| | |
|---|---|
| **Titre:** Title: | Exploration d'une méthodologie de développement matériel et logiciel au niveau système appliqué à un système d'encodage de flux vidéo évolutif |
| **Auteur:** Author: | Etienne Gauthier |
| **Date:** | 2017 |
| **Type:** | Mémoire ou thèse / Dissertation or Thesis |
| **Référence:** Citation: | Gauthier, E. (2017). Exploration d'une méthodologie de développement matériel et logiciel au niveau système appliqué à un système d'encodage de flux vidéo évolutif [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/2747/ |

UNIVERSITÉ DE MONTRÉAL

# EXPLORATION D'UNE MÉTHODOLOGIE DE DÉVELOPPEMENT MATÉRIEL ET LOGICIEL AU NIVEAU SYSTÈME APPLIQUÉ À UN SYSTÈME D'ENCODAGE DE FLUX VIDÉO ÉVOLUTIF

ETIENNE GAUTHIER
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2017

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

# EXPLORATION D'UNE MÉTHODOLOGIE DE DÉVELOPPEMENT MATÉRIEL ET LOGICIEL AU NIVEAU SYSTÈME APPLIQUÉ À UN SYSTÈME D'ENCODAGE DE FLUX VIDÉO ÉVOLUTIF

présenté par : GAUTHIER Etienne
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

Mme NICOLESCU Gabriela, Doctorat, présidente
M. BOIS Guy, Ph. D., membre et directeur de recherche
M. DAVID Jean Pierre, Ph. D., membre

# DÉDICACE

*À mon père qui est pour moi l'exemple de la persévérance, de la détermination et de l'intégrité.*
*Tu m'as prouvé qu'on pouvait accomplir de grandes choses lorsqu'on se donnait à fond.*
*Merci de m'avoir encouragé toutes ces années...*

## REMERCIEMENTS

J'aimerais tout d'abord remercier mon directeur de recherche, le professeur Guy Bois, qui m'a guidé durant ce projet. Votre support et vos conseils m'ont été d'une grande aide pour bien orienter ma recherche.

J'aimerais aussi remercier mes collègues de laboratoire Mathieu Goudron et Arnaud Desaulty d'avoir rendu chaque journée plus intéressante et enrichissante au laboratoire.

Finalement, je tiens à remercier l'équipe de SpaceCodesign plus particulièrement Hubert Guérard et Fellipe Monteiro qui m'ont offert un excellent support pour l'application SpaceStudio.

# RÉSUMÉ

La compagnie Grass Valley, fabricant de cartes de traitement vidéo, désire mettre à jour leur sous-système « thumbnail » qui produit des vidéos à échelle réduite à des fins de diagnostic. Afin de le moderniser, ils ont arrêté leur choix sur une implémentation d'un « proxy » vidéo produisant un flux vidéo compressé avec la norme H.264. Afin d'épargner en coût de développement et assurer son indépendance au cycle de vie des composantes tierces, Grass Valley est à la recherche d'une implémentation évolutive et indépendante d'une plateforme. Afin de résoudre ce problème, Grass Valley a fait appel à Polytechnique.

Le développement d'un encodeur H.264 pour système sur puce personnalisé peut nécessiter plusieurs mois à plusieurs années de développement pour une équipe d'ingénierie. Il existe actuellement peu de solutions possibles pour concevoir un tel sous-système rapidement. Afin de développer le sous-système d'encodage H.264 rapidement, nous avons opté pour une méthodologie de développement à l'aide de l'approche du point de vue du système basée sur une spécification exécutable d'un encodeur H.264 en utilisant l'outil SpaceStudio.

SpaceStudio est un logiciel permettant l'exploration architecturale à l'aide de plateforme virtuelle configurable. La conception de système à l'aide de cet outil se fait par une approche modulaire sous SystemC. Le système est séparé en module logiciel et matériel fonctionnel et ceux-ci sont développés itérativement. L'utilisation d'un code applicatif comme base afin d'en produire un système embarqué sous SpaceStudio n'a pas été expérimentée. Dans cette optique, ce travail à deux objectifs : 1) développer un système pouvant encoder un flux vidéo et 2) expérimenter avec une approche de développement du point de vue du système à l'aide d'une spécification exécutable sous SpaceStudio. Il est donc question de développer la méthodologie et le projet en parallèle.

Au terme de ce projet, nous aurons implémenté un système d'encodage H.264 sur une plateforme virtuel et défini la méthodologie nécessaire afin de produire un système sur puce à l'aide d'une référence logicielle. Cette recherche nous a permis de découvrir les obstacles à la conception de système complexe à l'aide de code C/C++ existant sous SpaceStudio et de développer les bases nécessaires pour rendre la totalité de la méthodologie réalisable dans le futur.

# ABSTRACT

GrassValley, a manufacturer of video processing cards, wants to upgrade their thumbnail subsystem which produces scaled-down videos for diagnostic purposes. In order to modernize this subsystem, they have decided to go with a video proxy producing a video stream compressed with the H.264 standard. In order to save development costs and ensure its independence of third-party components, Grass Valley is looking for a scalable platform-independent implementation. To solve this problem, they called upon Polytechnique.

The development of an H.264 encoder for custom system-on-a-chip may take several months to several years of development for an engineering team. There are currently very few possible solutions to design such a subsystem quickly. In order to do so, we opted for a development methodology using the system-level approach based on an executable specification of an H.264 encoder using SpaceStudio.

SpaceStudio is a computer aided design software for architectural exploration using a configurable virtual platform. Designing a system with this tool is done through a modular approach using the SystemC library. The designed system is separated into functional software / hardware modules developed iteratively. The use of a software application as a basis to produce a system under SpaceStudio has not been yet tested. This work has two objectives: 1) to develop a system capable of encoding a video stream and 2) to experiment with a system level development approach using a executable specification under SpaceStudio. It is therefore a question of developing the methodology and the encoder in parallel.

At the end of this project, we will have implemented a H.264 encoding system on a virtual platform and defined the methodology needed to produce a full system on chip using a software reference as a basis for development. This research allowed us to discover the obstacles associated to the design of a complex systems using C "legacy" code under SpaceStudio and to develop the necessary tools to make the whole methodology achievable in the future.

