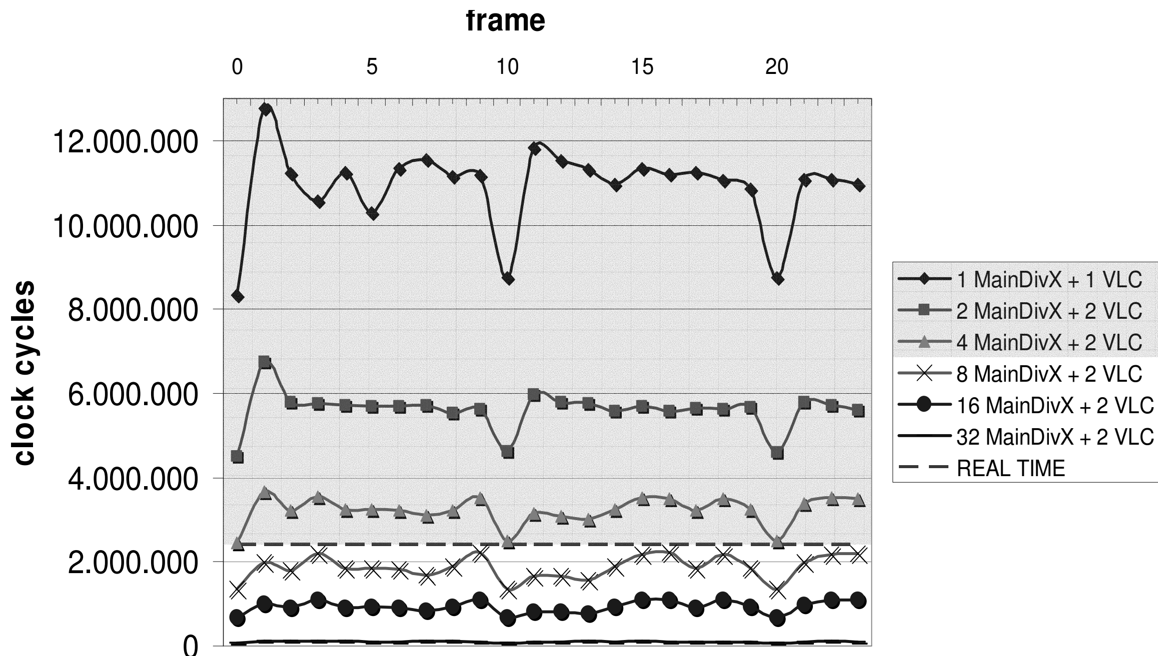


Figure 65. Performance estimated for CIF, using ARM946E-S CPUs, 4kI$, 4kD$, 60 MHz

4.3.5 Results analysis

For higher resolutions of MPEG4, ARM7 and ARM9 are not powerful enough. Additionally the amount of embedded memory gets higher than the amount allowed by the current technology. In term of memory, an off chip memory may be required which might change the required interconnect. For computations, powerful DSP or VLIW processors are needed, or HW instructions [19][37][38].

To validate the precision of the High-Level Algorithm/Architecture Exploration, an RTL architecture was built more or less manually, using one of the architecture configurations obtained during the high-level architecture exploration. Figure 66 shows that the precision of the performance estimations, obtained during the high-level architecture exploration, are close to the one measured at RTL level. The communication infrastructure used in

case of the RTL architecture, is a customizable data transfer architecture [40] providing high performances, that can easily be configured with the communication parameters obtained by architecture explorations.

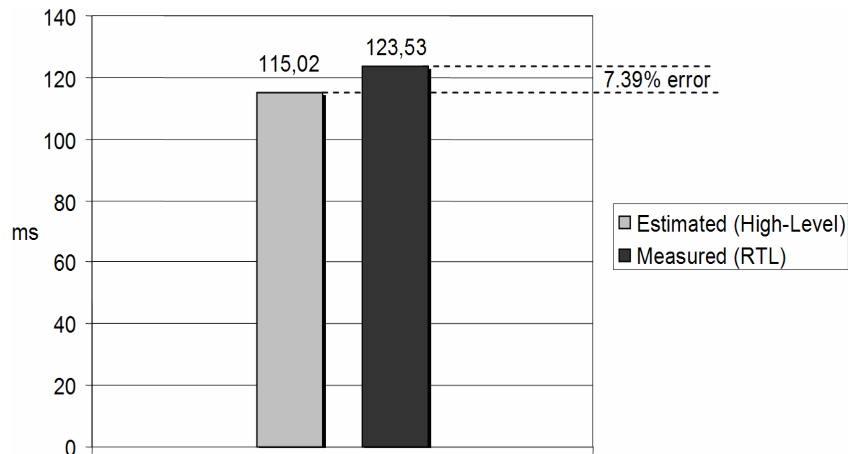


Figure 66. Estimated vs. Measured performance precision
[QCIF, 1 frame, 2 ARM7 CPUs (1 MainDivX + 1 VLC), 60MHz]

To compress 1 frame at QCIF resolution, using 2 ARM7 CPUs (1 *MainDivX* + 1 *VLC*) running at 60MHz, the high-level estimations predicted that 115.02 ms are required (in Figure 61, 6.82 million cycles is equivalent to 115.02 ms). The performance measured for the obtained RTL architecture proved that 123.53 ms were required to compress 1 frame. The 7.39% precision error comes from the impossibility to capture with our proposed high-level estimations, the performance degradations of the:

a) OS (scheduling, service calls latencies induced by the API calls)

b) Interconnect between the CPU buses and communication infrastructure (the conflicts for local bus grant between the CPUs and the Network Interfaces).

c) HW/SW Wrappers

By using the proposed High-Level Algorithm/Architecture Exploration, different and already validated architecture configurations were quickly explored, even for a big number of CPUs, complex communications and different algorithm configurations (video resolution). This process dramatically shortened the time required to do the architecture exploration. As an example, in case of 25 frames of QCIF resolution video and using ARM7 proc-

essors running at 60MHz, approximately 15 minutes were required to generate the *Timed Executable Model*. The simulation for 25 frames took approximately 2 minutes. Exploring one architecture solution takes less than one hour. This is the time required just to simulate one frame at RTL level. Approximately 25 hours were required to simulate 25 frames using the RTL model. Therefore, the high-level performance estimations error of less than 10% compared with the low-level performance measurements is more than acceptable, considering the gain of design time. In these experiments we have used a Pentium4, 3GHz, 1Gbytes RAM, using Linux Mandrake 9.2.

Currently, in order to adapt this approach to other video applications, the *Flexible Algorithm/Architecture Model for MPEG4* must be adapted manually. In addition, if the communication network is changed (using topologies different from the ones already supported), the *MPI-SystemC HLPPM* model needs to be adapted to the new communication constraints. Automating these tasks or finding a method to reduce the effort needed for applying the proposed design paradigm to different applications are, in our opinion, open research subjects, and we have chosen to leave this point for future works.

To improve the estimation precision, we intend to use in the future the approach presented in [62] which proposes to use an general abstract CPU architecture model, instead of simply abstracting the CPU by a SystemC module. This approach will allow us to capture (even if still not 100% precisely) the low-level details (i.e. HW/SW wrappers, grant on CPU bus, etc) which in current approach are completely abstracted (missing).

4.4 Conclusions

The objective of this chapter is to present the proposed High-Level Algorithm/Architecture Exploration flow. This flow is used to explore different algorithm and architecture configurations to be chosen from a large solution space. The flow uses a unique *Flexible Algorithm/Architecture Model for MPEG4*. For different algorithm and architecture configurations, this model can be macro-expanded to obtained different *Combined Algorithm/Architecture Executable Models*. These models capture the behavior of both algorithm and architecture already customized for different algorithm/architecture configurations. *Timed Executable Models* are used for performance estimations. These are

obtained by annotating the computations and communications delays into the *Combined Algorithm/Architecture Executable Model*. These delays depend on the used algorithm and architecture configurations.

As a result, multiple MPEG4 encoders on MP-SoC were successfully explored, using different algorithm and architecture configurations. The obtained estimations precisions proved to be very close to the performances measured at RTL level. This assured the feasibility of this approach. Since the entire exploration is done before implementing the MPEG4 encoder at RTL level on MP-SoC, this flow helps to reduce the design cost.

5 MPEG4 video encoder architecture implementation

This chapter presents the flow used for the implementation of the MPEG4 encoder on MP-SoC. This flow uses as input a high-level model for MPEG4 configured with the algorithm/architecture configurations determined during the high-level algorithm/architecture exploration. In this model, all the low-level details are completely abstracted. The flow is in charge of generating all these low-level details. Since the flow's steps are automated, this approach is drastically reducing the design time needed to implement the MPEG4 encoder on MP-SoC. Using this flow, different MPEG4 encoders were implemented, on completely new RTL architectures, or on an existing MP-SoC platform.

5.1 Introducing the MPEG4 video encoder architecture implementation

5.1.1 Difficulties of implementing the MPEG4 video encoder on an MP-SoC architecture

The implementation of the MPEG4 video encoder on MP-SoC architecture (RTL level) requires a long design time, because the designer has to manage several difficulties:

a) The first aspect is related to the configurations of the architecture that has to be implemented. These configurations are related to both, high-level (portioning, mapping, communication network topology, etc) and low-level details (HW/SW Interfaces: Operating Systems [41], CPU SubSystems, Wrappers[45]) of an architecture.

   The high-level details can be found using the previously proposed High-Level Algorithm/Architecture Exploration flow. In case of low-level details, the designer should use a tool capable of configuring them automatically, depending on the application's requirements.

b) Another aspect is related to the actual architecture implementation phase. In classical approaches, the designer has to implement manually the complete RTL architecture. This is an exhaustive approach, which requires long design time, involves many bugs, and many design loop-backs.

   The solution is the implementation of the architecture at a high-level that abstracts many of the low-level details (HW/SW Interfaces), by implementing an abstracted architecture model of the targeted RTL architecture. For obtaining the final RTL architecture, the designer should use a component based approach [39], in order to refine (generate) the low-level details which were abstracted. This generation should be as automatic as possible.

c) When designing MPEG4 video encoders on MP-SoC, there are always two possibilities: either the MPEG4 video encoder will be implemented on an existing architec-

ture, or a complete new MP-SoC architecture has to be built for the MPEG4 video encoder. Using separate flows for these two approaches becomes a difficult aspect to be managed by the designer. One reason is the need of becoming familiar with these flows. Another reason is the need of implementing differently the same architecture specifications for each flow. Finally, porting required libraries from one flow to another can become a difficult thing, sometimes even impossible.

The solution would be to use a unique flow for both cases. In both cases, the flow has to use the same input specifications, the same abstract architecture model, and the same tools. In the end, the flow has to be capable of obtaining different results. These depend on the case if the MPEG4 video encoder is implemented on an existing architecture, or on a completely new architecture.

5.1.2 Principle of the proposed flow used for implementing the MPEG4 video encoder on MP-SoC

The proposed flow uses as input specifications the *Flexible Algorithm/Architecture Model for MPEG4*, as the one used during the High-Level Algorithm/Architecture Exploration. Thus, we benefit from the fact that these specifications were already validated and debugged.

Using the *Flexible Algorithm/Architecture Model for MPEG4* and the algorithm/architecture configurations found during the High-Level Algorithm/Architecture Exploration, an abstract architecture model can be automatically generated (Figure 68), called *Executable Model with Explicit Network*. This model is generated using the same macro-expansion approach used to obtain the *Combined Algorithm/Architecture Executable Model* during the High-Level Algorithm/Architecture Exploration. The only difference is that instead of using the MPI-SystemC HLPPM, we will use an explicit communication network configured automatically with the configurations found during the high-level exploration. This *Executable Model with Explicit Network* is still a high-level model, in which all the low-level details are abstracted. More details about this model will be presented later in this document.

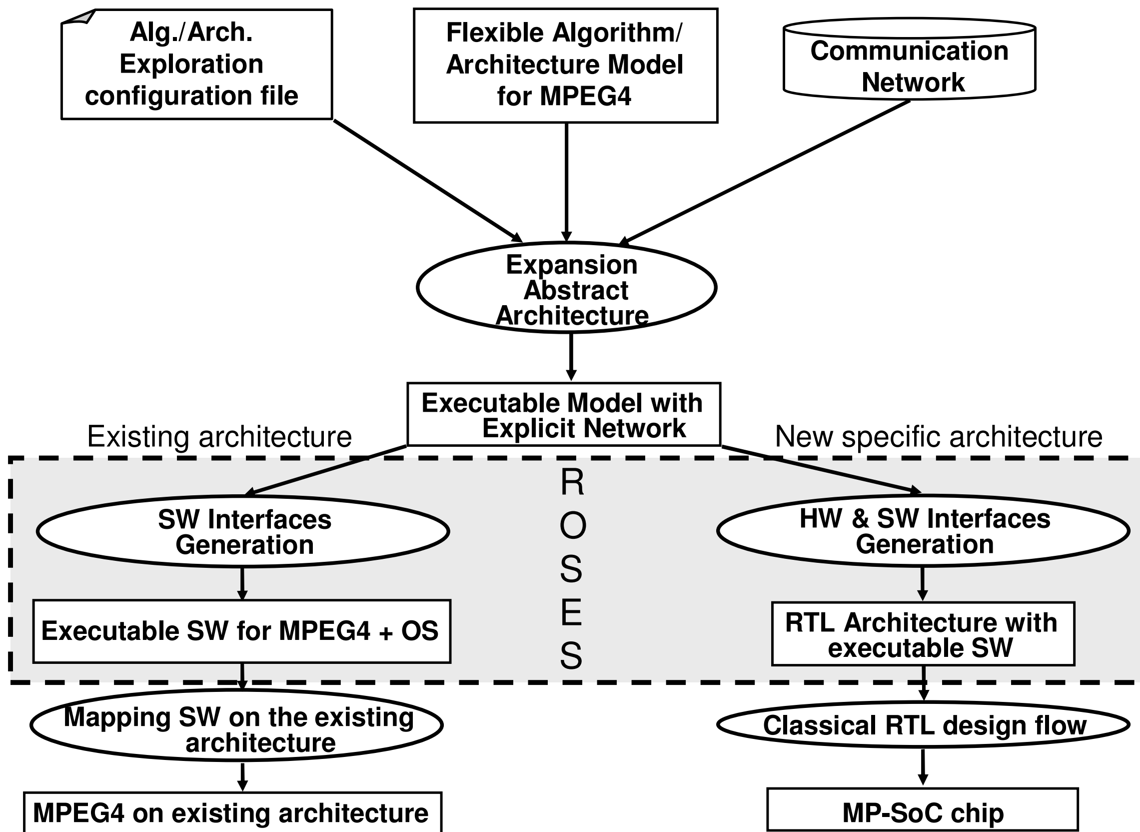


Figure 67. Proposed flow for implementing the MPEG4 encoder on MP-SoC architectures

In order to obtain the final implementation, we propose the use of ROSES flow, designed by TIMA-SLS Group, for refining the required abstracted low-level details (HW/SW Interfaces). Depending on the targeted architecture case for the MPEG4 video encoder, there are two possibilities:

a) Existing architecture: since in this case, the architecture is already implemented, the only low-level details that have to be refined are the SW interfaces, which represent the Operating Systems (OS) of each CPU of this architecture.

After obtaining all the OS by the use of ROSES flow, along with the tasks of the MPEG4 encoder, the entire MPEG4 video encoder application can be mapped on the existing architecture, and executed.

2) New specific architecture: in this case, the RTL architecture has to be obtained, along with the SW application (OS+MPEG4 tasks). The OS for each CPU are obtained using the same principle as for an existing architecture. Additionally, the HW

interfaces (CPU-SubSystems, HW Wrappers) required by the RTL architecture have to be generated. The result will be the synthesizable RTL architecture with the executable SW application.

These results will be integrated on silicon, using a classical RTL design flow, in order to obtain the final MP-SoC chip for the MPEG4 video encoder. This last phase will not be covered in this document.

5.2 ROSES: a component based approach used for Hardware/Software integration

5.2.1 Representing the ROSES flow used for Hardware/Software integration

The ROSES flow developed at TIMA Laboratory-SLS Group is used for the design of single-network MP-SoC at RTL level starting from high-level abstract architecture models using heterogeneous components. This flow solves the problems of the automatic generation of the HW and/or SW interfaces required by the architectures at RTL level. In addition, it allows us to validate the obtained results during the flow at different levels of abstraction. A simplified diagram of the ROSES flow is presented in Figure 68.