# TABLE DES MATIÈRES

# LISTE DES TABLEAUX

# LISTE DES FIGURES

# LISTE DES SIGLES ET ABRÉVIATIONS

| | |
|---|---|
| ACP | Accelerator Coherency Port |
| AVC | Advanced Video Coding |
| CABAC | Context Adaptive Binary Arithmetic Coding |
| CAPI | Coherent Accelerator Processor Interface |
| CAVLC | Context Adaptive Variable Length Coding |
| EDA | Electronic Design Automation |
| ESL | Electronic System-level |
| FME | Fractionnal motion estimation |
| FPGA | Field-Programmable Gate Array |
| HLS | High level synthesis |
| IDR | instantaneous decoding refresh |
| IME | Integral motion estimation |
| ISS | Instruction Set Simulator |
| ITU | International Telegraph Union |
| ITU-T | ITU Telecommunication Standardization Sector |
| JVT | Joint Video Team |
| MPEG | Moving Picture Experts Group |
| PPS | picture parameter sets |
| RBSP | Raw byte sequence payload |
| RTL | Register transfert level |
| SoC | System on Chip |
| SPS | sequence parameter sets |
| TLM | Transaction level modeling |
| UMA | Unified Memory Architecture |

# LISTE DES ANNEXES

# CHAPITRE 1 INTRODUCTION

La compagnie Grass Valley conçoit un grand nombre de systèmes de traitement vidéo. Inclu avec ces systèmes de traitement vidéo se trouve un sous-système qui produit des vidéos de type « thumbnail ». Ceux-ci consistent en l'acquisition de petites images instantanées prises dans le traitement principal d'un flot vidéo. Ces images sont donc des versions à échelle réduite des images originales et sont utilisés pour surveiller, assurer la qualité ainsi que le bon fonctionnement d'une production télévisuelle. La compagnie Grass Valley est en processus de modernisation de sa gamme de produits et aimerait améliorer les fonctionnalités de ce sous-système tout en respectant diverses contraintes associées à la consommation de puissance et au coût de production. De plus, elle aimerait migrer vers un système offrant des images suivant un standard plus moderne, laissant de côté l'approche basse qualité actuellement offerte.

Il existe actuellement deux solutions possibles pour concevoir un sous-système pour la création de séquence vidéo dit « thumbnail ». La première consiste en l'utilisation seulement d'un processeur bas de gamme, par exemple ARM. Cette option est la moins chère, mais offre de moins bonnes performances comparativement à la seconde qui consiste en l'utilisation d'un processeur commercial spécialisé jumelé à une logique dédiée pour l'accélération d'encodage/-décodage. Celle-ci offre de meilleures performances comparativement à la première solution, mais l'évolutivité n'est pas garantie en plus de devoir faire face à un potentiel problème d'obsolescence lorsque le processeur spécialisé arrive en fin de vie. D'énormes coûts peuvent ainsi être engendrés afin de recibler l'architecture initiale vers une nouvelle plateforme.

Afin de pallier en coût de développement et assurer son indépendance au cycle de vie des composants tierce, la compagnie Grass Valley est à la recherche d'un système indépendant et évolutif pour leur prochaine version de leur sous-système « thumbnail ». Celui-ci doit être réalisé à l'aide de composantes pouvant être implémentées sur FPGA facilitant ainsi ces deux points. De plus, l'utilisation de la spécification H.264 pour l'encodage des images a été choisie. Ce point est particulièrement important, car non seulement une telle spécification est très complexe, mais cela demande aussi une profonde exploration architecturale afin de bien évaluer et répartir la charge des algorithmes vers une implémentation logicielle ou matérielle. Il est donc nécessaire de développer un nouveau flot de conception afin de livrer un prototype rapidement malgré la complexité tout en tenant compte des restrictions imposées par le client.

## 1.1 Éléments de la problématique

La création de circuits numérique est un processus dont la complexité ne cesse de grandir depuis l'arrivée des circuits intégrés. Chaque nouvelle génération du processus permet d'intégrer plus de transistors sur une même puce suivant la loi de Moore [31]. Or, afin d'utiliser ces nouvelles ressources, les concepteurs doivent produire des modules permettant l'utilisation de logique plus complexe.

Pour augmenter la productivité des ingénieurs, un flot de conception s'est tranquillement développé. On cherchait à normaliser la description de système et trouver un langage commun désignant les procédés numériques. De cela sont nés plusieurs langages de description matérielle tels que ABEL, AHPL , ELLA , VHDL et Verilog. Par la suite, ont été développés des logiciels pouvant simuler ces descriptions, ainsi que des synthétiseurs logiques traduisant ces modèles en porte élémentaire et pouvant être éventuellement intégré sur un support physique (silicium).

Bien que cette approche apporte plus de flexibilité, elle n'est plus à elle seule suffisante afin de réduire le manque de productivité apporté par le développement des systèmes moderne. Ces derniers se sont complexifiés rapidement. Il suffit de regarder les téléphones portables modernes : unité graphique (GPU), décodeur, encodeur, processeur haute performance, processeur applicatif, etc. Il ne s'agit plus seulement de produire un seul module, mais des systèmes hétérogènes complexes sur puce tout comme l'encodeur/décodeur H.264 à développer pour la compagnie Grass Valley [44, 38, 42] . Un module matériel peut être créé et ajouté au système afin d'augmenter la performance de celui-ci et réduire la consommation. Un tel module nommé logique dédiée est toujours plus performant. Afin de pallier l'augmentation des besoins d'accélération, les logiciels de synthèse algorithmique ont fait leur apparition. Ceux-ci sont en mesure de prendre un code logiciel et de créer un modèle RTL synthétisable. À l'aide de ces outils, nous avons maintenant une approche plus automatisée pour passer du modèle logiciel au modèle matériel.