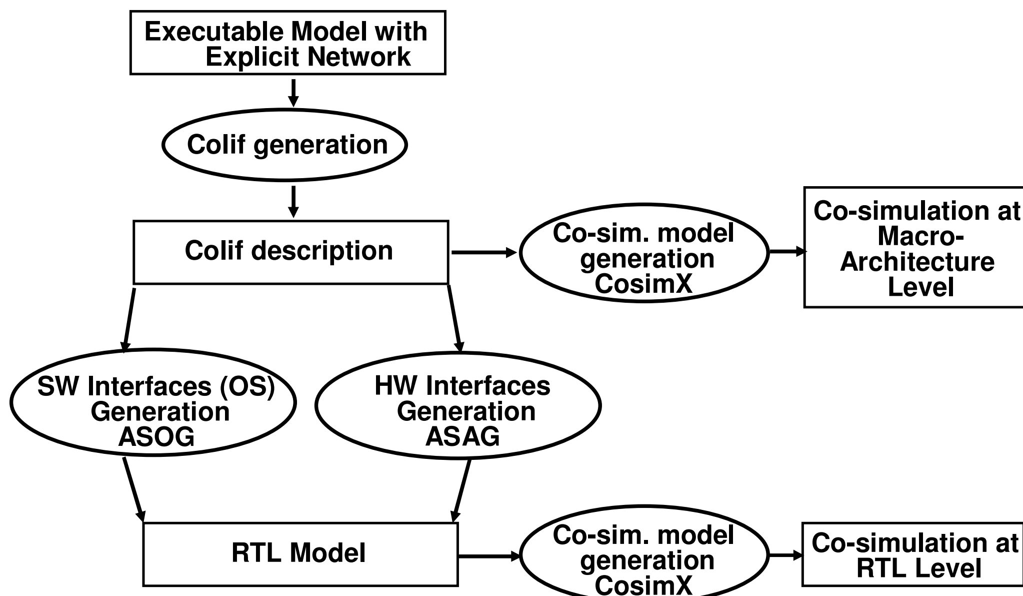


Figure 68. Representation of the ROSES flow used for HW/SW interfaces generation

The flow uses as input model an abstract architecture, called *Executable Model with Explicit Network*. This is made of multiple components described in SystemC. In this model, all the low-level details are completely abstracted, like OS, CPU-SubSystems and HW Wrappers.

The input model is transformed into an intermediary architecture description model, described using Colif language, used by all the tools of this flow. This language allows the description of a heterogeneous system containing modules at different levels of abstraction.

The SW interfaces, also known as Operating System (OS), are generated using the ASOG tool [41]. The HW interface generations, which consists of building the CPU-SubSystems and HW wrappers, are generated using the ASAG tool [42]. During the flow, the architecture can be executed and validated at different abstraction levels using the Co-simX tool [43].

As a result, the ROSES flow generates for the initial abstract architecture (*Executable Model with Explicit Network*) all the low-level architecture details, in order to obtain the final RTL architecture (Figure 69).

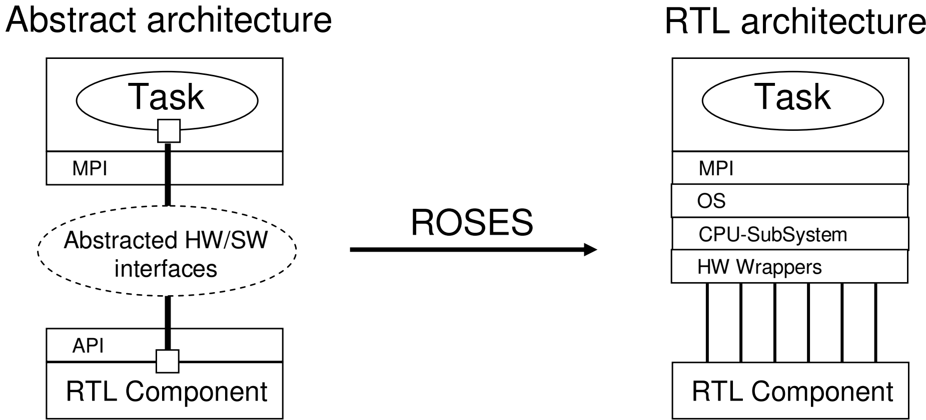


Figure 69. Refining an Abstract architecture to RTL architecture using ROSES

5.2.2 Describing the COLIF representation model

This language allows the description of a heterogeneous system containing different types of modules (CPUs or IPs), and it is saved into an XML type of file. With Colif, the

system description is done independently of its behavior. This allows modeling a heterogeneous system by using its level of abstraction, description language (SystemC in our case) and their type (CPUs or IPs). This language describes an architecture based on 4 concepts: modular structure, different levels of abstraction, virtual components and hierarchical ports.

a) modular structure:

Colif describes a system under the form of an ensemble of interconnected objects, which are of 3 types: modules, ports and nets [44]. An object is composed of two parts: an interface called *entity*, and a *contain*. Two objects can be interconnected by connecting their corresponding *entities*. The *contain* of an object points to a reference to an actual behavior, or to another object. This “linking” possibility allows the development of modules, hierarchical ports and hierarchical nets. This way, into a module, it is possible to instantiate complete subsystems formed by other modules, ports or nets (Figure 70).

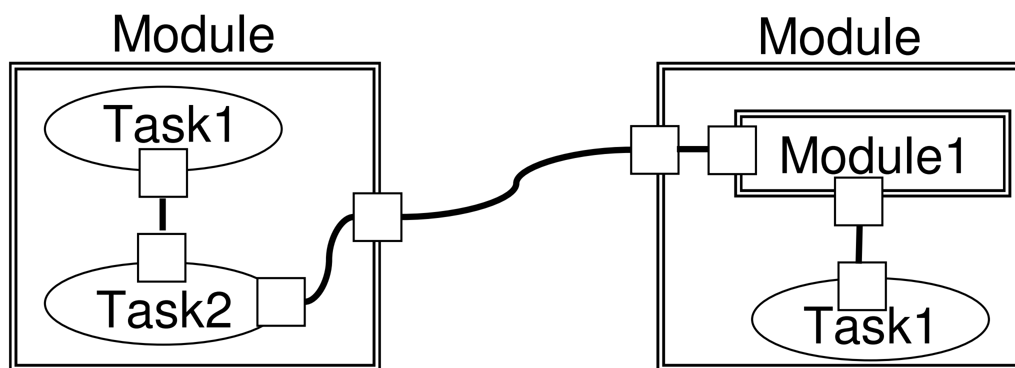


Figure 70. Example of architecture containing modules, ports and nets

The module represents a hardware (IP) or software (CPU) component. Its *entity* gives the types of the ports through which it communicates. The *contain* points to a behavior which can be a typical C/C++ file, or another subsystem which can be formed by other modules, ports and nets.

The port represents a communication point of a module. Its *contain* is composed of two parts: one part is in relation with the interior of the module, the other with the exterior of the module. These parts can be either hidden, or point to a behavior or to other ports.

The net represents a connection between two ports. In this case, the net can also point to an actual behavior, or to other net instances.

b) different levels of abstraction

In Colif, a component can be described at two levels of abstraction: Macro-Architecture level, and RTL level.

At Macro-Architecture level, the communication is done using abstract wires, which contain drivers in charge of managing the communication protocols. Thus, at this level, only the protocol and the topology of the communication are chosen. In addition, the time granularity is at the transaction level.

At RTL level, the communication is done using physical wires and buses. At this level, all the ports and interfaces are refined, and cycle accurate adapters [46] are used to “link” different communication protocols. In this case, the time granularity is cycle accurate.

c) virtual components and hierarchical ports

To allow the interconnection between the components described at different levels of abstraction, the Colif uses the concept of virtual components. Each of these virtual components encapsulates one or more modules.

A virtual component contains two communication interfaces: internal and external. The internal interface is adapted at the level of abstraction used by the module encapsulated in this virtual component (Figure 71). The external interface is adapted at the level of abstraction used by the channel linked to the virtual component. For communication, the internal interface uses internal ports, while the external interface uses external ports. A virtual port encapsulates one or more of these internal/external ports, in order to form a communication protocol.

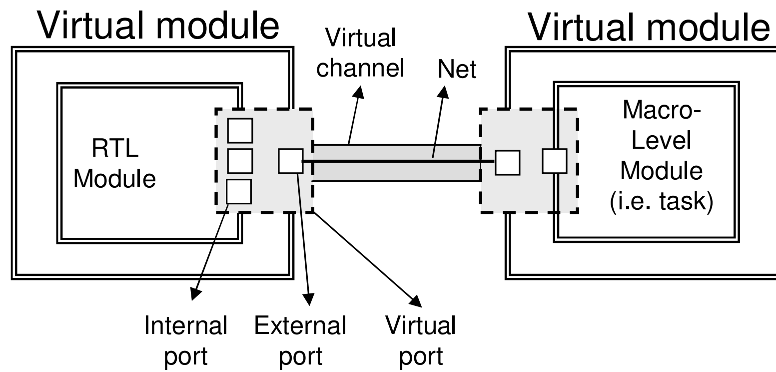


Figure 71. Structural representation of a virtual component

5.2.3 Describing the ASOG tool used for OS generation

This tool generates one application specific OS for each processor. The OS generation consists of assembling the required OS, using a library which contains a set of macrocodes files for each of the services of an OS, and a service dependency graph used to determine which services will form the current generated OS.

1) macro-code files for the services of an OS

These files are written using a macro-language. From these files are macro-generated the customized files for the current OS. This macro-generation approach is identical to the one used during the previously presented High-Level Algorithm/Architecture Exploration.

2) service dependencies graph

The service dependency graph is described using a structural description language, called LiDeL (Library Description Language), developed at SLS group. It is composed of a set of data structures manipulated by some APIs. The description is composed of 3 items:

a) *elements* represent an OS part, and they are the basic components of an OS. They represent a non-specialized component, which is not yet dedicated for a particular architecture case.

b) *services* represent a system functionality. It is an abstract term, which allows dividing and structuring the behavior of an OS. The *services* are provided by *elements*, but an *element* may also require a *service* from another *element.*

c) *implementations* represent a particular behavior description. An *element* can have multiple *implementations*. Each *implementation* corresponds to a part of generic code of an OS.

The ASOG tool uses as input the system description in Colif, the tasks code, the LiDeL library and the library containing the elements written as macro-code. The Colif description contains the services needed by the application, along with the parameters needed for these services (Figure 72).

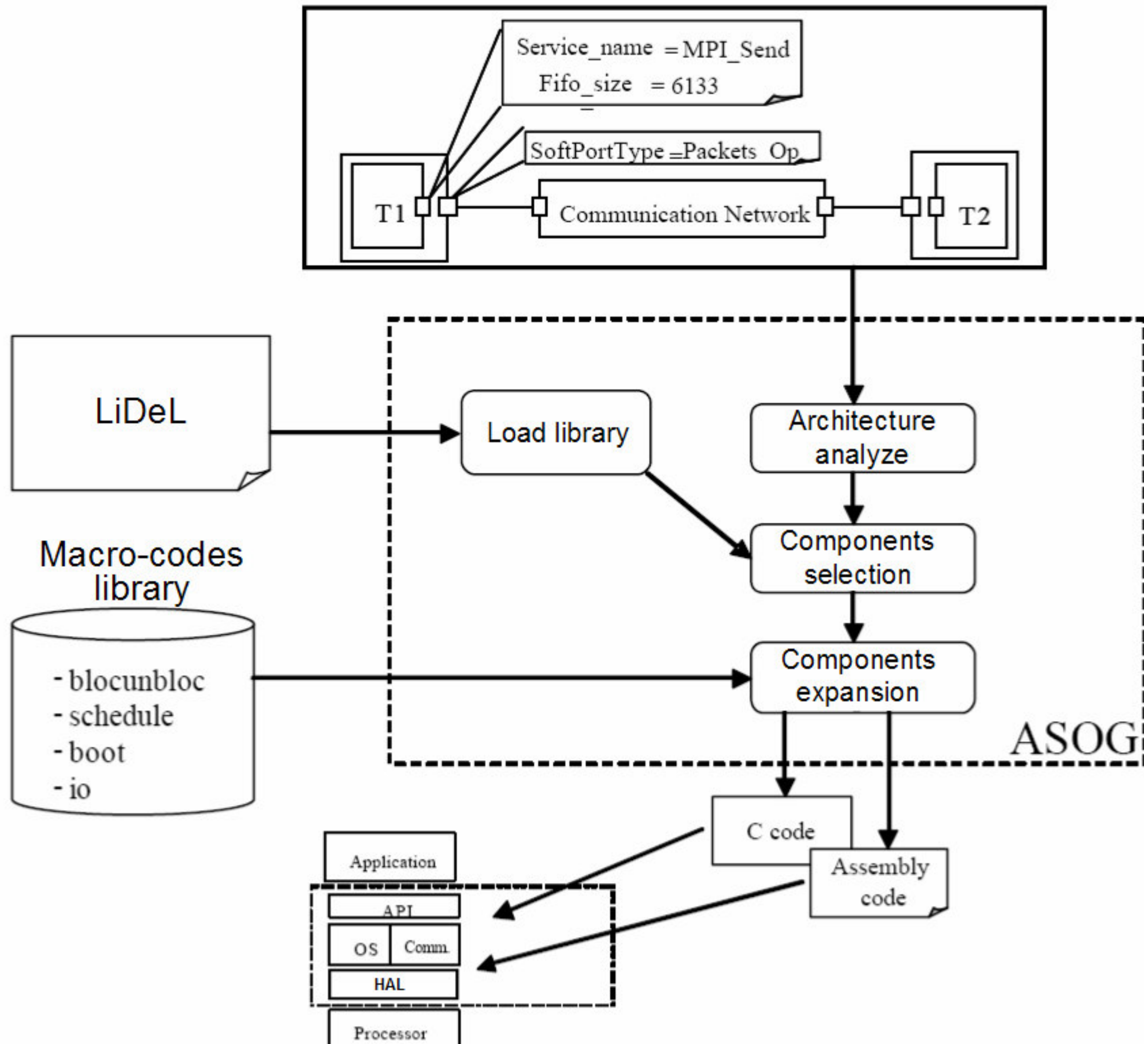


Figure 72. Representation of the flow used by ASOG tool for OS generation

When SW tasks require a *service* (i.e. services for MPI communication), the ASOG tool starts crossing the service dependency graph from the required *service* down to the low-level *services*. Based on the crossed *services*, the ASOG will macro-generate the files

for the *implementations* of the *elements* associated to these *services*. The generated files are C/C++ or ASM files. The ASOG also generates the required compilation Makefile scripts, along with some log files useful for debugging the OS generation process.

5.2.4 Describing the ASAG tool used for hardware generation

The concept behind this tool is similar to the one used by ASOG, except that in this case the HW components (which were abstracted in the virtual architecture) are assembled. Hence, the ASAG uses a library containing a set of generic structures for the interfaces, and a behavior library containing the macro-codes of their required components.

a) structure library : this library contains two types of components described in Colif. One of them is the component forming the local architecture of the current processors, also known as CPU-SubSystem. The other components are targeted for the communication adapters, also known as HW Wrappers.

b) behavior library: this library contains a set of files described in macro-code, and contains the generic behavior of the interfaces. For each of these interfaces, multiple behavior codes can be associated, depending on their targeted description language: i.e. SystemC, VHDL.

The ASAG flow (Figure 73) consists of 5 phases:

a) analysis of architecture description: the ASAG parses the Colif system description (XML file) and gathers all the configuration parameters from it.

b) load structure library: this allows us to know the availability of the architectural resources required for the current architecture refinement. This library contains a set of different architecture parts, formed from one component or multiple interconnected components. For each component, its behavior is pointed into the behavior library.

c) module refinement: based on the configuration of the module (virtual module) contained in the Colif description, the module is replaced with a local architecture (described also in Colif) contained into the structural library. Based on the parameters

used for its virtual ports, a set of channel adapter interfaces are instantiated, in order to adapt the ports of the local architecture with the ports of the channel connected to this module [45].

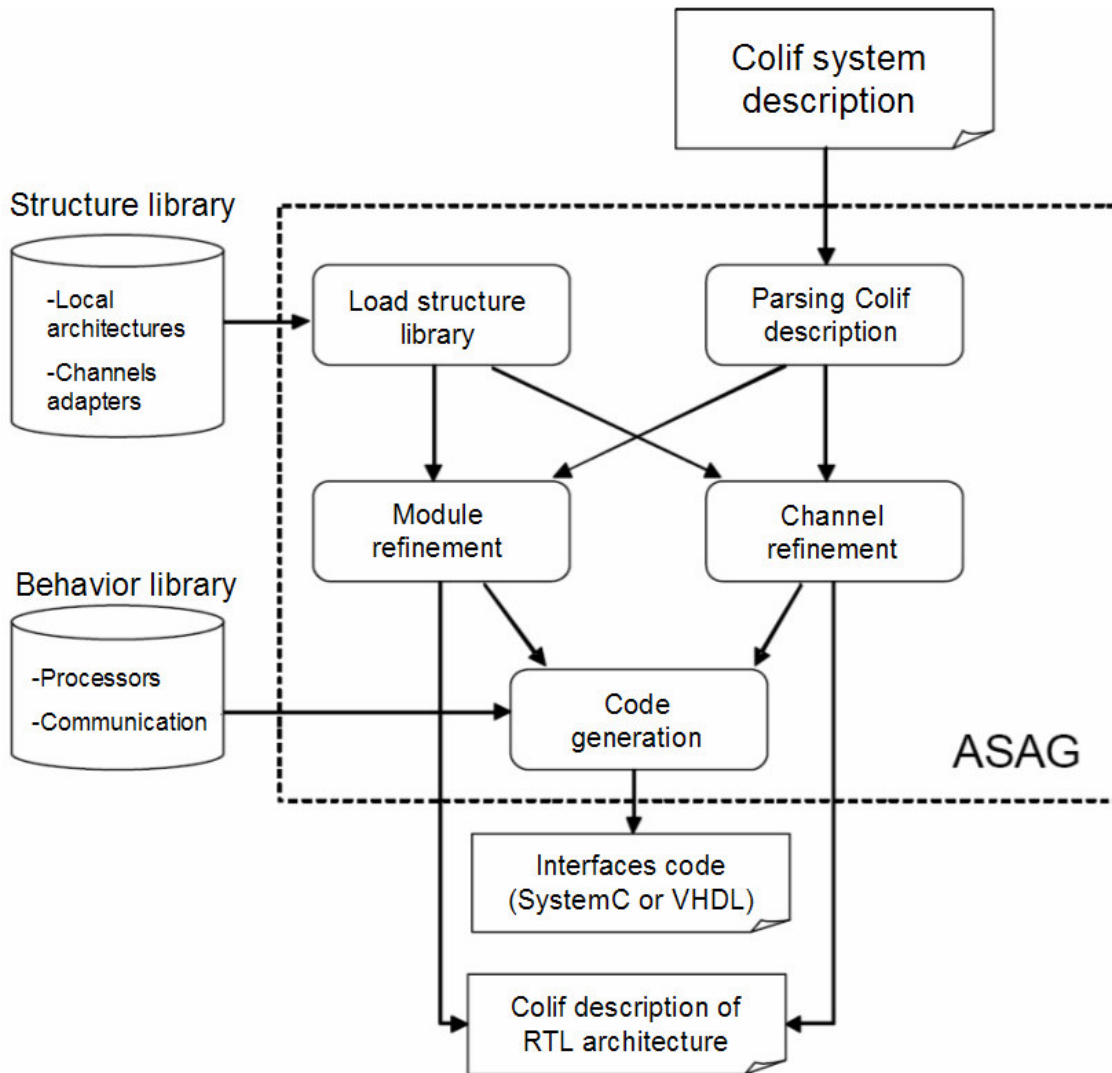


Figure 73. Representation of the flow used by ASAG tool for HW generation

d) channel refinement: by analyzing all the ports of the channel, it is possible to find out the characteristics of the refined channel to be chosen from the library.

e) code generation: to each component of the local architecture and channel adapters corresponds a macro-code generic behavior implementation contained in the behavior library. Even if this macro-code is not representing an executable or synthesizable component, it can become after its macro-expansion using the configuration pa-

rameters contained in the Colif system description. The resulted macro-expended code becomes the final generated code (SystemC or VHDL) of the module adapters (HW wrapper). In addition, the new resulting Colif system description contains only RTL components. Even the virtual modules that were previously only "marked" as CPUs, now they contain a complete local RTL architecture (CPU SubSystem).

5.2.5 Describing the CosimX tool used for mixed level architecture co-simulation

It does not matter if the architecture is described at a high-level (no low-level details) or at RTL level (with low-level details), the architecture can be executed and validated using the CosimX tool.

The tool is used to validate the system at different levels of abstractions. It takes the Colif representation of the architecture and builds an executable SystemC model. For example, the mixed level interconnections are replaced by the CosimX with SystemC components (from library) which "emulates" their behavior. The CPU cores are replaced by processor simulators: ISS (Instruction Set Simulator) plus simulation adapter (BFM). It is obvious that, the co-simulation speed depends on the abstraction level used by the architecture that has to be validated.

5.3 Implementing a complete MP-SoC architecture for MPEG4 video encoder using ROSES tool

5.3.1 Describing the Executable Model with Explicit Network used for ROSES flow to build the MPEG4 encoder on a complete new MP-SoC architecture

In order to implement an MPEG4 video encoder on a complete new MP-SoC architecture using ARM processors, there are three elements which have to be implemented: the MPEG4 tasks, the Operating System, and the RTL architecture of the MP-SoC. This subchapter will describe the model used as input for the ROSES flow, in order to obtain all of these three elements.

The model used as input for the ROSES flow, is called Executable Model with Explicit Network. This model is obtained from expanding the *Flexible Algorithm/Architecture Model for MPEG4* using the algorithm/architecture configurations determined from the architecture exploration. This process is similar with the ones used to obtain the *Combined Algorithm/Architecture Executable Model*. The only difference is that the Module encapsulating the MPI-SystemC HLPPM is replaced by a Module containing an actual Communication Network implementation.

This Communication Network is called DMS (Distributed Memory Server) [40]. The DMS is a highly configurable, flexible and scalable communication network. As communication APIs, the DMS uses MPI primitives, like the ones used during the algorithm/architecture exploration phase. This is why there is absolutely no need to change the code of the applications, which are using MPI primitives to communicate between them. If during the high-level exploration, the MPI primitives were managed by the MPI-SystemC HLPPM, during the architecture implementation the DMS will manage them. The difference is that what was first a High-Level Parallel Programming Model, now is an actual RTL communication network.

The DMS is also configured according to the selected communication concurrency and topology. For instance, in our case we have configured the DMS to use a P2P communication topology. The connection graph was configured accordingly with the data flow used for the MPEG4 encoder. Figure 74 presents the connection graph used inside the DMS, in the case of MPEG4 video encoder, QCIF resolution, with 4 MainDivXs, 1 VLC, and of course, Splitter and Combiner.

As described in [40], the DMS uses an MSAP (Memory Service Access Point) connected to each processing unit (Module containing an MPEG4 task). Each MSAP is a highly customizable data transfer engine that can be seen as a super DMA capable to transfer multiple local and external requests. All these MSAPs are in charge of managing the message requests from the corresponding processing unit, or from other MSAPs. The interconnection graph between these processing units is done by interconnecting the MSAPs between them.

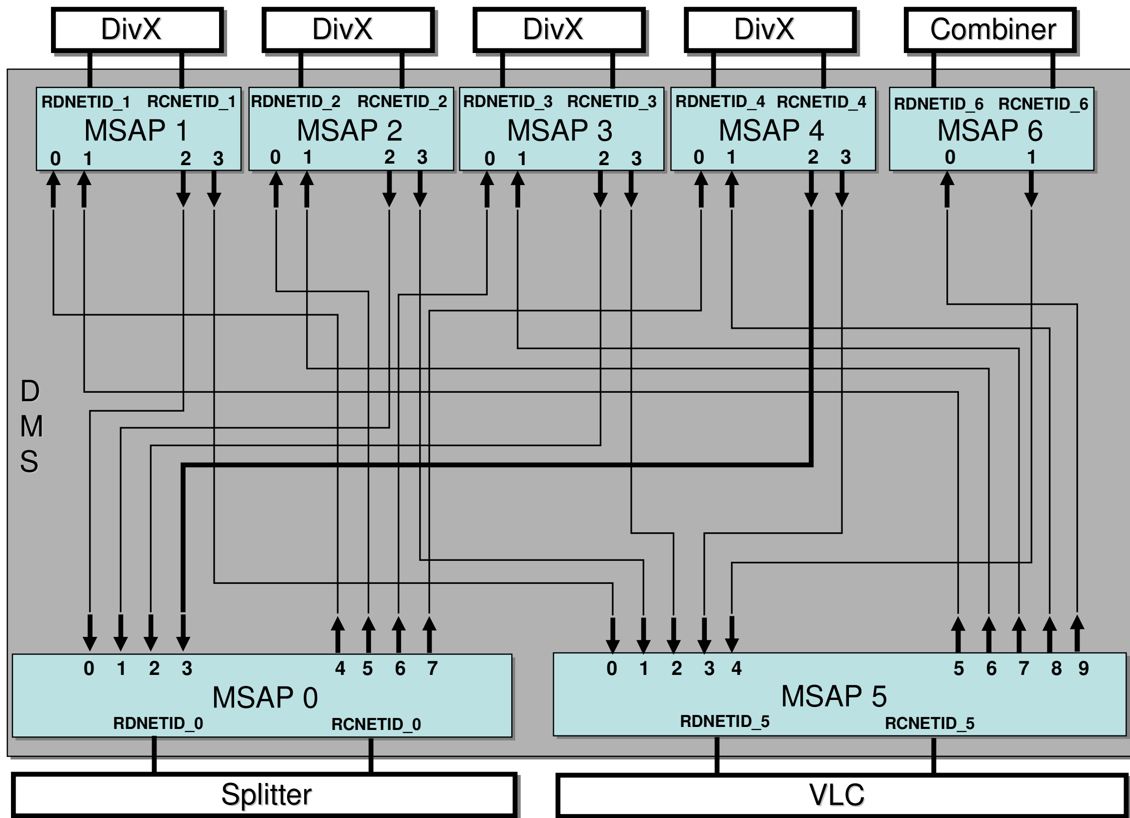


Figure 74. DMS internal architecture for MPEG4 video encoder QCIF,4 MainDivX,1 VLC

For the example presented in Figure 74, the 4th DivX is connected to the Splitter and the VLC. Therefore, the ports 0 and 2 of the MSAP4 (connected to this DivX) are connected to the ports 3 and 7 of the MSAP0 (connected to the Splitter). In addition, the ports 1 and 3 of MSAP4 (connected to this DivX) are connected to the ports 3 and 8 of the MSAP5 (connected to the VLC). There are always two connections between two MSAPs, in order to achieve communications in both directions for data and control. However, these ports and their ID are “invisible” for the outside world, and the application programmer does not need to know them.

The DMS is capable of sending data (commanded by the MPI primitives) for different message sizes, data widths and protocols. Even latencies can be adjusted, by configuring the MSAPs scheduling policy (i.e. 3 cycles are required for communication instantiation, and extra 3 cycles if acknowledgement is desired).

All these features of the DMS enabled the replacement of the MPI-SystemC HLPPM. Figure 75 presents an example of generated Executable Model for MPEG4 video encoder, with DMS, 4 MainDivXs, 2 VLCs, Splitter and Combiner.

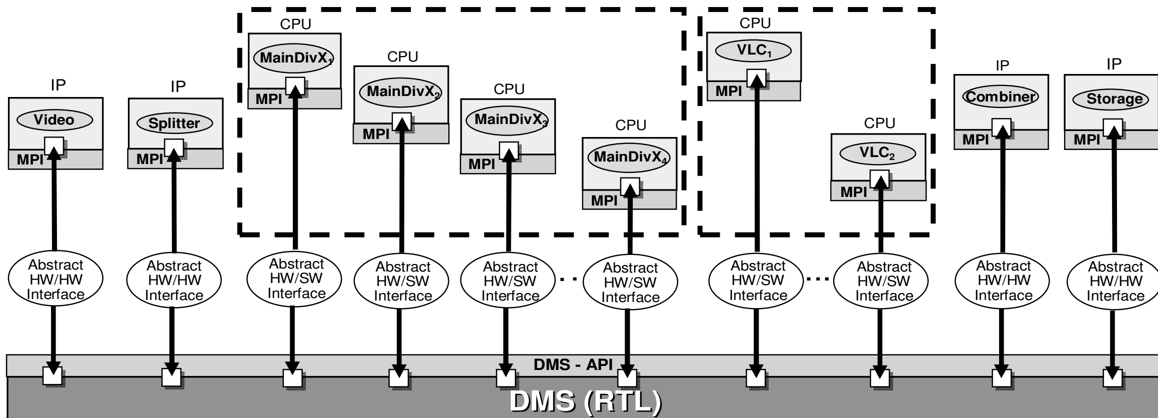


Figure 75. Executable Model for MPEG4 with DMS, 4 MainDivXs, 2 VLCs

In these models, the MPEG4 tasks contain already the final customized code. The last step that has to be done is to refine the Abstract HW/SW Interfaces by replacing them with Operating Systems, CPU Local Architectures and HW Wrappers. To achieve this, we will use the ROSES flow starting from the generated *Executable Model with Explicit Network*.

5.3.2 Representing the flow used for the implementation of a complete MP-SoC architecture for MPEG4 video encoder using the ROSES tool

The ROSES flow uses as input the generated *Executable Model with Explicit Network*. In this model, the MPEG4 tasks are ready and customized. The ROSES flow will build the file for the Operating Systems, and the RTL architecture (Local CPU architectures, and HW Wrappers). The flow used to achieve this is presented in Figure 76.

The *Executable Model with Explicit Network* is obtained by expanding *the Flexible Algorithm/Architecture Model for MPEG4* with the algorithm/architecture configurations found during the High-Level Algorithm/Architecture Exploration. The module containing the MPI-SystemC is replaced by a Module containing a DMS communication network.

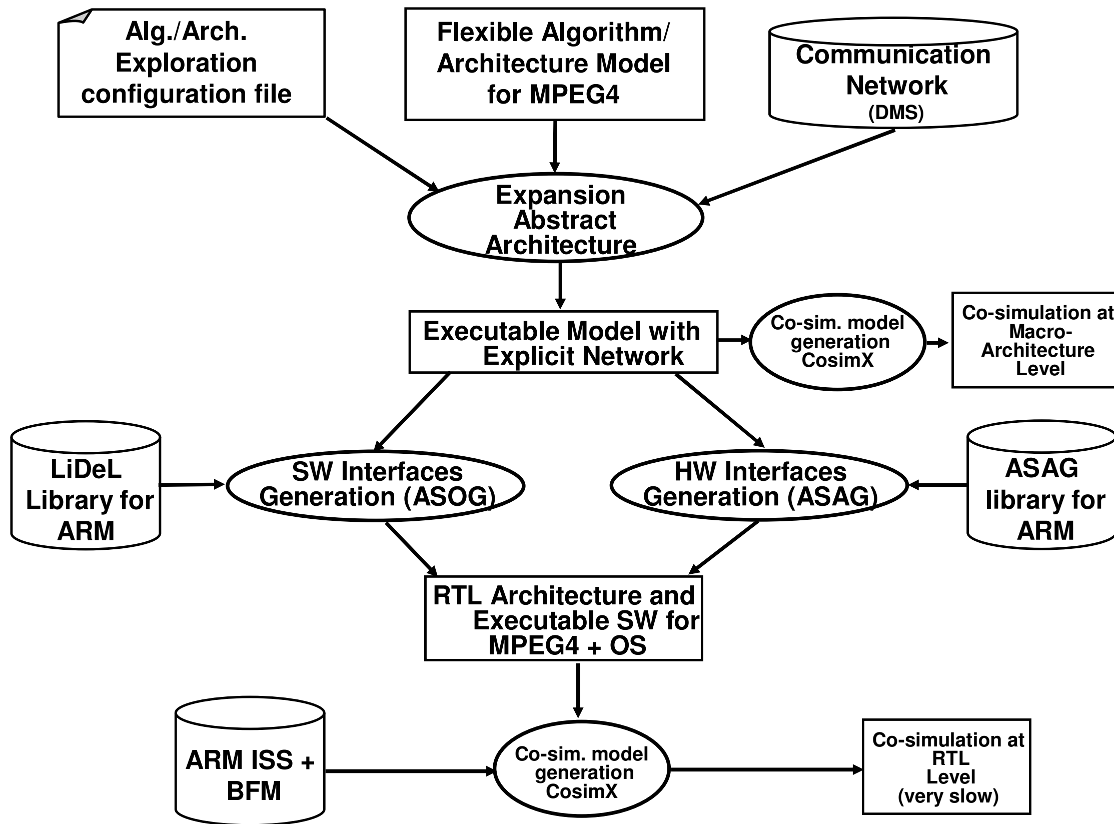


Figure 76. Detailed flow used to implement the MPEG4 on a new MP-SoC architecture

The resulting model can be executed using the CosimX tool in order to do some early architecture validations, tasks debugging, testing the synchronization between the tasks and the DMS, debugging the DMS, etc.

The next step is to generate the OS of the application, using the ASOG tool and a LiDeL library containing the general OS services for the ARM processors.

The RTL architecture is obtained by replacing the CPU Modules containing the MPEG4 tasks, with an actual RTL architecture for the CPU Subsystems of each targeted ARM processors. To be able to interconnect each CPU SubSystem with the DMS, a set of HW Wrappers is generated.

After these steps, the result is a complete synthesizable RTL architecture for the MPEG4 video encoder using ARM processors, along with executable SW for MPEG4 tasks and Operating System. This RTL architecture can also be co-simulated using the CosimX tool. During this co-simulation, there are many low-level details that are simulated

with cycle-accurate precision, the simulation time is slow in this case (approximately 25 hours for co-simulating the MPEG4 video encoder, for 25 frames of QCIF movie, using 5 ARM7 processors for 4 MainDivX and 1 VLC). Instead of the actual CPU core, an ISS (Instruction Set Simulator) is used connected to the RTL architecture through a BFM (Bus Functional Model) [43]required to link the RTL architecture simulation environment with the ISS environment.