### 1.1.1 Problématique principale

Bien qu'ils facilitent la tâche des ingénieurs, les logiciels de synthèse algorithmique ne répondent pas à toutes ces nouvelles questions qui font surface lors de la création de systèmes hétérogènes. Par exemple, quelle partie du système doit être accélérée à l'aide de logique

dédiée et de quelles façons ? Est-ce que telle portion en matériel me permet une meilleure performance plutôt que tel autre ? Quels modèles de communication utiliser ? Ces questions sont encore plus difficiles lorsqu'on doit aussi respecter les objectifs de consommation (thermique), de surface (coût) et de performance. Dans cette optique, il ne s'agit plus d'utiliser une grande quantité de transistors, mais aussi de les utiliser le plus efficacement possible. Comment faisons-nous pour trouver la meilleure configuration pour le système, afin de respecter tous ces requis surtout lorsque ceux-ci peuvent être orthogonaux ? Comment pouvons-nous connaître les compromis le plus tôt possible lors de la conception pour réduire le risque d'erreur ? Une approche de bas vers le haut « bottom up » traditionnelle utilisant seulement les langages de description matérielle ou la synthèse algorithmique ne peut certifier que le produit final saura répondre à ces exigences. Cette méthodologie considère que ces décisions ont déjà été prises préalablement. Si une erreur s'est glissée lors des spécifications, qu'une mauvaise élaboration a été choisie ou si nous voulons tester d'autres architectures, le temps de conception sera grandement prolongé si celui-ci nécessite beaucoup d'itération afin d'atteindre ces différents objectifs. Même chose si la plateforme cible change (différents processeurs, mémoires, etc.). L'itération de logique est toujours plus ardue lorsqu'on travaille à plus bas niveau (RTL).

Il est alors nécessaire de développer une méthodologie permettant de travailler à l'aide d'une plus grande abstraction et d'être en mesure de faire la recherche de solutions et de modéliser les communications le plus rapidement possible dans la conception d'un système. Afin d'être en mesure d'effectuer une exploration architecturale tôt dans le développement on se doit de réduire le couplage des communications et les détails d'implémentation matérielle ou du moins en cacher l'implémentation. On doit travailler sur une vue système en terme de tâches que celui-ci doit accomplir et leurs interactions plutôt que de penser en terme de composantes (Processeurs, coprocesseurs, logiques dédiées, etc.). On doit donc travailler à très haut niveau comparativement aux méthodes de développement actuelles. Plusieurs articles sont disponibles sur l'implémentation matérielle en format RTL (circuit numérique) de diverse partie de l'algorithme d'encodage H.264. Or, ces implémentations RTL sont, par définition, liées à une architecture et peuvent être difficilement portées vers d'autres (changement de processeur ou fabrique). De plus, les optimisations appliquées en RTL ne peuvent directement s'appliquer lors de la conception d'un modèle plus abstrait du système. Jusqu'à maintenant, les méthodologies de développement du point de vue du système sont très peu couvertes et nécessitent encore une grande portion de développement manuel.

### 1.1.2 Problématique explorée dans ce mémoire

Les méthodologies de conception actuelles prévoient un modèle système à des fins d'évaluation fonctionnelle et de référence aussi appelé « golden model ». Cette référence exécutable n'est pas elle-même utilisée pour le produit final, c'est-à-dire que le code fonctionnel du modèle, préalablement validé, peut ne pas être réutilisé pour l'élaboration même du système sur puce. Cette référence qui est alors du jetable pourrait servir comme base afin de réduire le temps entre la conception et la création du système. Ce format exécutable devient la référence du projet et c'est celle-ci qui sera transformée en système sur puce. De cette façon, le temps de développement fait sur la spécification est alors sauvé par sa réutilisation et permet une mise en marché plus rapide. Il s'agit donc d'augmenter l'intégration et l'utilisation du modèle exécutable dans le flot de conception.

Un système tel que demandé par Grass Valley est un bon exemple de cette problématique. Nous disposons d'un grand nombre d'implémentations logicielles d'encodeur/décodeur H.264 et il serait intéressant de les réutiliser afin d'accélérer la production de l'encodeur sur puce. L'implémentation complète de la norme à partir de ces spécifications nécessite une très grande charge de travail qui dépasse le cadre de cette maîtrise.

## 1.2 Objectifs de recherche

Ce projet comporte deux principaux volets. Le premier volet consiste à explorer la possibilité de développer un système sur puce (SoC) à l'aide d'un modèle système, examiner la possibilité d'intégrer les techniques de codesign matériel/logiciel ainsi qu'ESL (Electronic System Level) et définir la méthodologie requise pour mettre à terme ce type de projet. Ce que nous proposons est le développement et la modélisation du système en entier à l'aide d'un langage haut niveau sans déterminer les particularités logicielles et matérielles préalablement, mais que ce choix soit fait lors de l'exploration architecturale. Le modèle devra être découplé de son architecture pouvant alors être modifiée et reciblée rapidement. Ce sont ces choix architecturaux qui deviendront l'implémentation concrète du système qui sera ensuite synthétisé vers une puce FPGA.

Le second volet porte sur le développement d'un module « proxy » qui agira comme encodeur H.264 afin de remplacer le module « Thumbnail » de Grass Valley et de leur fournir une plateforme évolutive leur permettant la mise à jour de celui-ci à un faible coût tout en

évitant le problème de dépendance à une tierce partie. Ce module servira donc de cas de figure pour la recherche d'une méthodologie de développement au point de vue du système mentionné au premier volet.

## 1.3 Résumé des contributions

Ce projet de maîtrise a conduit aux contributions suivantes :

1. Démonstration de la faisabilité de développer un encodeur H.264 à partir d'une référence logicielle utilisée en entrée d'un processus de codesign logiciel/matériel. (Autant que nous sachions, il s'agit là d'une première.)
2. À partir des résultats du point 1), proposition d'étapes d'une exploration architecturale afin de déplacer un code applicatif vers une implémentation matérielle.
3. Expérimentation du développement d'un SoC encodeur H.264 logiciel/matériel s'exécutant sur une plateforme virtuelle du FPGA Zynq de Xilinx. Cette même plate-forme virtuelle servant à faire la validation de l'exploration architecturale.
4. L'identification d'obstacles qui rend complexe la migration d'algorithmes logiciels vers un modèle matériel durant l'exploration architecturale.
5. Réalisation d'une preuve de concept visant à éliminer certains obstacles qui n'ont pu être traités dans le cadre de ce projet et qui serviront aux travaux futurs.

## 1.4 Plan du mémoire

Ce mémoire est organisé comme suit. Le second chapitre sera dédié à la revue de littérature traitant du codesign, de la méthodologie ESL, du développement système et de la norme H.264. Le troisième chapitre sera dédié à l'expérimentation de la méthodologie et le développement de l'encodeur alors que le quatrième chapitre présentera les résultats. Le chapitre 5 fera suite au résultat avec une exploration afin d'améliorer le processus de développement. Finalement le chapitre 6 présente la conclusion et les travaux futurs.