5.3.3 Libraries used for the implementation of a complete MP-SoC architecture for MPEG4 video encoder

In order to build the *Executable Model with Explicit Network*, generate the Operating System, the CPU Subsystems and HW Wrappers, and build the model that will be co-simulated at RTL level, multiple libraries are used:

a) Communication network: as explicit communication network was used a DMS communication network between the processors. This model is completely described in SystemC, as a macro-code. This macro-code can be expanded and configured with the required parameters, in order to obtain the currently needed configured DMS module. Its usage was presented previously in this document.

b) LiDeL Library for ARM: this library is used by the ASOG tool to build the application specific OS for the ARM processors. It contains a generic macro-code for the boot of the ARM processors, application loader, scheduling, MPI behavior, context switches (a simple one between a task and the sleep task which executes the idle status of the processor) and interrupt programming. Depending on the configuration of the current Executable Model with Explicit Network, the ASOG tool will build the files of a customized OS for each processor, starting from these macro-codes.

c) ASAG library for ARM: this library contains the Colif architecture for the ARM processors subsystem, which is dedicated to the MPEG4 application. In addition, this library contains the synthesizable files describing the behavior of each component of this processor subsystem (i.e. Memory Controller, Address Decoder, etc). Figure 77 presents the CPU SubSystem used for the MainDivX3 task.

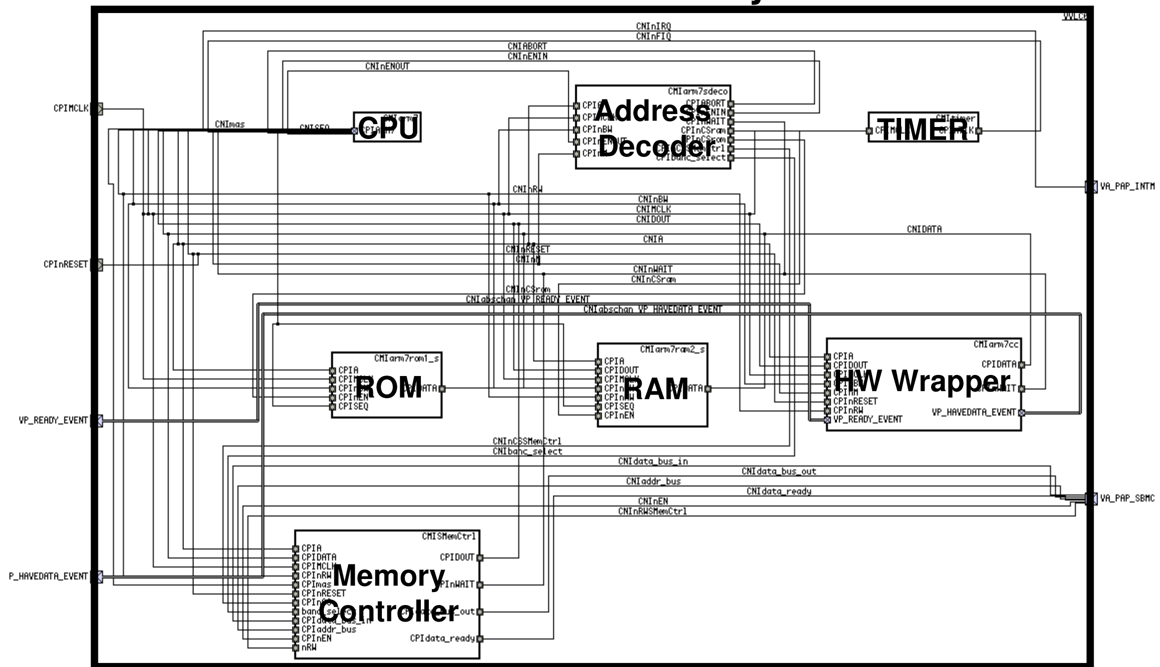


Figure 77. CPU SubSystem RTL architecture for the processor of MainDivX3 task

It can be noticed that the CPU module is in fact the core of the ARM processor. The Timer module provides the clock of the processor. The ROM and RAM modules are the local memory of the processor. The Address Decoder and Memory controller module provides the arbitration between the CPU and the memory modules. The HW wrapper is required to interconnect the entire CPU SubSystem to the Communication Network (DMS). All these modules are fully synthesizable, and their behavior is described in SystemC.

d) ARM ISS and BFM: in order to co-simulate the core of the ARM processors, the CPU Module from Figure 77 is replaced for the co-simulation by an ARM ISS and a BFM.

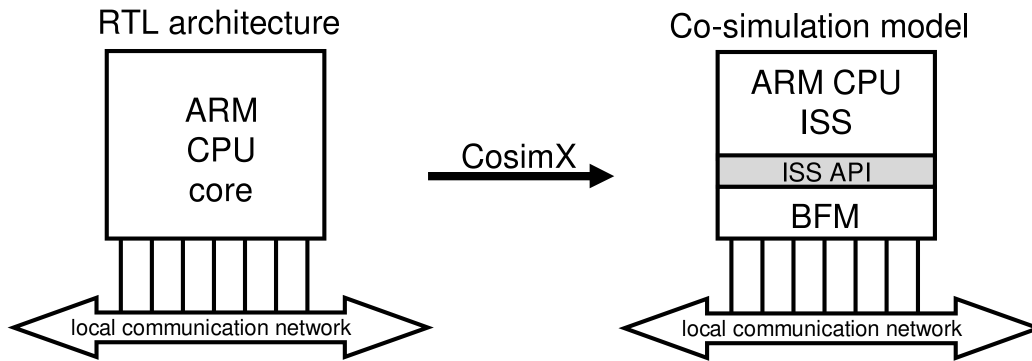


Figure 78. Replacing the ARM core by the CosimX with an ARM-ISS and BFM

The BFM is a SystemC module, which "translates" the signals of the ports connected to the local communication network into APIs required to control the ISS environment. This ISS environment is in fact a tool (i.e. armsd [34], Vtune [35]) which simulates a cycle accurate processor behavior. Every time the processor is changed, the ISS and BFM have to be changed with ones specific for this new processor.

5.3.4 Architecture implementation results for the MPEG4 video encoder on MP-SoC

We have implemented multiple configurations of MPEG4 video encoder on MP-SoC. The configurations used were those provided after the High-Level Algorithm/Architecture Exploration for the MPEG4 video encoder for QCIF and CIF resolutions, using ARM7 respectively ARM946E-S processors.

Figure 79 presents the diagram of the resulting RTL architecture for the MPEG4 video encoder for QCIF video using 5 ARM7 processors (4 MainDivXs + 1 VLC). It can be noticed that, for all the CPUs, the ROSES flow generated the SW Wrappers (OS), the CPU-SubSystems, and for all the Modules were generated the HW Wrappers to interconnect them to the ports of the DMS communication network.

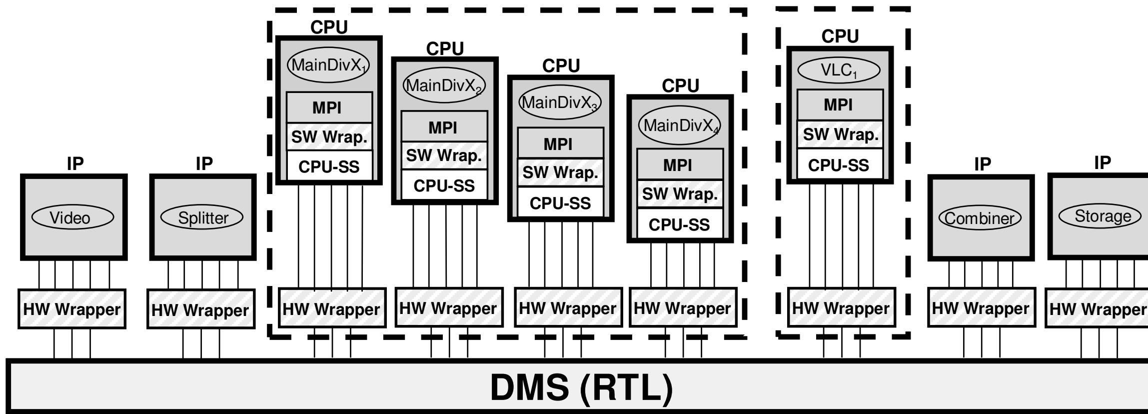


Figure 79. RTL architecture for MPEG4, QCIF, ARM7, 4 MainDivX, 1 VLC

Figure 80 presents the Colif system specifications for the obtained RTL architecture, displayed using the CView tool. The CView tool was developed at TIMA-SLS Group, and it displays graphically and hierarchically the architecture described in the XML file containing the Colif system specifications.

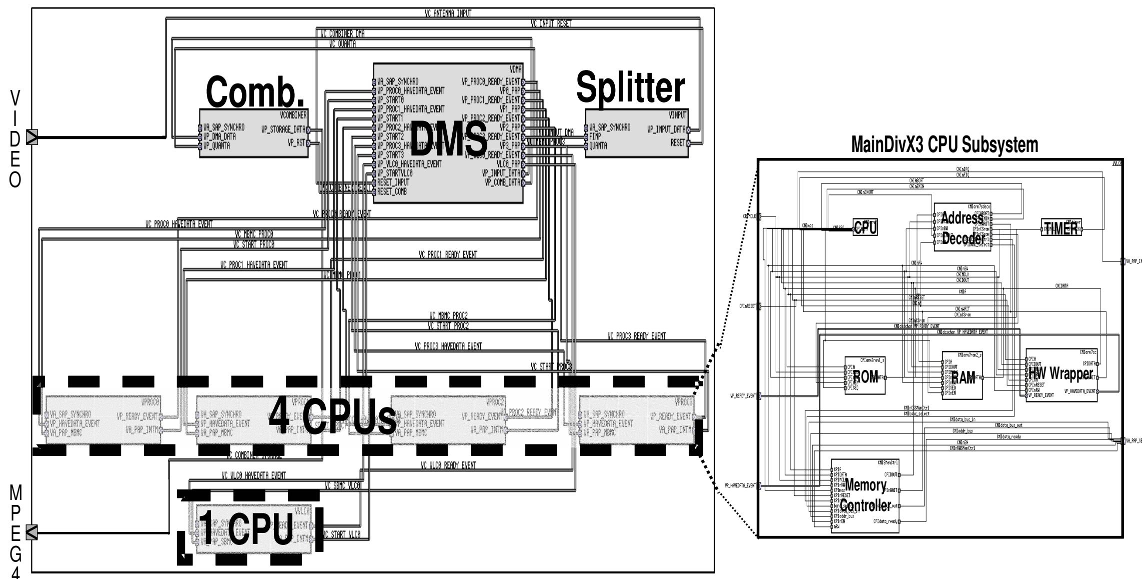


Figure 80. Colif representation of RTL Arch. for QCIF, ARM7, 4 MainDivX, 1 VLC

There are 5 CPUs in the obtained RTL architecture: 4 CPUs for the 4 MainDivX tasks, and 1 CPU for the VLC task. For each of these CPUs, an internal architecture was generated, containing the ARM7 CPU core, Address Decoder, Memory Controller, ROM, RAM and HW Wrapper. The *Splitter*, *Combiner* and *DMS* are HW modules. The modules con-

taining the Video and Storage tasks were not included in the architecture, because those are just benchmark modules required during the simulation/validations.

After the RTL synthesis, this architecture required additional 54.7K gates in addition to the 5 CPUs. The size of the entire embedded memory required for this case, is approximately 594Kbytes (ROM+RAM).

Figure 81 presents the Colif system specifications for the RTL architecture obtained for the MPEG4 video encoder for CIF, using 10 ARM9 processors (8 processors for 8 MainDivXs tasks and 2 processors for 2 VLC tasks). It can be easily noticed that the DMS has now much more ports than the previous example. The Splitter, DMS and Combiner are IPs in this case also.

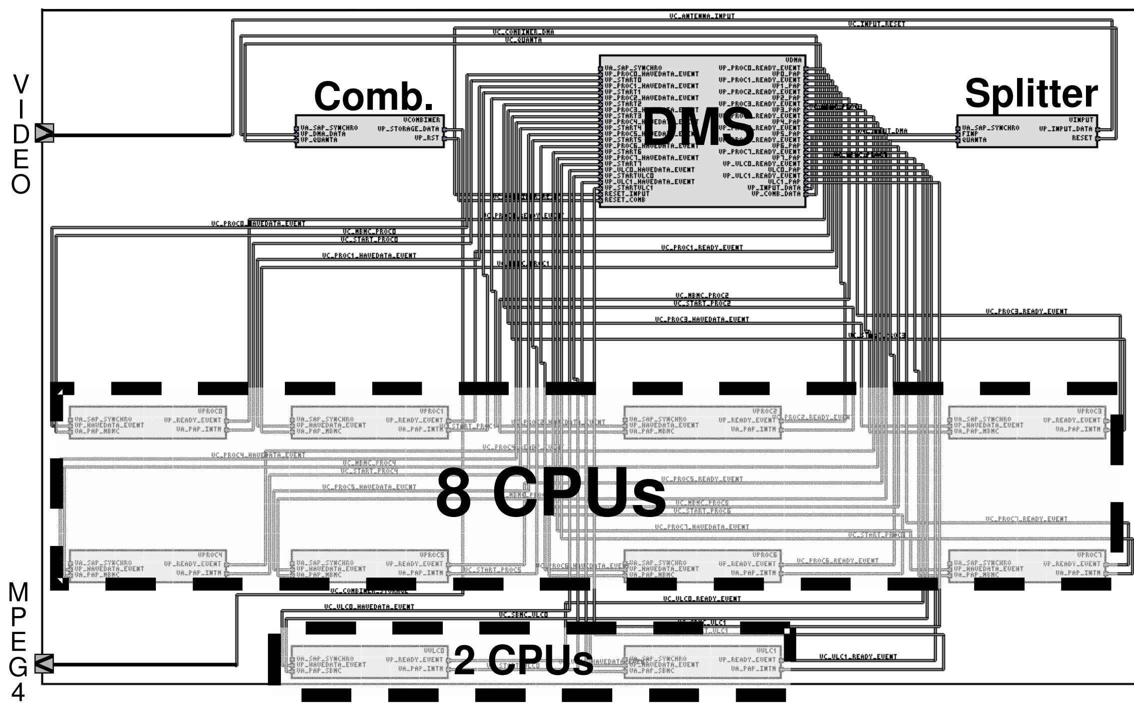


Figure 81. Colif representation of RTL Arch. for CIF, ARM9, 8 MainDivX, 2 VLC

After the RTL synthesis, the architecture presented in Figure 81 required 100.4K gates in addition to the 10 CPUs, and 2.418Kbytes embedded memory (ROM+RAM)

5.3.5 Performance measurements for the final MPEG4 video encoder on MP-SoC

All the obtained RTL architectures can be co-simulated at this low-level using the Co-simX tool. The simulation is extremely slow, but it is precise.

We have co-simulated the RTL architecture for MPEG4 video encoder for QCIF, using 2 ARM7 processors (1 MainDivX and 1 VLC) running at 60MHz. The co-simulation consisted of encoding 25 frames of QCIF movie, movie that was used also during the High-Level Algorithm/Architecture Exploration.

The co-simulation for those 25 frames took approximately 25 hours. Hence, we were able to check the precision factor of the performances estimated during the High-Level Algorithm/Architecture Exploration, by comparing them with the ones measured at RTL level.

To compress 1 frame at QCIF resolution, using 2 ARM7 CPUs (1 MainDivX + 1 VLC) running at 60 MHz, the high-level estimations predicted that 115.02 ms are required (in Figure 61, 6.82 million cycles is equivalent to 115.02 ms). The performances measured for the obtained RTL architecture proved that 123.53 ms were required to compress 1 frame. Therefore, there was a 7.39% precision error at the high-level estimations (Figure 82).

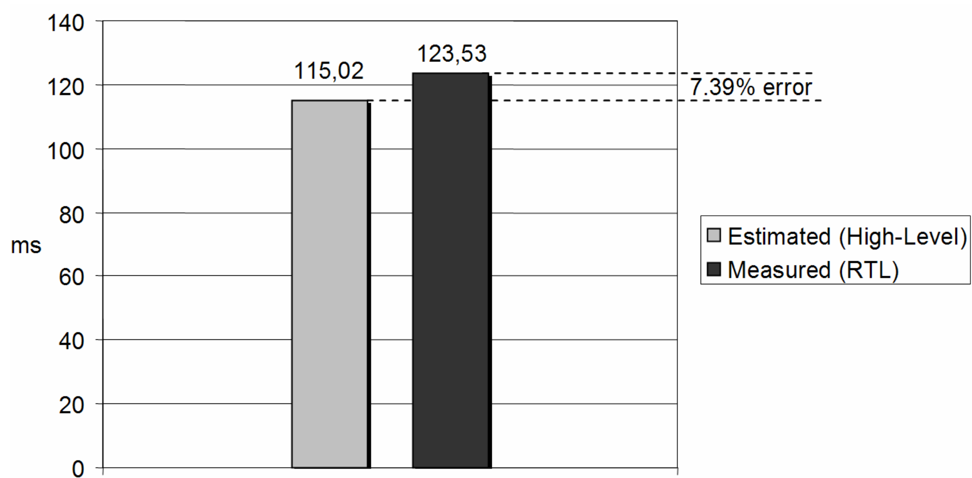


Figure 82. Estimated vs. Measured performances [QCIF, 1 frame, 2 ARM7 CPUs]

The 7.39% precision error comes from the impossibility to capture with our proposed high-level estimations, the performance degradations of the:

- OS (scheduling, service calls latencies induced by the API calls)

- Interconnect between the CPU buses and communication infrastructure (the conflicts for local bus grant between the CPUs and the Network Interfaces).

- HW Wrappers

Six months were required to build the ROSES libraries (by 5 persons) which are required to generate the RTL architecture for the current MPEG4 video encoder application. Once all the libraries are ready, approximately 5 minutes are required to generate the *Executable Model with Explicit Network*. Obtaining the final RTL architecture by the ROSES flow is completely automatic (all the tools are automated) and it requires approximately 15 minutes (caused mainly by compilations). 25 hours are required to co-simulate at RTL level the encoding of 25 frames. Any error found in the implementation leads to the need of correcting it, regenerating the *Executable Model with Explicit Network*, regenerating the RTL architecture, and executing again the 25 hours of co-simulation. Comparing this effort with the one required by the high-level architecture exploration, the latter requires a much smaller effort and it is logical to consider that the 7.39% estimation error is more then affordable. Most of all, during the high-level algorithm/architecture exploration, many errors can be found and corrected quickly. Thus, the number of possible errors that may appear at RTL level is drastically reduced.

5.4 Implementing the MPEG4 video encoder on an existing Quadric-Processors platform using ROSES tool

5.4.1 Describing the targeted Quadric-Processors platform

The second implementation for MPEG4 video encoder consists of integrating the MPEG4 video encoder on an already existing architecture. This approach consists of map-

ping the MPEG4 video encoder application on the platform along with an application specific OS.

The Quadric-Processors platform (Figure 83) is formed of 4 processor sub-systems. All of these processors have the same type of RISC core: Sparc V8.

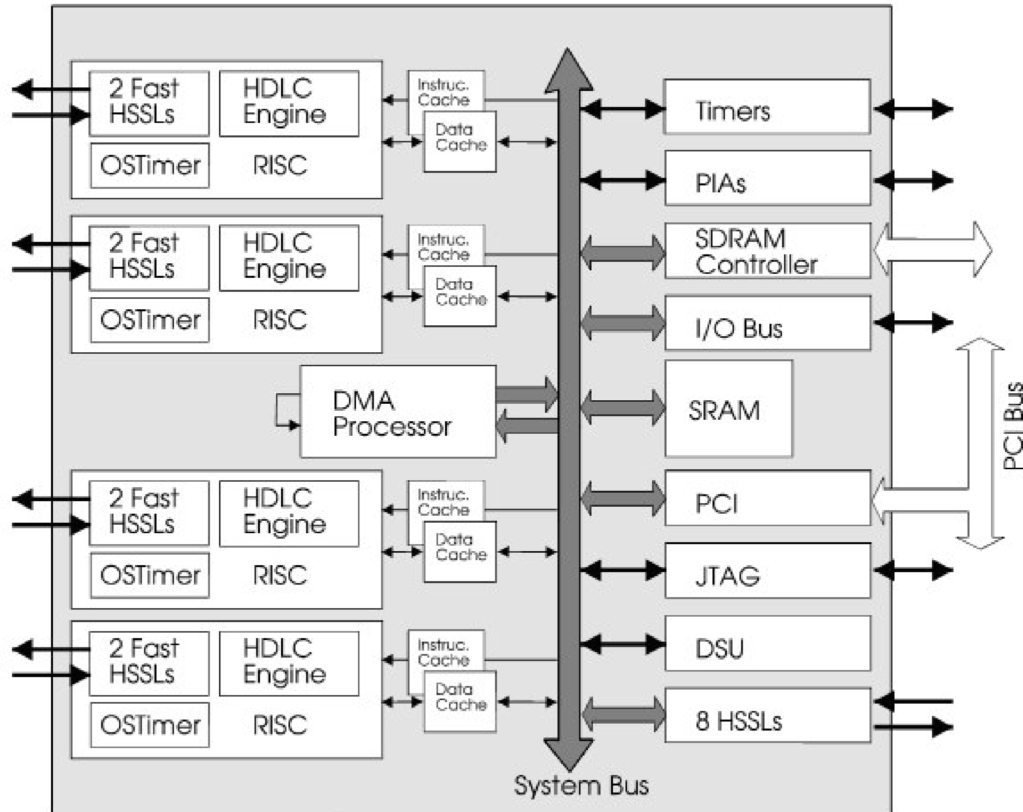


Figure 83. System architecture of the Quadric-Processors Platform

The system uses a centralized memory architecture, using an external 32 bits shared SDRAM memory of 32 Mbytes (extensible at 64 Mbytes). For memory access, 2 FIFOs are used for storing the request "rockets": 32 words FIFO for store, 6 words FIFO for reads. In addition, as internal memory, a fast 32 bits SRAM memory of 32 Kbytes is used. The memory access requests are fetched between the processors and the memories by a global DMA processor.

The system uses a unified memory address for all its memories, caches, I/O, etc. The entire communication is done using a bus-centric communication topology. This bus is similar with the AMBA bus, with some architecture specific modifications meant to in-

crease the system performances (i.e. uses separate channels for read & write into the memories). The entire system works at 90MHz clock frequency.

The local architecture of each processor is presented in Figure 84.

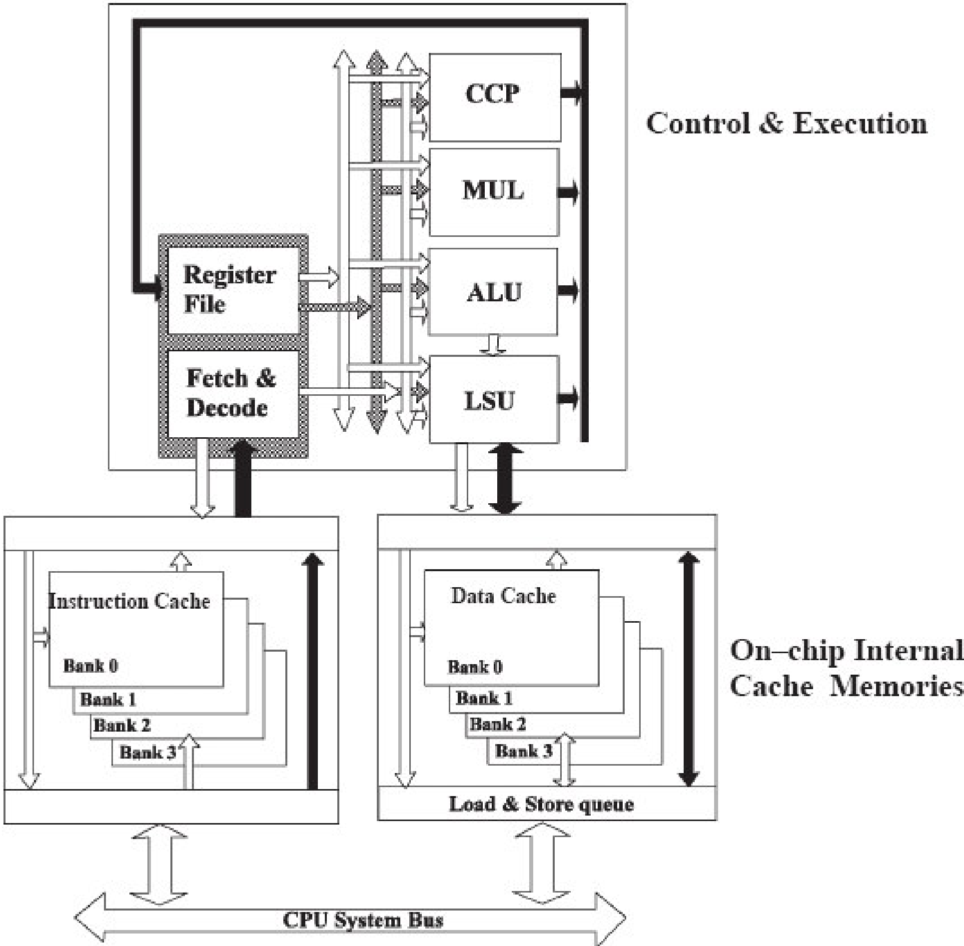


Figure 84. Diagram of the CPU architecture used on the Quadric-Processors Platform

The processors do not dispose of any local memory, except for two 16kbytes caches: one for instruction, one for data. Each cache is four way associative with 32words/cache line. Between the data cache and the system bus are used 2 FIFOs with a depth of 32 words of 32 bits each, for storing the load and store requests. The instruction cache uses only one FIFO (for load) that is 16 instructions deep.

As local execution units, the processor uses 4 separate execution units: CCP (Communication CoProcessor) used for instruction set extensions, MUL (Multiplier Unit), LSU

(Load-Store Unit) with the possibility of managing in parallel multiple independent load and store requests, and an ALU (Arithmetical Logic Unit).

Because of these separated execution units, the processor is capable of executing in parallel multiple operations on the condition that there is no interdependency between them. For instance, when the processor requests a data load, it can also execute an operation that does not depend on that data load. In addition, multiple loads and stores can be requested in parallel, and the data coherence is completely managed by the LSU. Moreover, because of this parallelism, it can be said that the store feature is virtually "free" in terms of performance.

5.4.2 Describing the Executable Model with Explicit Network used for ROSES flow to build the MPEG4 application and the Operating Systems for the Quadric-Processors Platform

To integrate the MPEG4 video encoder on the Quadric-Processors Platform, there are two elements that have to be implemented: the MPEG4 tasks, and the Operating System. In order to do this, we use the ROSES flow. This sub-chapter will describe the model used as input for the ROSES flow, in order to obtain the MPEG4 tasks, and the Operating System.

The model used as input for the ROSES flow, is called *Executable Model with Explicit Network*. Like the implementation of the MPEG4 video encoder on a complete new architecture, this model is obtained from expanding the *Flexible Algorithm/Architecture Model for MPEG4* using a set of algorithm/architecture configurations. For these experiments, we have improved the *Flexible Algorithm/Architecture Model for MPEG4* to be able to map multiple tasks into the same Module, depending on the user's preference. However, the current configuration of the Quadric-Processors Platform is restricting the number of processors to maximum 4. As explicit network, we have decided to use a P2P communication between the processors and an abstract global memory. This memory will play the role of storing all the data, which in the Quadric-Processors Platform will be mapped in the SDRAM.

Figure 85 presents an example of *Executable Model with Explicit Network* used as input for the ROSES flow, when using 2 CPUs for 2 MainDivXs, 1 CPU for 1 VLC, and 1

common CPU for Splitter and Combiner. Figure 86 presents another example of Executable Model used as input for the ROSES flow, when using 3 CPUs for 3 MainDivXs, and 1 common CPU for Splitter, Combiner and VLC. Other configurations can be also obtained.

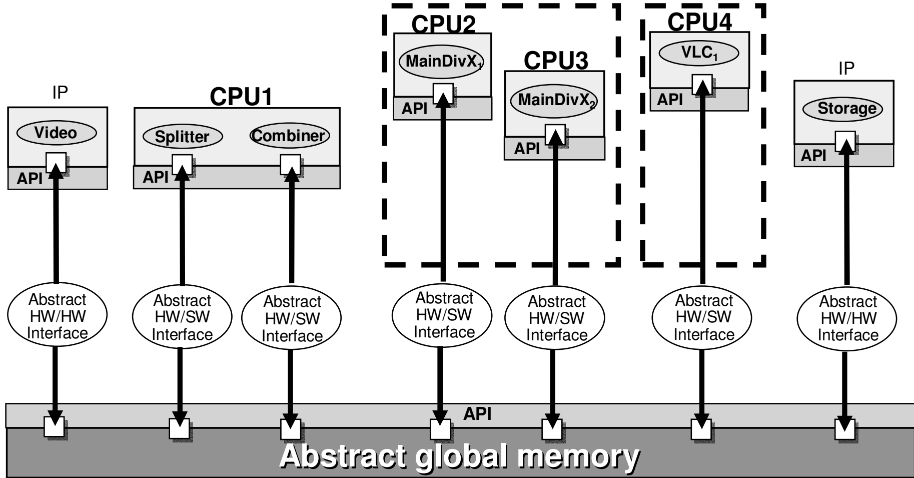


Figure 85. Executable Model for MPEG4 with 2 MainDivXs, 1 VLC, Splitter+Combiner

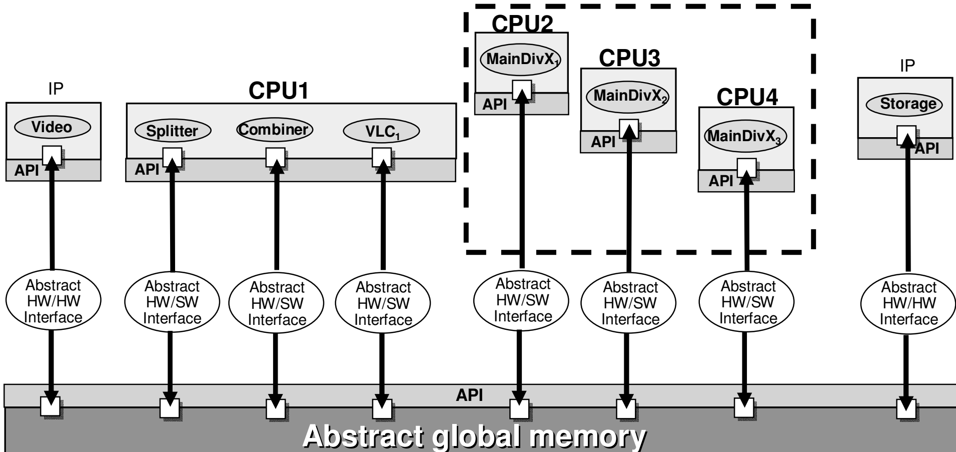


Figure 86. Executable Model for MPEG4 with 3 MainDivXs, Splitter+Combiner+VLC

In these models, the MPEG4 tasks are already ready for being mapped on the platform. However, the last thing that still has to be done is to obtain the Operating System for the CPUs. Therefore, we use the ROSES flow in order to obtain the SW interfaces (OS) for the current configuration of Executable Model with Explicit Network. Only after this step, the entire application can be mapped and executed on the platform.

5.4.3 Representing the flow used for mapping the MPEG4 video encoder on the Quadric-Processors platform using the ROSES tool

The flow uses as input an expanded *Executable Model with Explicit Network*, and obtains in the end the tasks of the MPEG4 application and the specific OS for the CPUs of the platform (Figure 87).

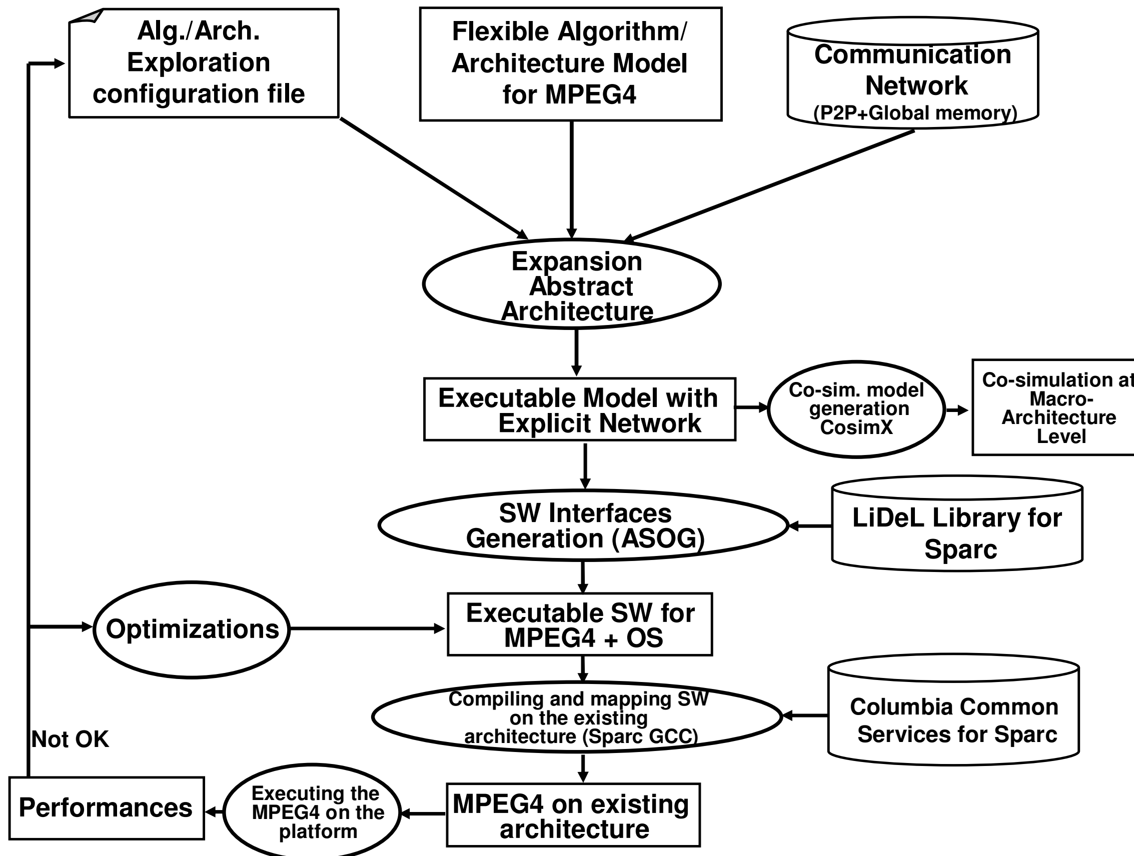


Figure 87. Detailed flow used to map the MPEG4 on the Quadric-Processors Platform

The *Executable Model with Explicit Network* is obtained by expanding the *Flexible Algorithm/Architecture Model for MPEG4* with the desired algorithm/architecture configurations. The module containing the MPI-SystemC HLPPM is replaced by a module containing a P2P+Global memory communication network.