# CHAPITRE 2 REVUE DE LITTÉRATURE

Ce chapitre se veut un survol des domaines pertinents pour la compréhension de ce mémoire. Nous allons d'abord définir les concepts de codesign, de la méthodologie ESL et de divers outils nécessaires à l'application de celle-ci. Nous présenterons par la suite l'approche du développement système et les divers travaux effectués dans ce domaine, pour ensuite poursuivre sur une introduction de la norme H.264. Finalement, quelques implémentations d'encodeur H.264 seront présentées.

## 2.1 Le codesign

La méthodologie codesign [20, 23, 44, 51] dans le domaine des circuits numériques apporte une solution afin d'accélérer l'exploration architecturale d'un système logiciel et matériel. Il s'agit donc de concevoir à l'aide d'une vue système plutôt qu'une vue par module. De haut vers le bas « Top down » plutôt que de bas vers le haut « Bottom up ». Le but est d'atteindre plus facilement les contraintes telles que le coût, la performance et la puissance tout en réduisant le temps de développement nécessaire pour la mise en marché d'un produit.

Cette méthodologie invite les membres des équipes logicielles (OS, firmware, et application) à travailler de pair avec l'équipe de développeurs matériels sur tous les modules du système. On veut réduire le temps de mise en marché, alors le développement logiciel et matériel doit se faire en parallèle afin d'augmenter la productivité comparativement à l'approche séquentielle traditionnelle. Tout comme la validation du produit qui doit aussi se faire sur la totalité du système [44].

La première ébauche du codesign tentait de formuler une approche aux problèmes de partitionnement des tâches et leur ordonnancement sur une plateforme multiprocesseur [44]. Elle consistait à des plateformes fixes comportant un processeur et un groupe de coprocesseur de type ASIC communiquant sur un bus et utilisant de la mémoire partagée. Les développeurs analysaient la complexité algorithmique et déterminaient quelles tâches étaient candidates à la conception de modules matériels et la plateforme était fixe tout le long du développement. Aussi, ce modèle n'avait aucune notion de concurrence. Une tâche s'exécute sur le processeur et celui-ci attend après ses coprocesseurs.

La seconde approche ajouta l'inclusion de plusieurs processeurs, le support d'exécution de plusieurs tâches en parallèle et apporta un des éléments clés du codesign, la cosimulation [44]. La cosimulation consiste en la simulation de l'environnement logiciel et matériel sous le même système [40, 47]. À ce moment, on utilisait un modèle virtuel des processeurs sur lequel on exécutait le code logiciel ou sous une version synthétisé des modules matériels. Le modèle virtuel est l'approche la plus souvent utilisée, car la simulation par RTL est typiquement trop lente. Une chose était claire, on se devait d'aller plus haut en abstraction afin de gagner en temps d'exécution. Il s'agit de gérer correctement l'abstraction en fonction de la précision voulue [20].

De nos jours, les systèmes modernes sont beaucoup plus complexes et remettent en question deux hypothèses à la base des premières générations de codesign. La première hypothèse stipulant que la plateforme donnée est fixe. Ceci n'est plus toujours le cas. En effet, les systèmes sont maintenant hétérogènes [49, 38] et souvent taillés sur mesure selon leurs fonctions [46, 25, 52, 38]. La seconde hypothèse dit qu'il est possible de minimiser ou maximiser le coût et la performance par l'approche de techniques d'optimisation. Étant donné que chaque produit nécessite alors une approche particulière, ceci devient grandement complexe étant donné le nombre de paramètres à optimiser. Afin de répondre adéquatement au besoin des utilisateurs, une application suit habituellement un groupe de requis. Par exemple, une acquisition de 30 images seconde, un débit de communication, etc. Or, un système embarqué possède lui aussi des requis qui sont de différentes natures telles que la performance (la rapidité de traitement), les ressources disponibles (espace, transistors, etc.), la consommation (watt), la fiabilité, le coût, etc. Tel qu'illustré à la figure 2.1, les fonctions d'optimisation pilotant le partitionnement doivent donc être flexibles et à multiples objectifs. Ceci est considéré comme un problème NP complet [45].

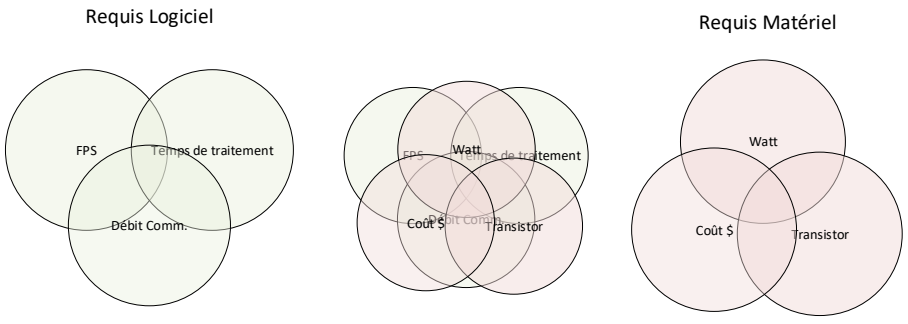



Figure 2.1 Espace de solution matériel/logiciel

## 2.2 Méthodologie ESL

Afin de faire face à la croissance en complexité des systèmes sur puce, le codesign s'est modernisé avec l'arrivée de la méthodologie « Electronic System Level » (ESL). L'approche ESL a pour but de simplifier le développement de système électronique/informatique en proposant d'augmenter le niveau d'abstraction des spécifications, de l'analyse, de la conception et de la vérification afin que cela soit fait sur l'entièreté du système. L'approche vise à développer le système comme un tout plutôt que d'opérer module par module. L'objectif de la méthodologie est d'implémenter les fonctionnalités du système sans égard à ce qu'elle doit se faire en matériel et logiciel ou une combinaison des deux. On parle alors de conception du point de vue du système [29].

La méthodologie ESL définit seulement un flux de conception qui décrit les étapes nécessaires afin de pouvoir produire un système à divers niveaux d'abstraction. Tel que décrit dans [29], il n'y a pas une seule bonne méthodologie pour tous les types de système. On parle alors de méthodologies ESL. Pour cette recherche, nous nous concentrons sur ESL appliqué au système sur puce avec une approche du haut vers le bas.