The resulting model can be executed using the CosimX tool, in order to do some code or architecture validations, debugging, testing the good interaction/synchronization between the tasks and the global memory, etc.

The next step is to generate the OS of the application, using the ASOG tool and a LiDeL library containing the general OS services for the Sparc. After this step, the code of all the application's tasks and OS are ready.

In order to map the application on the platform, all the application's codes and a library containing a set of Common Services provided for this platform are cross-compiled using the SPARC-V8 instruction set.

After the compilation, the resulting binary is loaded on the platform, and it is executed. The platform is connected to a Web Camera that provides the input uncompressed movie, and the platform compresses the movie, and stores the MPEG4 bitstream at a specific address into the memory. In parallel, this bitstream is loaded by a PC workstation via the PCI interface, and played by a media player (Xine under Linux). As a result, it is possible to see in real-time the resulting compressed movie (Figure 88).

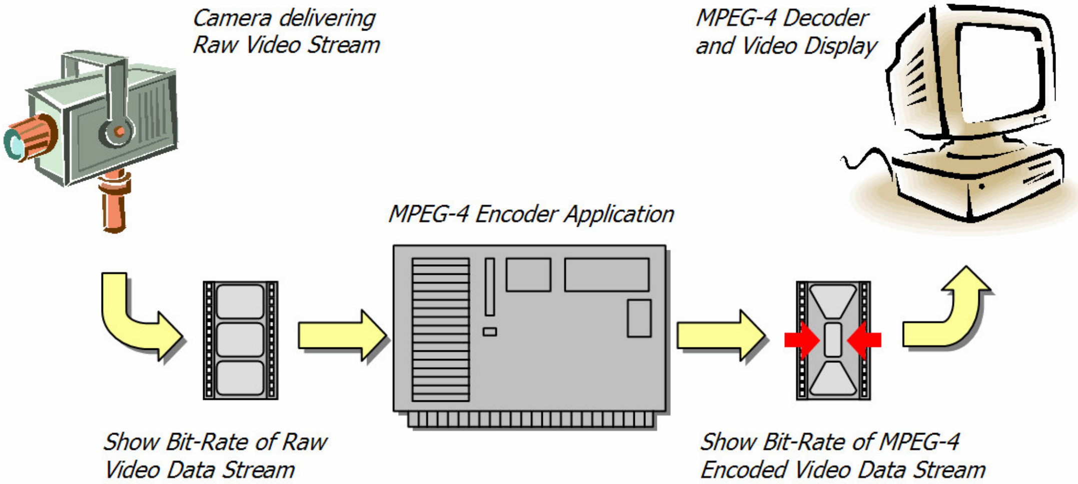


Figure 88. Real-time execution method used for the MPEG4 video encoder on the platform

During the execution, the performances can be determined. In case these performances are not satisfactory, the application requires some modification. These modifications consist either of changing the algorithm/architecture configurations (i.e. changing from using 2 MainDivX to using 3 MainDivX), or optimizing the MPEG4 code to take advantage of some of the platform's specific features (optimizations described later in this document).

5.4.4 Libraries used for mapping the MPEG4 video encoder on the Quadric-Processors platform

To build the *Executable Model with Explicit Network*, to generate the application specific OS, but also to obtain the application's executable binary code which will be mapped on the platform, a set of libraries are used.

a) Communication network: a P2P interconnect was used as communication network between the processors and an abstract global memory. This abstract global memory is described as a SystemC module using generic macro-code. This macro-code can be expanded and configured with the required parameters, in order to obtain the currently needed parameterized global memory module.

   Its behavior is a passive one (it does nothing). Its own purpose is to instantiate all the structures, arrays and data that will finally be mapped on the SDRAM. The number of ports connected to this abstract global memory is identical to the number of modules that will contain data to be stored into the SDRAM. It is important to mention that during the CosimX co-simulation is impossible to restrict the address where the compiler will allocate these data into the workstation's memory. We leave the compiler decide these addresses. The abstract memory will *Send* to the MPEG4 tasks, at the beginning of the simulation, messages containing all the addresses of these structures to the MPEG4 tasks.

b) LiDeL Library for Sparc: this library is used by the ASOG tool, in order to build the application specific OS. It contains generic macro-code for the boot of the Sparc processors, the application loader and reset, scheduling, context switches, API behavior, interrupt programming, timer watch for performance monitoring, etc. Depending on the algorithm/architecture configuration used for the input Executable Model, the ASOG tool will assemble a customized OS starting from these macro-codes.

c) Columbia Common Services for Sparc: this library is provided in the SDK package (Software Development Kit) for the current Quadric-Processors Platform. When compiling the application for the platform, this library has to be included to accom-

plish the compilation. The Common Services library contains some low level OS services, which are completely independent of the application (i.e. processor timer reset).

5.4.5 Implementation results for MPEG4 video encoder on the Quadric-Processors platform

Using this flow, multiple configurations of MPEG4 video encoders were successfully mapped on the Quadric-Processors Platform. Approximately 3 minutes are required to generate automatically the *Executable Model with Explicit Network* and the Colif system description. Approximately 30 seconds are required to obtain the files of the OS, and approximately 10 minutes are required for the final compilation phase needed to obtain the executable MPEG4 binary code for the platform. To conclude, approximately 10-15 minutes are sufficient to obtain a running MPEG4 video encoder on the platform.

Figure 89 presents the encoding performances for different MPEG4 video encoder implementations on the platform, for QCIF and CIF video resolutions. It is important to mention that all these performances are obtained using the initial MPEG4 algorithm, before any optimization.

Resol.	CPU1 tasks	CPU2 tasks	CPU3 tasks	CPU4 tasks	Encoding frame_rate	Bottleneck
QCIF	Splitter+Combiner	MainDivX1	MainDivX2	VLC	15 fps	CPU2,3
	Splitter+Combiner+VLC	MainDivX1	MainDivX2	MainDivX3	18 fps	CPU1
CIF	Splitter+Combiner	MainDivX1	MainDivX2	VLC	5 fps	CPU2,3
	Splitter+Combiner+VLC	MainDivX1	MainDivX2	MainDivX3	7 fps	CPU1

Figure 89. Encoding speeds for the MPEG4 encoder on the Quadric-Processors platform

For instance, in case of QCIF video resolutions, when using 2 CPUs for 2 MainDivX tasks, 1 CPU for VLC, and 1 common CPU for Splitter and Combiner, the platform can encode 15 frames/sec. The bottlenecks are the two MainDivX tasks.

However, when using 3 CPUs for 3 MainDivX tasks, and 1 common CPU for Splitter, Combiner and VLC, the maximum possible encoding frame rate became 18 frames/sec.

Thus, even if the MainDivXs bottleneck was "relaxed" by adding more parallelism, the CPU1 became the new computational bottleneck.

5.4.6 Performance analysis for the obtained MPEG4 video encoder on the Quadric-Processors platform

As can be seen in Figure 89, the Quadric-Processors Platform is not capable of processing the MPEG4 video encoder in real-time for a QCIF video resolution movie at 25 frames/sec. Of course, the current encoding speed of approximately 15 frames/sec can be sufficient in some application domains (i.e. video surveillance). However, in order to target 25 frames/sec encoding speed (i.e. for mobile applications), the current MPEG4 video encoder implementation on the Quadric-Processors Platform requires some optimizations.

The following section presents the approach used to optimize the *MainDivX* tasks. In order to optimize the *MainDivX* tasks, the first thing that has to be done is to analyze the computational bottlenecks inside the MainDivX. This can be done by dynamically profiling the application. After profiling the application, it was easy to find out which of the functions of the MPEG4 tasks are consuming the most computational resources (Figure 90), and the number of calls for each of these functions (Figure 91). The profiling was done for the case when encoding 905 frames of QCIF movie that contained variable motion complexity scenes.

In order to simplify these graphs, only the relevant functions were presented. The rest of the functions contained into the MainDivX tasks which are not presented in these figures, were not significant in terms of computations (all of them represented ~ 1% of the total computation requirements).

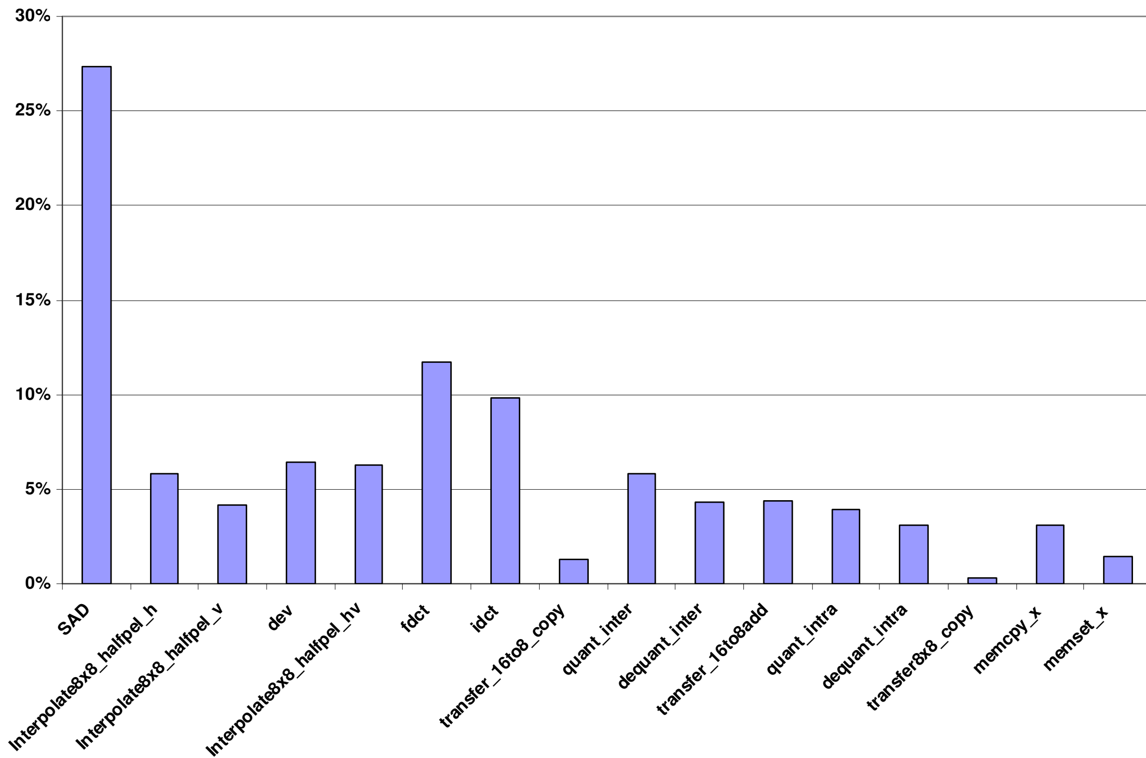


Figure 90. Computation requirements for the functions used by the MPEG4 task

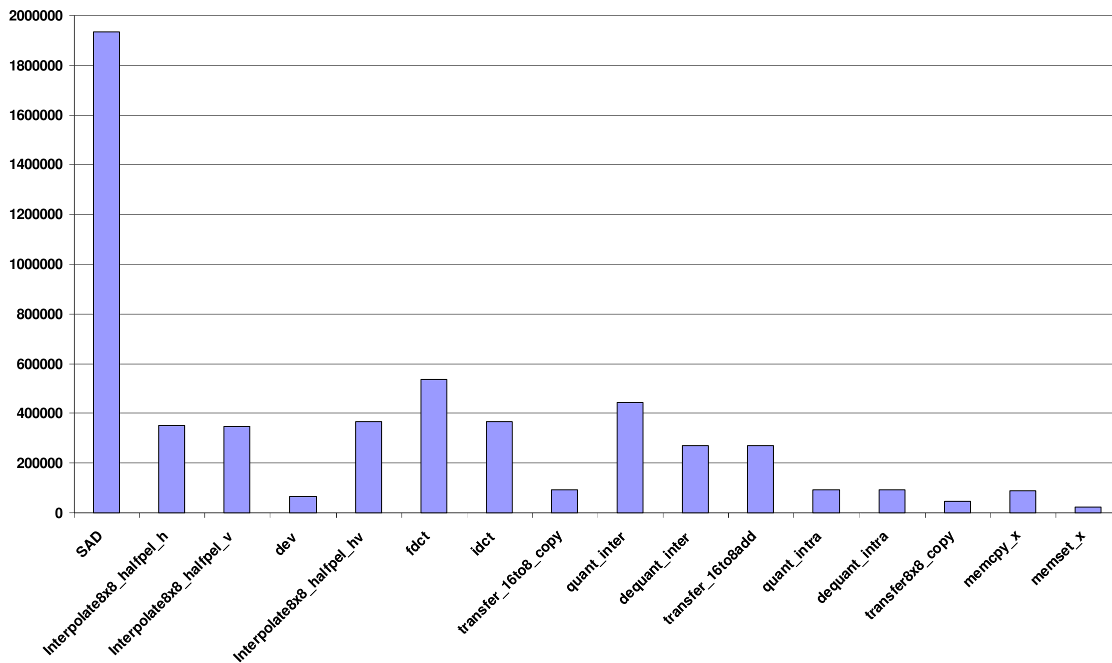


Figure 91. Number of calls for the functions used by the MPEG4 task

5.4.7 Optimizing the MPEG4 algorithm to increase the performances of the MPEG4 video encoder on the Quadric-Processors platform

The best candidate functions for the optimization can be found through the analysis of profiling graphs. Taking into account that the original code was already optimized for computations (from the implementations using ARM CPUs), the main optimizations remaining to be done were related to memory access.

The memory access optimizations can be done by reducing the number of data cache misses. In case some data cache misses cannot be eliminated, to avoid stalling the processors when these misses occur. This last aspect can be achieved by exploiting the parallel LSU feature of the processors from this platform, feature described earlier in this document.

To achieve the memory optimizations, it had to separate the instructions related to memory access (load and store) by the instructions related to register based computations. When the processors are accessing a non-cached data from the global memory and immediately after that, it is using that data, the processor will stall (wait). The stall time depends on the time required to load the data from the global memory (along with its “neighbor” data, forming a cache line), the time to transport the data through the communication network and the time to load the data in the cache and in the register. In case of this architecture, this latency to the global memory access is about 50 cycles.

The idea behind this optimization is to load the data from the global memory, but to use it only later. This way, the memory access becomes in fact a “touch” to the global memory, to be sure that we are forcing that data to arrive in the cache. More exactly, we are using the fact that a processor contains parallel execution units for the LSU, MUL, ALU, CCP, and 6 independent parallel Load & Store capability in the LSU. Thus, while the data is loaded in the cache from the global memory, the processor will execute other instructions using other data, which preferably are already in the cache (Figure 92). This process is called data preloading. Using this strategy, the number of cycles on which the processor is stalled is significantly decreased. However, considering the complexity and unpredictability of this application, it is practically impossible to eliminate all the stalls.

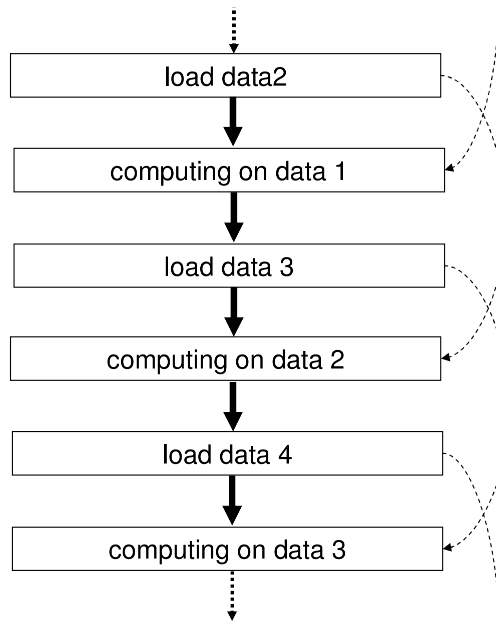


Figure 92. Principle of the memory access optimization used for the MPEG4 video encoder

In Figure 92, a 2-stage preload is used. This means that only one data is preloaded at a time, and it will be used only after two application steps (instructions). However, it is possible to have multiple stages of preloads. For example it is possible to preload the *data1*, *data2*, *data3*, and only after that to start to compute on *data1*.

The first function that was optimized using this preload strategy was the *SAD* function, which represents the performance bottleneck of the *MotionEstimation* function. In the *Original code* (Figure 93, left), to compute the value of *T1*, the processor had to wait all the time the values from address *ptr_cur* and *ptr_ref* to be available (the *ptr_cur* and *ptr_ref* represent the addresses of the two MacroBlocks which have to be compared). This meant that, if those data were not already in the cache memory, the processor had to wait for them to be loaded from the global memory. Therefore, the processor got stalled (wait) while loading those data.

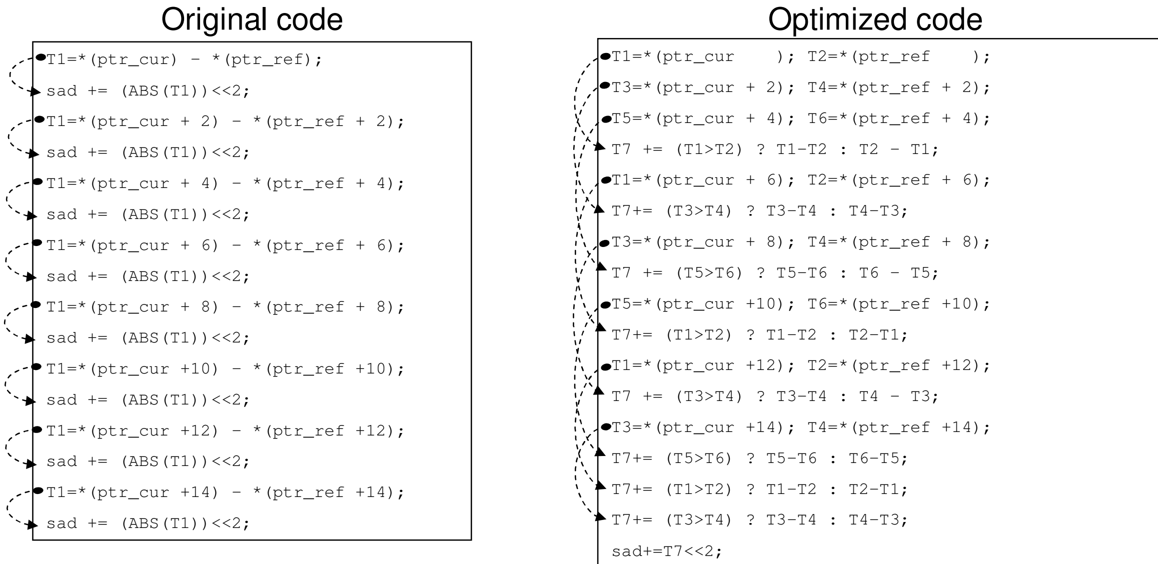


Figure 93. Main part of SAD function, before and after optimization

In the *Optimized code* (Figure 93, right) we adapted the code so that the processor loaded in the variable *T1* the data from the address *ptr_cur*, and in a second variable *T2*, the value from the address *ptr_ref*. The most important thing is that we did not use those variables immediately, giving time for those data to be loaded from the global memory. Instead, we started to execute other loads from the global memory, into other variables (which will be used later). Only after a while, we used those previously loaded *T1* and *T2*, when computing the *T7*. As a result, while the data was loading, we have given to the processor other things to do.

In this case, we have used 6-stage preload. Currently, finding the best stage number of preloads is done using brute-force method, by checking with different stages, and finding out which one provides the best performances. However, in case of this SAD function, just by doing these optimizations we have gained almost 350.000 cycles, from the required time to encode one P frame. This leads to an increase of possible frame rate with plus ~2.5 frames/sec.

Another problem found during the optimizations using preloads was about the preloading strategy used in case of loops. As presented in Figure 94 (part of the FDCT function), the idea is to preload some of the data before entering into the loop, and to preload the same data before the end of the loop (to be available for the next loop step).

```
//preloading i1 & i6 for the loop
i1=(short) (*(img + 1)); i6=(short) (*(img + 6));
for (short int i=7;i>=0;i--)
{
  //i1 & i6 are already preloaded before the loop or from a previous step of loop
  //i1=(short) (*(img + 1)); i6=(short) (*(img + 6));
  i0=(short) (*(img + 0)); i7=(short) (*(img + 7));
  tmp1=i1+i6; tmp6=i1-i6;
  i1=(short) (*(img + 3)); i6=(short) (*(img + 4));
  tmp0=i0+i7; tmp7=i0-i7;
  i0=(short) (*(img + 2)); i7=(short) (*(img + 5));
  tmp3=i1+i6;tmp4=i1-i6;
  //preloading the i1 & i6 for the next loop step
  i1=(short) (*(img + 1 + stride)); i6=(short) (*(img + 6 + stride));
  tmp2=i0+i7;tmp5=i0-i7;
  img += stride;
  ………… //some other computations, irrelevant for this example
}
```

Figure 94. Preloading optimization for loops

For optimizations, the *i1* and *i6* where preloaded before the loop, even if they are the first values to be used in the loop. In addition, the *i1* and *i6* are also preloaded in the end of the loop, to be available for the next step of the loop. Just by applying this method, we have gained approximately 15.000 cycles required to encode one frame.

5.4.8 Performance analysis for the obtained optimized MPEG4 video encoder on the Quadric-Processors platform

After optimizing the functions contained in the MainDivX tasks, using the preloads strategy, the performance gain is approximately 1.231.000 cycles. In terms of frame rate encoding capability of the MPEG4 task, this is equivalent to an extra 9 frames/sec. Figure 95 shows the graph with the amount of cycles gained for each function, after applying the preloads optimizations. It can be seen that the first benchmarks were useful in finding the actual bottlenecks.

One can notice from Figure 95, the reason why we build our own functions for the memory management, like **memcpy_x**, instead of using the standard C/C++ **memcpy**. This way, we where able to “control” how this memory copy behaves in terms of memory accesses, which lead to gain of almost 175.000 cycles required to encode 1 frame.

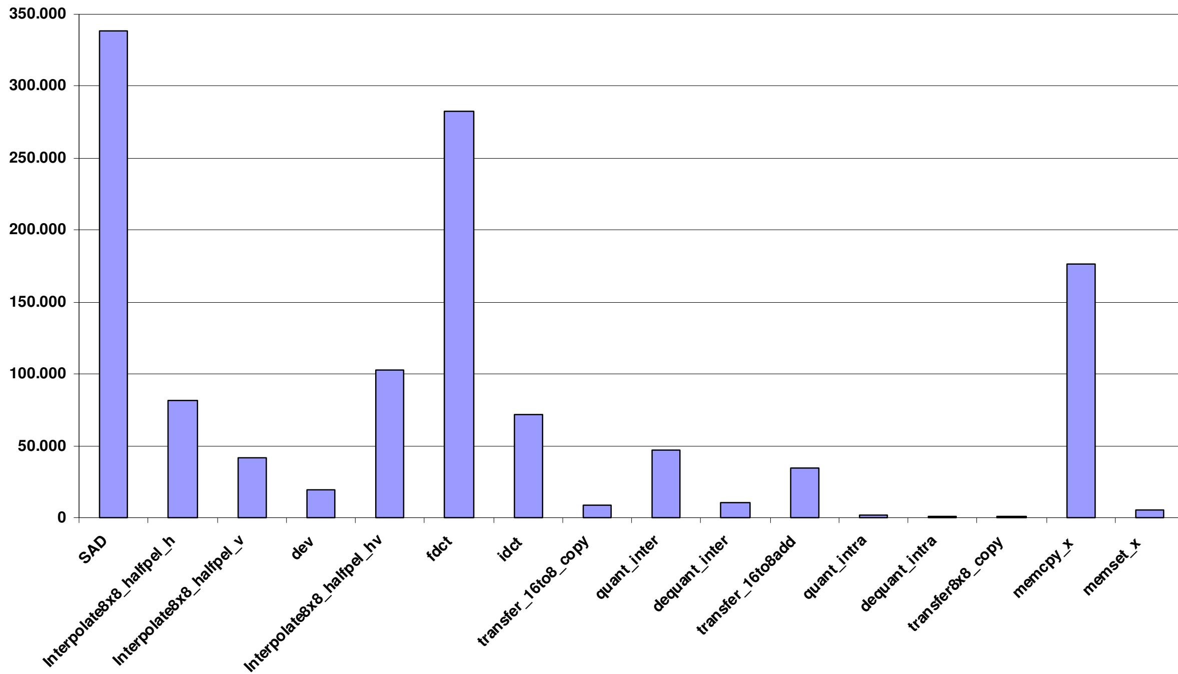


Figure 95. Cycles gained after the optimization of the functions used by the MPEG4 task.

From the total cycles gained thanks to the preloads optimizations, almost 28% were gained by the optimization of the **SAD** function, 23% because of the **FDCT**, and surprisingly 14% because of the **memcpy_x**. This means that the standard C/C++ **memcpy** instruction is inappropriate for the used Quadric-Processors Platform. Figure 96 shows the percentage of each function optimization, from the total gained cycles.

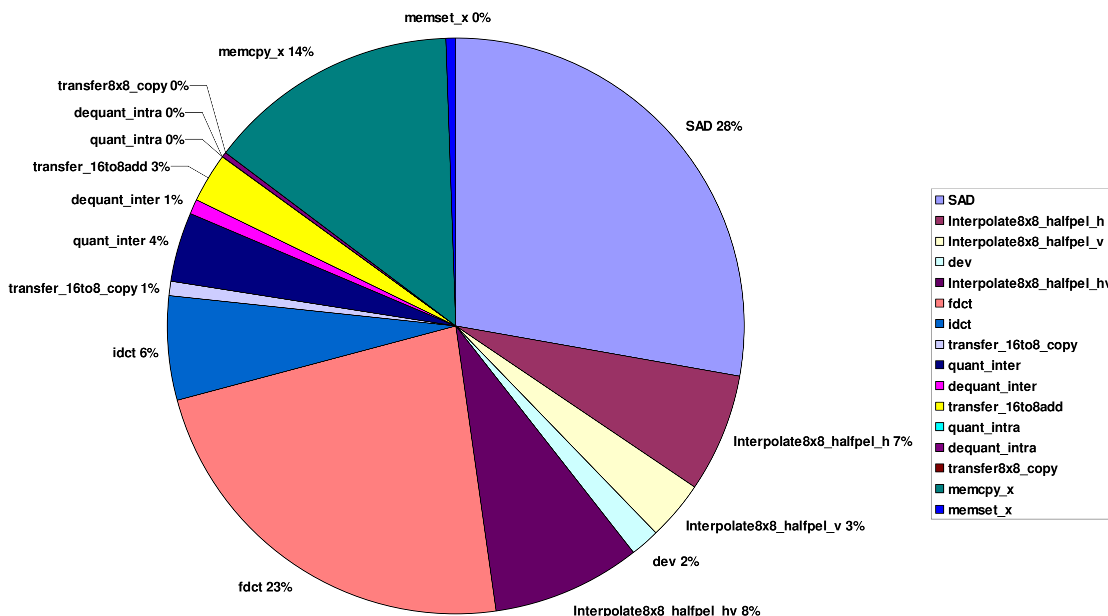


Figure 96. Percentage of cycles gained for each function from total gained cycles after optimizations

After the memory access optimizations into the MainDivX tasks, the new encoding speeds for the MPEG4 video encoder on the Quadric-Processors Platform are presented in Figure 97. A significant performance increase for the MPEG4 video encoder after the optimizations phase can be noticed.

Resol.	CPU1 tasks	CPU2 tasks	CPU3 tasks	CPU4 tasks	Encoding frame_rate	Bottleneck
QCIF	Splitter+Combiner	MainDivX1	MainDivX2	VLC	25 fps	CPU2,3
	Splitter+Combiner+VLC	MainDivX1	MainDivX2	MainDivX3	18 fps	CPU1
CIF	Splitter+Combiner	MainDivX1	MainDivX2	VLC	10 fps	CPU2,3
	Splitter+Combiner+VLC	MainDivX1	MainDivX2	MainDivX3	7 fps	CPU1

Figure 97. Encoding speeds for the MPEG4 video encoder after the optimizations

For some configurations, the new bottleneck is the CPU1. This is due to the mapping of the VLC along with the Splitter and Combiner, tasks on which no memory optimization were yet done. Thus, in order to increase even more the encoding performances, these tasks will also have to be optimized. As a result, the encoding performances may be bigger than 25 frames/sec. Based on personal estimations, the performances in this case might arrive at approximately 35 frames/sec.

In case of CIF video resolution, these optimizations are not sufficient to obtain a 25 frames/sec encoding performance. To conclude, HW accelerators or local coprocessors have to be integrated. Taking into account that these are improving mainly the computations of the encoding, they still might not be sufficient in case the memory becomes the bottleneck. In this case, the only feasible solutions would be to change the global memory architecture, increase the cache sizes, or to increase the platform's clock speed. However, these aspects will have to be covered during future works.

5.5 Conclusion

The objective of this chapter is to present the ROSES flow used during the implementation of the MPEG4 encoder on MP-SoC. The flow uses as input a *Flexible Algorithm/Architecture Model for MPEG4*, the algorithm/architecture configurations obtained

after the High-Level Algorithm/Architecture Exploration and a *Communication Library*. From these, different *Executable Models with Explicit Networks* can be macro-generated. Since in these models all the low-level architecture details are abstracted, the flow is automatically generating them using a component-based approach. The ROSES-ASOG tool is generating the low-level SW details (OS). The ROSES-ASAG tool is generating the low-level HW details (wrappers, CPU-Subsystems). Using the ROSES-COSIMX tool, it is possible to co-simulate the resulting models at different abstraction levels. This helps to validate and debug the results obtained by the flow.

Using this flow, multiple MPEG4 encoders on MP-SoC architectures were implemented, starting from high-level (*Executable Model with Explicit Network*) until RTL level. The resulted RTL architectures used different numbers of ARM processors. These architectures were validated using cycle-accurate ISS co-simulation approach.

Using the same flow, multiple MPEG4 encoders were implemented on an existing Quadric-processor platform. The implementation consists of obtaining the low-level SW details, and mapping them with the application's tasks on the platform's CPUs. The results were validated using native execution on the platform. Additionally, some algorithm optimizations were done in order to take advantage of the platform's features.

As a result, the used flow proved to be efficient for the implementation of the MPEG4 encoders on different MP-SoC. Because the flow is automated, it significantly reduces the design time.

6 Conclusions and perspectives

The implementation of video encoders, like MPEG4 encoder, into MP-SoC is largely required into many of today's applications: mobile telecom, home-cinema, video surveillance, etc. Since each of these applications imposes different algorithm and architecture constraints and requirements, the implementation of the MPEG4 encoder into MP-SoC raises many challenges. These challenges are related to different aspects:

a) being able to implement quickly different algorithm/architecture specifications for MPEG4, using different algorithm and architecture configurations, and different parallelism/pipeline execution schemes.

b) being able to explore quickly multiple algorithm and architecture configurations, in order to find the optimal configurations which are satisfying the performance requirements.

c) using a common flow for the implementation of the MPEG4 encoder on different types of targeted architectures, is either a completely new architecture, or an existing one.

This document presented a new approach that was successfully used for the implementation of the MPEG4 encoder in MP-SoC with different algorithm and architecture configurations. Using this approach, the MPEG4 encoder was successfully implemented on new architectures with large number of processors, architectures which may become a common practice in a few years (e.g. in 65nm technology in 2007). Additionally, the MPEG4 encoder was also successfully implemented into one of today's existing quadric-processors architecture. This was possible by using our proposed solutions:

a) a flexible modeling style was used to obtain automatically different *Combined Algorithm/Architecture Executable Models* for MPEG4 for different algorithm and architecture parameters, starting from a unique *Flexible Algorithm/Architecture Model* for MPEG4. These models contain a highly parallel/pipelined MPEG4 encoder algo-

rithm with 2 SMPs for the tasks with heavy computations. This algorithm can be parameterized (e.g. resolution, frame rate, bitrate, MotionEstimation precision, motion search area, quantization type, etc) and parallelizable, by adjusting the level of parallelism into each of the algorithm's SMP. The obtained models can easily be configured for multiple application requirements, but also for different architectures with either a small number of CPUs, or a large number of CPUs.

b) it was used a High-Level Algorithm/Architecture Exploration, in order to quickly explore multiple algorithm/architecture configurations, and to find the optimal parameters which are satisfying the requirements even before starting the actual architecture implementation. By doing the exploration at a high-level, many of the low-level architecture details are completely abstracted. This makes the simulation much faster. In addition, the simulation models were automatically obtained starting from a unique *Flexible Algorithm/Architecture Model for MPEG4*, from which many customized high-level simulation models can be automatically obtained, using a macro-generation approach. The performance estimations were done by inserting time annotations for computations and communications into these simulation models. This document presented many estimation results for different algorithm and architecture configurations. The precision of this high-level exploration approach proved to be close enough to the real measurements at a low-level. This assured the feasibility of our approach for this kind of applications.

c) a component based approach was used for the implementation of the MPEG4 encoder in either completely new architectures, or in an already existing architecture. This approach is based on the ROSES flow, developed at TIMA Laboratory – SLS Group. The main purpose of this flow is to refine the low-level details (i.e. HW/SW interfaces) which were abstracted during the previous high-level algorithm/architecture exploration. Using the flow presented in this document, the MPEG4 encoder was successfully implemented into completely new cycle-accurate MP-SoC architectures using ARM7 and ARM9 processors. The same approach was also used for the implementation of the MPEG4 encoder on an existing quadric-processors architectures (provided by one of our industrial partners), for multiple resolutions, frame rates, bitrates, etc. The final application is directly linked with a

video camera that is providing in real-time the images to the encoder chip, which compresses them, and saves them (with the help of a host driver) on a storage unit. Simultaneously, the resulting bitstream is displayed on the PC monitor using the Xine media player under Linux.