### 2.2.1 Modélisation

La première étape de la méthodologie ESL est celle de la modélisation du système. Le modèle d'un système est la description (ou l'analogie) utilisée pour aider la compréhension de quelque chose qui ne peut être observé directement. Par exemple, une liste de spécifications ou de requis. La plupart du temps, un modèle utilise l'abstraction qui peut camoufler les détails d'implémentation et ainsi mettre en avant-plan les aspects importants du modèle en les rendant plus simples à comprendre [29].

Pour se faire, la méthodologie ESL nécessite un langage permettant de spécifier le système à différents niveaux d'abstraction. Il existe plusieurs langages de modélisation tels que MATLAB M-Code, SystemC, SystemVerilog, Specification and Description Language (SDL), UML, eXtensible Markup Language (XML), BlueSpec, etc [29]. Dans le cadre de cette recherche, l'utilisation d'outils comme SystemC d'Accelera et de librairie d'abstraction telle que « Transaction Layer Model » (TLM) [13] nous a permis de décrire un système sur puce à très haut niveau de façon simple.

**SystemC**

SystemC est une librairie de conception système pour le langage C++. Elle permet, entre autres, la définition de module similaire à ceux pouvant être créés à l'aide de langage HDL tel que Verilog et VHDL.

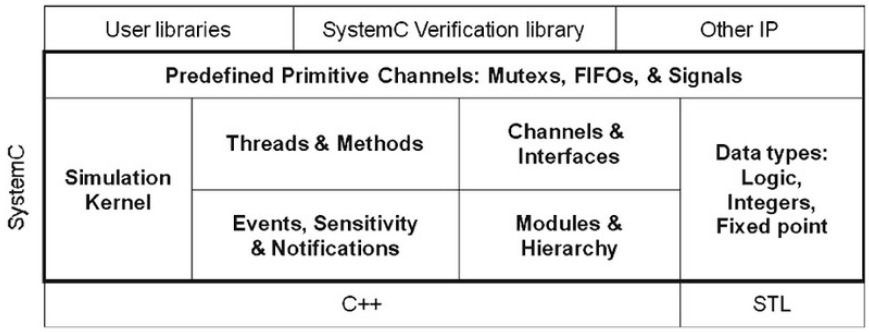



Figure 2.2 Architecture de la librairie SystemC

La figure 2.2 tirée de [11] décrit ce que la librairie apporte au concepteur système. Elle ajoute les types, interfaces, canaux, primitive de temps et autres objets nécessaires pour la description de module matériel, leur interaction et la concurrence. De plus, elle inclut un noyau permettant la simulation de ces modules.

Cette librairie permet à l'aide d'un seul langage de spécifier les modules matériels et logiciels. Elle permet l'utilisation de C++ comme langage de base faisant ainsi levier sur l'utilisation d'un grand nombre de logiciel et de librairie déjà existante. Cela apporte une meilleure réutilisation et un gain de productivité. La librairie possède aussi un sous-ensemble permettant la synthèse algorithmique. Les modules conçus strictement à l'aide de ce sous-ensemble peuvent être transformés par un outil en module RTL.

L'utilisation de SystemC, comparativement au langage HDL traditionnel, apporte aussi une meilleure abstraction et permet de définir un module à plusieurs niveaux de précision tel que le demande la méthodologie ESL. Il est possible de cette façon de définir des algorithmes et des canaux de communications de haut niveau en utilisant les types entiers de C++ ou en concevant des objets spécifiques à l'aide de classe.

**TLM**

Afin de modéliser ces abstractions, SystemC est utilisé de pair avec la méthodologie de modèles de niveau transactionnel « transaction-level model » (TLM). La méthodologie TLM consiste à séparer la représentation de l'algorithme ou le fonctionnement d'un module de ses communications. Les communications sont définies par des canaux alors que la demande de transactions est effectuée par l'appel de fonctions sur les interfaces représentant ces canaux. De cette façon le détail des communications (signaux, synchronisme, etc.) est caché par cette abstraction. Les détails des communications peuvent ainsi être définis plus tard dans la conception du système [13]. Cela permet donc l'implémentation des niveaux d'abstraction mentionnés par la méthodologie ESL. En effet, la représentation de chaque partie du système peut évoluer indépendamment, c'est-à-dire que le module peut être du niveau RTL alors que les communications peuvent être du niveau transactionnel. Cette approche permet aussi d'accélérer la simulation du système. Afin d'implémenter ces niveaux d'abstractions, la méthodologie TLM définit quatre modèles de programmation [37].

— **Untimed** : La conception d'un modèle « untime », aussi appelé « untimed functionnal » (UTF), inclut peu ou pas de notion de temps. La synchronisation se fait de façon explicite à l'aide du mécanisme d'évènements « event » et des fonctions « wait » et « notify ». Ce modèle est utile pour l'approche fonctionnelle où l'on conçoit le système sans se préoccuper des contraintes de temps. Cela permet de valider que le système respecte les spécifications fonctionnelles. Ce modèle n'offre aucune distinction entre matériels et logiciel. L'architecture n'est pas considérée dans ce modèle.

— **Loosly timed** : Ce modèle ajoute deux points de chronométrage/synchronisation au modèle de simulation. Un à l'envoi d'une transaction et l'autre à la réponse à celle-ci. De plus, la simulation utilise un logiciel d'ordonnancement permettant la synchronisation à un delta de temps paramétrable (1 ns, ms, s, etc.). Avec l'ajout de ces fonctionnalités, cette approche permet de modéliser des horloges et interruptions. Ceci est suffisant pour exécuter un système d'exploitation et exécuter des tâches sur celui-ci. Cela permet donc une première ébauche d'une plateforme virtuelle pour concevoir et exécuter le logiciel. Ce modèle permet aussi aux tâches SystemC, tout en respectant les détails de temps, de s'exécuter plus rapidement que la simulation en effectuant le traitement jusqu'à la rencontre d'un point de synchronisation de la simulation (quanta).