Based on the approach presented in this document, and in collaboration with our industrial partner, multiple other video applications were successfully implemented: MPEG1 encoder, MPEG1 decoder, MPEG2 encoder (SimpleProfile, MainProfile, HighProfile), MPEG2 decoder (SimpleProfile, MainProfile, HighProfile), MPEG4 decoder (SimpleProfile) and H264 encoder (Baseline, MainProfile, HighProfile). This was mainly possible by adapting the *Flexible Algorithm/Architecture Model for MPEG4* to support additionally Algorithm_Type and Encoder_Decoder parameters.

There are already plans for implementing the computational extensive functions into HW, and also to extend this approach to H264 decoder. However, extending the proposed approach to other application domains except the video applications is still an open research subject, and we have decided to leave it for future works.

Additionally, we intend to improve the precision of the High-Level Algorithm/Architecture Exploration, similar with the approach presented in [59].

Also, there is already ongoing work regarding the modeling of MPEG4 algorithm on MP-SoC using POSIX. Modeling it using the approach from [60] seems an interesting perspective, and we intend to achieve it in the future.

7 Bibliography

[1]ITRS, International Technology Roadmap for Semiconductors, Design, Edition 2001

[2]Nomadik, http://www.st.com/stonline/prodpres/dedicate/proc/proc.htm

[3]Nexperia, http://www.semiconductors.philips.com/products/nexperia/

[4]http://eet.com/news/latest/showArticle.jhtml?articleID=174909813

[5]Iain Richardson, Gary J. Sullivan, "*H.264 and MPEG-4 Video Compression*"

[6]MPEG AOE Group, "*Proposal Package Description (PPD) - Revision 3*", Doc. ISO/IEC JTC1/SC29/WG11 N998, Tokyo meeting, July 1995

[7]Fernando Pereira, "*MPEG-4: a new challenge for the representation of audio-visual information*", Keynote speech at Picture Coding Symposium' 96, Melbourne-Australia, March 1996

[8]http://www.projectmayo.com (project and web page closed in 2003)

[9]http://www.divx.com

[10]http://www.xvid.org

[11]Recommendation ITU-R BT.601-5, "*Studio encoding parameters of digital television for standard 4:3 and wide-screen 16:9 aspect ratios*", ITU-T, 1995

[12]Nam Ik Cho, Il Dong Yun, Sang Uk Lee, "*A Fast Algorithm for 2D-DCT*", International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 2197, May 1991, Toronto, Canada

[13]A.M. Tourapis et al, "*Optimizing the MPEG4 encoder – Advanced diamond zonal search*", IEEE International Symposium for Circuits and Systems, Vol. 3, pp. 674-677, May 28-31, 2000

[14]Jian-Wen Chen, Chao-Yan Kao, Youn-Long Lin, "*Introduction to H.264 Advanced Video Coding*", Asian and South Pacific Design and Automation Conference 2006, ASP-DAC2006, Special Session Presentation, pp. 736-741, Yokohama, JAPAN, Jan. 2006

[15]S.W.Golomb, "*Run-length encoding*", IEEE Trans. On Inf. Theory, pp.399-401, 1966

[16]D.Huffman,"*A method for the construction of minimum redundancy codes*", Proceedings of IRE, pp.1098-1101, 1952

[17]eXpressDSP compliant MPEG4 Simple Profile Video Encoder for TI TMS320C64x DSPs, http://focus.ti.com/catalog/docs/thirdpartysoftwarefolder.tsp?softwareId=193

[18]AV140 Video Recorder, http://www.archos.com/products/prw_500431.html

[19]Pierre Paulin et al , "*Parallel Programming Models for a Multi-Processor SoC Platform Applied to High-Speed Traffic Management*", Best paper, ISSS/CODES 2004, pp. 48-53, September 2004, Stockholm, Sweden

[20]Yong He et al: "*A Software Based MPEG-4 Video Encoder Using Parallel Processing*", IEEE Trans. on Circuits and Systems for Video Tech., Vol.8, No. 7, pp. 909-920, Nov.1998

[21]Mrudula Yadav, Gayathri Venkat, B.L. Evans, "*Modeling and Simulation on H.26L Encoder*", Final Report for EE382C Embedded Software Systems, May 8, 2002

[22]M.Raulet et al, "*Automatic coarse-grain partitioning and automatic code generation for heterogeneous architectures*", in proceedings of IEEE Workshop on Signal Processing Systems, SiPS'03, Seoul, Korea, August 2003

[23]Mohamed Wassim Youssef, Sungjoo Yoo, Arif Sasongko, Yanick Paviot, Ahmed Jerraya, "*Debugging HW/SW Interface for Multiprocessor SoC: Video Encoder System Design Case*", 41st DAC, pp. 908-913, San Diego, CA, June 2004

[24]Pieter van der Wolf et al, „*Design and Programming of Embedded Multiprocessors: An Interface-Centric Approach*", CODES-ISSS, pp. 206-217, Stockholm, Sweden, September 2004

[25]Satya Kiran M.N.V et al, "*A Complexity Effective Communication Model for Behavioural Modelling of Signal Processing Applications*", 40th DAC, pp. 412-415, Anaheim, CA, June 2003

[26]L. Formaggio et al, "A *Timing-Accurate HW/SW Co-Simulation of an ISS with SystemC*", CODES-ISSS 2004, pp.152-157, Stockholm, Sweden, September 2004

[27]SystemC User's Guide, www.systemc.org

[28]Atomium Project, IMEC University, http://www.imec.be/design/atomium

[29]Xtensa LX Xplorer 1.0.1, www.tensilica.com

[30]Chia-Chu Chiang, "*High-Level Heterogenous Distributed Parallel Programming*", Proceedings of the 2004 international symposium on Information and communication technologies, pp. 250-255, Las Vegas, Nevada, June 2004.

[31]MPICH – A Portable MPI Implementation, http://www-unix.mcs.anl.gov/mpi/mpich/

[32]M4 - a GNU implementation of the UNIX macro processor, http://www.seindal.dk/rene/gnu/whatis.htm

[33]Luciano Lavagno et al, "*Specification, Modelling and Design Tools for System-on-Chip*", ASP-DAC 2002/VLSI Design, pp.21, January 07-11, 2002

[34]ARM Developer Studio, www.arm.com

[35]VTune™ Performance Analyzer, http://support.intel.com/support/performancetools/vtune/

[36]ARM7 & ARM946E-S, http://www.arm.com/products/CPUs/index.html

[37]Tensilica XTENSA-LX configurable processors, http://www.tensilica.com

[38]Christophe Wolinski, Maya Gokhale, Kevin McCabe, "*A Polymorphous Computing Fabric*", IEEE Micro, Volume 22, Number 1, pp. 56-68, January/February 2002,

[39]Sherry Solden, ”*Architectural Services Modeling for Performance in HW-SW Co-design*”, Proceedings of the Workshop on Synthesis And System Integration of Mixed Technologies, IEEE Press, pp. 72-77, 2001

[40]S.I. Han, A. Baghdadi, M. Bonaciu, S.I. Chae, A.A.Jerraya, "*An Efficient Scalable and Flexible Data Transfer Architecture for Multiprocessor SoC with Massive Distributed Memory*", 41st DAC, pp. 250-255, San Diego, CA, June 2004

[41]Gauthier Lovic, "*Génération de système d'exploitation pour le ciblage de logiciel multitâche sur des architectures multiprocesseurs hétérogènes dans le cadre des systèmes embarqués spécifiques*", PhD thesis, INPG, Microelectronics, TIMA Laboratory, May 2001

[42]Cesario W., Paviot Y., Baghdadi A., Lovic G., Damian L., Nicolescu G., Yoo S., Jerraya A., Diaznava M., "*HW/SW interfaces design of a VDSL modem using automatic refinement of a virtual architecture specification into a multiprocessor SoC: a case study*", Design Automation and Test in Europe (DATE'02), pp. 165-169, Paris, France, March 4-8, 2002

[43]Gabriela Nicolescu, "*Spécification et validation des systèmes hètèrogènes embarqués*", PhD thesis, INPG, Microelectronics, TIMA Laboratory, 2002

[44]Cesario W., Nicolescu G., Gauthier L., Lyonnard D., Jerraya A., "*Colif: a Multilevel Design Representation for Application-Specific Multiprocessor System-on-Chip Design*", in Proceedings of Twelve IEEE International Workshop on Rapid System Prototyping, pp.110-116, California, USA, June 2001

[45]Paviot Yanick, "*Partitionnement des services de communication en vue de la génération automatique des interfaces logicielles/matérielles*", PhD thesis, INPG, Microelectronics, TIMA Laboratory, July 2004

[46]Adriano Sarmento, "*Génération automatique de modèles de simulation pour la validation de systèmes hétérogènes embarqués*", PhD thesis, TIMA Laboratory, SLS Group, Grenoble, France, 2005

[47]A. A. Jerraya, W. Wolf, ”*Multiprocessor Systems-on-Chips*”, Morgan Kaufmann Publishers, ISBN 0-12-385251-X, September 2004

[48]http://www.fourcc.org/yuv.php

[49]Tung-Chien Chen, Chung-Jr Lian, Lian-Gee Chen, "*Hardware Architecture Design of an H.264/AVC Video Codec*", 11th Asian and South-Pacific Design Automation Conference, ASP-DAC 2006, pp. 750-757, Yokohama, Japan, January 2006

[50]Hung-Chih Lin, Yu-Jen Wang, Kai-Ting Cheng, Shang-Yu Yeh, Wei-Nien Chen, Chia-Yang Tsai, Tian-Sheuan Chang, Hsueh-Ming Hang, "*Algorithms and DSP Implementation of H.264/AVC*", 11th Asian and South-Pacific Design Automation Conference, ASP-DAC 2006, pp. 742-749, Yokohama, Japan, January 2006

[51]Sung Dae Kim, Jeong Hoo Lee, Chung Jin Hyun, Myung Hoon Sunwoo, "*ASIP Approach for Implementation of H.264/AVC*", 11th Asian and South-Pacific Design Automation Conference, ASP-DAC 2006, pp. 758-764, Yokohama, Japan, January 2006

[52]Muhammad Omer Cheema, Omar Hammami, "*Customized SIMD unit synthesis for system on programmable chip: a foundation of HW/SW partitioning with vectorization*", 11th Asian and South-Pacific Design Automation Conference, ASP-DAC 2006, pp. 54-60, Yokohama, Japan, January 2006

[53]http://en.wikipedia.org/wiki/Symmetric_multiprocessing

[54]Thomas Braunl, Stefan Feyrer, Wolfgang Rapf, Hans Walischmiller, Michael Reinhardt,"*Parallel Image Processing*", Springer-Verlag Berlin Heidelberg 2001, ISBN 3-540-67400-4, Germany

[55]http://www.mds.com/products/ProductPages/ /NXPSW-MP4DECODER/pnx1300/ipbrief_VdecMpeg4.pdf

[56]Doina Zmaranda, Marius Bonaciu: "*Data Structures and Advanced Programming Techniques*", ISBN 973-613-302-8, University of Oradea, Romania, Faculty of Electrotechnics and Informatics , Computers Department, Oradea University Editor, May 2003

[57]Lobna Kriaa, "*Modélisation et Validation des systèmes hétérogènes : Définition d'un modèle d'exécution*", PhD thesis, UJF, Informatics, TIMA Laboratory, Nov. 2005

[58]Mohamed Wassim Youssef, "*Etudes des interfaces logicielles/matérielles dans le cadre des systèmes multiprocesseurs monopuces et des modèles de programmation parallele de haut niveau*",PhD thesis, UJF, Informatics, TIMA Laboratory, Mar. 2006

[59]Hector Posadas, F.Herrera, Pablo Sánchez, Eugenio Villar, Francisco Blasco, "*System-Level Performance Analysis in SystemC*", DATE'04, pp.378-383, Paris, France, February 2004

[60]Hector Posadas, Jesús Ádamez, Pablo Sánchez, Eugenio Villar, Francisco Blasco, "*POSIX modeling in SystemC*", 11th Asian and South-Pacific Design Automation Conference, ASP-DAC 2006, pp. 485-490, Yokohama, Japan, January 2006

[61]Florin Dumitrascu, Iuliana Bacivarov, Lorenzo Pieralisi, Marius Bonaciu, Ahmed Jerraya, "*Flexible MPSoC platform with fast interconnect exploration for optimal system performance for a specific application*", Design Automation and Test in Europe 2006, DATE'06, pp. 166-171, Munich, Germany, March 2006

[62]Aimen Bouchima, Iuliana Bacivarov, Wassim Youssef, Marius Bonaciu, Ahmed Jerraya, "*Using Abstract CPU Subsystem Simulation Model for High Level HW/SW Architecture Model*", Asian and South Pacific Design Automation Conference 2005, ASP-DAC'2005, pp. 372-377, Shanghai, China, January 2005

RESUME

Ces dernières années, la complexité des puces a augmenté exponentiellement. La possibilité d'intégrer plusieurs processeurs sur la même puce représente un gain important, et amène au concept du système multiprocesseur hétérogène sur puce (MP-SoC). Cet aspect a permis d'amplifier de manière significative la puissance de calcule fourni par ce type de puce. Il est même devenu possible d'intégrer des applications complexes sur une seule puce, applications qui nécessitent beaucoup de calculs, de communications et de mémoires. Dans cette catégorie, on peut trouver les applications de traitement vidéo MPEG4. Pour obtenir de bonnes implémentations en termes de performances, (1) un algorithme de l'encodeur MPEG4 flexible a été réalisé, pouvant être facilement adapté pour différents types de paramètres d'algorithme, mais également différents niveaux de parallélisme/pipeline. Puis, (2) une modélisation flexible a été utilisée, pour représenter différents models d'algorithme et d'architecture contenant 2 SMP. Utilisant ces models, (3) une exploration d'algorithme et d'architecture à un haut niveau d'abstraction a été proposé, en vue de trouver les configurations correctes d'algorithme et d'architectures, nécessaires pour différents applications. A partir de ces configurations, (4) un flot automatique d'implémentation d'architectures RTL a été utilisé. En utilisant ces aspects, l'encodeur MPEG4 a été implémenté avec succès dans plusieurs architectures spécifiques MP-SoC au niveau RTL. La même approche a été utilisée pour l'implémentation de l'encodeur MPEG4 sur une architecture quadri-processeurs existante, pour différentes résolutions, frame-rate, bitrates, etc.

TITRE EN ANGLAIS

Flexible and Scalable Algorithm/Architecture Platform for MP-SoC Design of High Definition Video Compression Algorithms

---

ABSTRACT

During the last years, the chip's complexity increased exponentially. The possibility to integrate multiple processors into the same chip represents an important gain, at it leads to the concept of Multi-Processors Systems on Chip (MP-SoC). This aspect allowed boosting the computational power offered by the chips. Thus, it became possible to integrate complex applications into a chip, applications that requires a large amount of computations, communications and memory. In this category, we can find the video treatment applications, like the MPEG4. To obtain good implementation results in term of performance, (1) a flexible MPEG4 encoder algorithm was developed, which can be easily adapted for different algorithm parameters, and different parallel/pipeline execution schemes. After this, (2) a flexible modeling was used, in order to represent different algorithm/architecture models containing 2 SMPs. Using these models, (3) a high-level algorithm/architecture exploration method was used, to find the optimal algorithm/architecture configurations required by different applications (i.e. mobile telecom). Using these parameters, (4) an automatic flow was used, to obtain final RTL architectures containing the MPEG4 encoder. Using all these aspects, the MPEG4 encoder was successfully implemented into multiple specific RTL architectures. Additionally, using the same approach, the MPEG4 encoder was implemented on an existing quadric-processors platform, for different video resolutions, frame rates, bitrates, etc.

SPECIALITE

Micro et Nano Electronique

MOTS CLES

MP-SoC, MPEG4, Vidéo, Exploration, Flexibilité, Parallélisme, Pipeline, Implémentation

---

INTITULE ET ADRESSE DU LABORATOIRE

Laboratoire TIMA, 46 Avenue Félix Viallet, 38031 Grenoble

ISBN : 2-84813-091-1
ISBNE : 2-84813-091-1