— **Approximated timed** : Le modèle « Approximated timed » aussi appelé « Timed

functionnal » apporte une plus grande granularité de temps que le modèle « loosly timed ». Les communications ont quatre points de chronométrage. Il s'agit d'un modèle comportemental avec concept de temps. Les modules et communications sont raffinés par l'ajout de latence à leur implémentation à l'aide de la commande SystemC « wait ». Par exemple, l'ajout d'annotations temporelles pour modéliser le temps d'exécution d'un corps de boucle « while » ou encore pour modéliser le temps de communication vers un autre module, etc. Tout comme le modèle précédent, une simulation de ce modèle opère avec des quantas de temps paramétrable (1ns, ms, s, etc.).

— **Cycle-timed** : Le modèle au précis au cycle « cycle-timed » ou « cycle accurate » correspond au RTL équivalent d'une implémentation à l'aide d'un langage HDL. Les modules sont définis à l'aide de chemin de données et de contrôle à l'aide de machine à états et les communications sont implémentées par des signaux élémentaires.

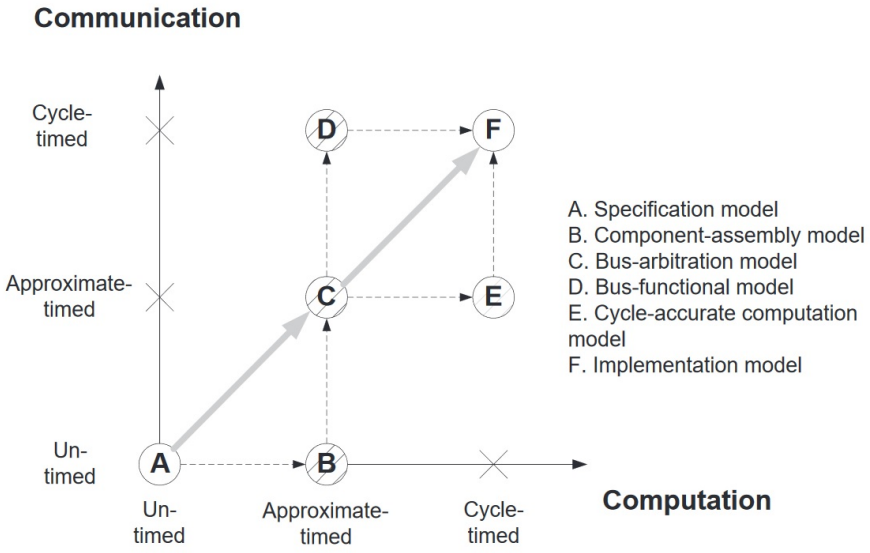



Figure 2.3 Modèle TLM [13]

Grâce au découplage des communications et des modules, chacun peut être raffiné indépendamment tel qu'illustré à la figure 2.3. Le modèle « loosly timed » a été ajouté pour la version 2 de TLM et n'est pas affiché sur cette figure. On peut voir que l'axe des communications peut évoluer indépendamment de l'axe de calcul et le concepteur est libre de choisir la direction pour le raffinement de ses modules.

À l'aide de la librairie SystemC et de la méthodologie TLM, nous sommes en mesure de développer un système avec plusieurs niveaux d'abstraction. Ce modèle doit être exécuté afin

d'avoir une idée des performances de celui-ci selon ces divers modèles d'abstraction et des choix architecturaux. Pour se faire, la méthodologie ESL préconise une approche à l'aide de plateforme virtuelle.

### 2.2.2 Platforme Virtuelle

Les plateformes virtuelles sont une représentation logicielle fonctionnelle d'un système matériel sous forme logicielle qui peut être basée sur une suite de processeurs (ARM, x86, PowerPC, etc.), de périphériques, de mémoires et d'autre module plus spécifique au domaine dans lequel le système opérera. Sur une plateforme virtuelle, ces éléments sont modélisés à haut niveau par des simulateurs de jeu d'instruction « Instruction Set Simulator » (ISS), des appels de fonction, des accès mémoire et des transactions de communication. Ces modèles matériels peuvent exécuter un système d'exploitation (Linux RT, uC-II, QNX, etc.) et des logiciels tout comme si cela était exécuté sur un système physique [5].

Comme discuté dans la sous-section précédente, une plateforme virtuelle peut implémenter plusieurs modèles de communication et d'architecture avec des contraintes de temps plus ou moins précises. Ceci est grandement plus rapide comparativement à l'approche de type précis au cycle et au bit près « cycle/bit accurate » associée à la simulation RTL traditionnel. La précision de ces modèles peut alors être augmentée en fonction de l'avancement du développement pour obtenir des métriques plus précises toujours en étant plus rapide [5].

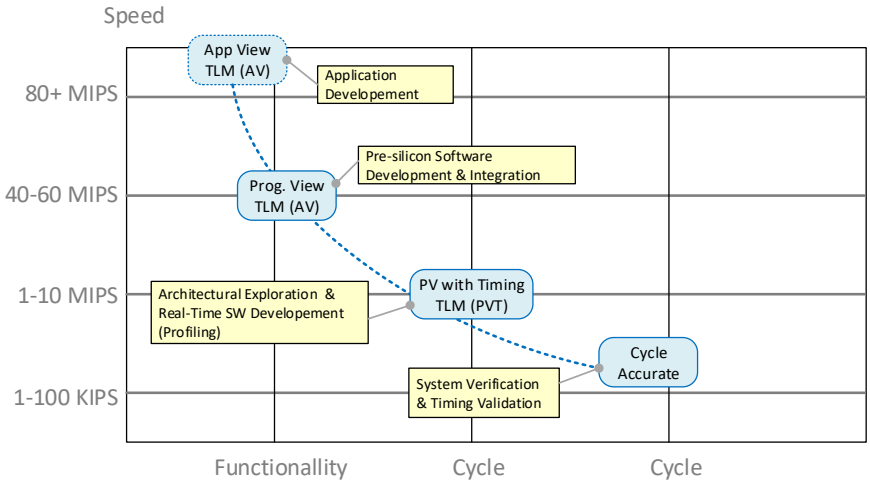



Figure 2.4 Vitesse de simulation en fonction de l'abstraction [41]

L'idée principale derrière l'utilisation de plateforme virtuelle pour le domaine des systèmes embarqués sur puce c'est d'accélérer le développement en permettant l'accès à un modèle fonctionnel du matériel afin d'entreprendre le développement des logiciels plus rapidement [5]. Cette approche offre aussi plusieurs autres possibilités. Une possibilité importante parmi ces dernières est la possibilité de tester en continu le développement du système. La plateforme virtuelle permet d'intégrer rapidement au niveau fonctionnel le matériel et le logiciel en simultané. Les changements incrémentaux peuvent alors être testés sur l'entièreté du système de façon continuel plutôt qu'à la fin d'un cycle de développement. [19]. Une autre fonctionnalité importante pour ce projet est l'extensibilité de la plateforme. Afin d'inclure le développement matériel, certaines plateformes possèdent une possibilité d'extension pour « cosimuler » avec les blocs matériels spécialement développés pour ce système. Il existe plusieurs plateformes virtuelles développées capables d'offrir cette fonctionnalité, dont SIMICS de Wind River et OVPSim de Imperas. Ces plateformes offrent une suite d'outils commerciaux permettant l'extension et la configuration d'ISS et des différents périphériques modélisés [28, 4]. Il existe aussi des plateformes de virtualisation à source ouvert comme QEMU[9] et GEM5[10]. Ces logiciels ont l'avantage d'être gratuits et peuvent aussi être étendus afin de répondre aux besoins des architectes système.

Dans le cas de ce mémoire, nous utilisons le logiciel SpaceStudio qui inclue la plateforme QEMU avec l'extension TLMu [18]. Cette extension permet la communication par protocole TLM avec QEMU. Cette approche permet d'utiliser QEMU afin de fournir un ISS pour le coeur ARM. Plus précisément le Cortex-A9 à coeurs double dans notre cas. Afin de définir des modules matériels spécialisés préalablement développés ou en développement pour ce système, on peut utiliser la modélisation SystemC ainsi que le noyau de simulation de celui-ci pour effectuer une cosimulation.

Suite au partitionnement, le code source C/C++ et SystemC peut être transformé en modèle RTL à l'aide d'une plateforme EDA permettant la synthèse haut niveau (HLS).

### 2.2.3 HLS

La synthèse comportementale (High Level Synthesis) aussi appelée « ESL synthesis » permet de créer un système à haut niveau à l'aide de code C/C++/SystemC et produire la logique numérique en format RTL (Verilog/VHDL). De ces sources RTL il est possible de créer une implémentation « gate-level » (porte logique de base) à l'aide de la synthèse logique.

La synthèse à haut niveau est un domaine de recherche très actif et est issue du besoin de palier à l'augmentation de la complexité des systèmes numériques. En permettant de définir un système à un plus haut niveau d'abstraction à l'aide d'un langage comme SystemC, la synthèse permet une plus grande productivité, mais aussi une meilleure optimisation des ressources et des performances de la microarchitecture du module synthétisé. Il existe plusieurs outils HSL notamment des outils commerciaux tels que Vivado HLS de Xilinx [21], CatapultC de Calypto Design Systems [12], Cynthesizer de Forte [30], A++ d'Altera [3] et des outils de recherche tels que DWARV de l'université de Technologie de Delft [35].

Il est aussi connu que les outils HLS d'aujourd'hui procurent de la contrôlabilité (à travers les pragma). Plus l'ingénieur sait comment contrôler, plus optimal est le résultat. Selon les algorithmes analysés et des synthétiseurs utilisés [36], l'approche HLS peut présentement offrir des performances comparables ou moins bonnes que la conception manuelle. En effet, l'article [17] démontre que le module créer à l'aide d'un outil HLS est 3.4 fois plus lent comparativement à une conception manuelle. Par contre, d'autre article comme [6] apporte une nuance en démontrant des performances comparables à un RTL manuel dépendamment de l'approche utilisée par les développeurs. Ceci dit, le bénéfice de la synthèse à haut niveau, pour l'approche ESL, est le gain en temps de productivité étant donné qu'un logiciel HLS peut donner un module RTL en quelques minutes, voir heures, plutôt qu'en semaines, voir mois, si le module doit être fait manuellement.

La synthèse comportementale ne comble que la partie calcul d'un bloc. En ce qui concerne les communications avec celui-ci, elles doivent être établies avant la synthèse en format « pin accurate ». Il y a donc une nécessité de travaillée à un plus haut niveau afin d'abstraire ces communications et permettre une définition d'interface automatiquement [32].

## 2.3 Logiciels de développement au niveau système

La méthodologie ESL tente d'introduire le développement en maintenant une vue globale du système. Comme discuté dans les sections précédentes, il existe différents outils permettant de développer les diverses étapes de la méthodologie ESL, mais qu'advient-il des environnements de développement les mettant en communs dans un même flot de conception ? Les environnements de développement intégré tel que Vivado de Xilinx [21] et Quartus d'Altera [2] ne représentent eux aussi qu'une partie de la méthodologie. Ils aident à l'implémentation finale du système sur FPGA en prenant en entrée les partitionnements préétablis lors de la conception de l'architecture.

Il existe pour le moment quelques logiciels permettant le design d'un système complet à l'aide de la méthodologie ESL au niveau système. Un de ceux-ci est le logiciel « CoFluent Studio » d'Intel, initialement développé par l'École polytechnique de l'Université de Nantes. Cet outil s'occupe de la modélisation du système à l'aide d'une approche d'ingénierie dirigée par les modèles. Une approche similaire au UML du côté logiciel. Les modèles développés par CoFluent peuvent être transformés automatiquement en module SystemC pouvant ensuite être simulé et testé. Pour valider plus en profondeur l'architecture, l'application est compatible avec les plateformes Virtuel SIMICS de Wind River [16]. Par contre, ce logiciel ne permet pas de continuer dans le flot de conception et produire une implémentation du système. Nous avons une modélisation système qui doit être par la suite répliqué manuellement en logiciel et RTL afin d'en produire une implémentation. Cette traduction a pour désavantage de possiblement introduire des défectuosités ou des modifications à la spécification initiale. De plus, l'environnement de test devra être reproduit partiellement ou en totalité.

Il existe un projet plus intégré à la méthodologie nommé SystemCoDesigner [24]. Ce logiciel prend en entrée une modélisation du système sous la forme « actor-oriented model » développée en SystemMOC. SystemMOC est une extension de SystemC pour y appliquer la modélisation du comportement des communications. Chaque acteur du système défini en SysteMoC est par la suite transformé en logiciel ou matériel à l'aide d'un outil HLS. La particularité de ce logiciel est l'exploration architecturale automatique. Afin de l'utiliser, l'information de la synthèse est réintroduite dans le logiciel. L'utilisateur élabore les choix architecturaux appelés « architecture template » et les relie au système à l'aide d'une représentation formelle de ces liens avec les requis logiciels et les communications. L'application fournit une suite d'outils afin de développer ces modèles formels. L'algorithme tente par la suite de trouver les meilleures solutions en suivant les compromis décrits dans le modèle formel [24]. Une des principales difficultés de cette application est la définition de ses modèles formels des communications et ressources. Les règles à appliquer nécessitent beaucoup de temps et sont loin d'être intuitif, et ce même si SystemCoDesigner fournit un éditeur pour créer ces modèles plus facilement.

Il y a donc pour le moment un nombre restreint de logiciels qui permet de produire la grande vision de la conception au niveau système et implémenter la fonctionnalité sans égard si cela doit se faire en matériel et logiciel ou une combinaison des deux. Le logiciel SpaceStudio, tout comme SystemCoDesigner tente de remplir ce vide, en apportant un environnement intégré

de développement logiciel/matériel en utilisant des modèles intuitifs incluant une plateforme virtuelle, divers niveaux d'abstraction et la possibilité de l'implémentation du système à l'aide d'outil EDA tierce.

### 2.3.1 SpaceStudio

L'application SpaceStudio comble ce vide à l'aide d'une vue à haut niveau sur le développement de système embarqué sur puce [34, 33, 32, 8]. L'application permet de manipuler à haut niveau les diverses applications et algorithmes du système afin d'explorer leur implémentation à l'aide de différentes ressources (processeur général, coprocesseur, protocole, etc.). Ceci de façon rapide à l'aide de sa plateforme virtuelle et d'une représentation par module des diverses tâches du système. Il utilise une approche similaire aux processus de khan [32, 8] et chaque module communique par lien bien défini modélisé par des fifo, bus, DMA, mémoire partagée et registre. SpaceStudio offre des facilités afin que chaque module puisse être exécuté comme logiciel ou coprocesseur matériel facilement. Ainsi le concepteur est capable d'obtenir une estimation des performances du système [8]. Ceci permet d'expérimenter plusieurs solutions en fonctions des requis pour réduire rapidement l'espace de recherche d'un système logiciel/matériel.

Afin de modéliser un système sous SpaceStudio, on doit implémenter les blocs fonctionnels de celui-ci sous forme de modules. L'application contient la libraire SpaceLib qui abstrait tous les interconnexions et périphériques de la plateforme. Elle implémente un premier niveau d'abstraction qui permet la simulation et vérification au niveau fonctionnel du système. La caractéristique principale de cette configuration est l'utilisation de modèle abstrait pour les communications. Celles-ci se font à l'aide soit d'un modèle de bus de type UTF (untimed functional) ou TF (timed functional). À ce niveau, les simulations sont rapides et peu précises. Elles ne donneront pas nécessairement un bon estimé des performances, mais elle permet rapidement de vérifier que notre implémentation est fonctionnelle. Suite à cela, le logiciel implémente une seconde phase d'où il est possible de raffiner en passant à une plateforme virtuelle plus précise (modélisation des délais, ISS plus précise, etc.) et effectuer une recherche en profondeur. C'est lors de cette configuration que l'on spécifie une plateforme matérielle du système (CPU, DSP, Bus, mémoire, DMA, etc.) et les protocoles de communication de type AT (Approximately Timed). À cette étape, il est possible de faire l'exploration architecturale et produire plusieurs configurations différentes afin de réduire l'espace de solution.

Les modules matériels sont exécutés à l'aide du simulateur de SystemC alors que le logiciel est exécuté par un ISS. Dans le cas d'un ISS pour Microblaze, son implémentation est faite en SystemC [8]. Cet ISS offre aussi la possibilité d'exécuter un RTOS grâce à une translation des appels système [15]. D'autre part, l'ISS du processeur ARM a pour noyau l'émulateur de QEMU qui communique par lien TLM au simulateur SystemC à l'aide de TLMu préalablement présenté. Cette approche permet d'exécuter différentes partitions logiciels/matérielles composées de tâches SystemC, soit sous format logiciel ou matériel, reliées entre eux par des interconnections TLMu.

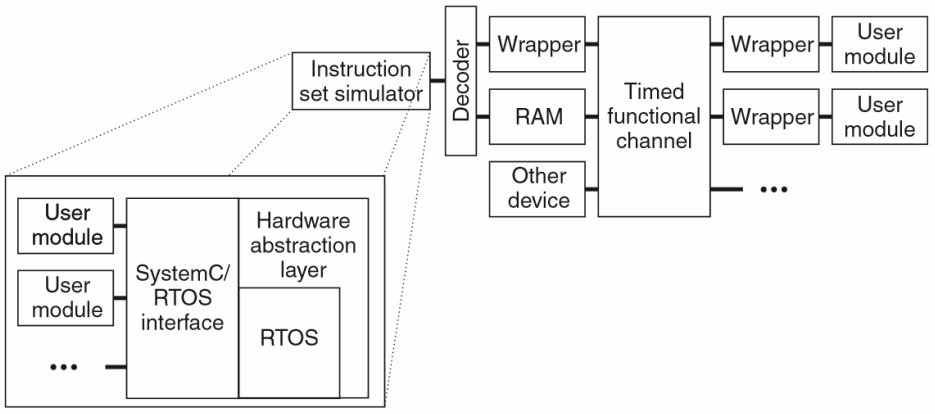



Figure 2.5 SpaceStudio RTOS sur ISS SystemC [15]

Finalement, les solutions élaborées à l'aide de la plateforme peuvent être implémentées et validées sur FPGA à l'aide de la Synthèse algorithmique (HLS) d'un CAD ou EDA tierce. Lors de l'utilisation des outils de Xilinx, il est aussi possible de spécifier une liste de « pragma » ou spécification pour optimisation. SpaceStudio vérifie le design du système selon les contraintes physiques du FPGA ciblé, traduit toutes les composantes du système vers des IP de la libraire utilisée (Xilinx ISE, Xilinx EDA), fait les interconnexions, instancie toute autre composante nécessaire, génère les déclarations d'entités pour chaque module utilisateur, associe les fichiers logiciels au processeur approprié, ajuste les options de compilation et joint les librairies nécessaires et ajoute un OS au besoin.

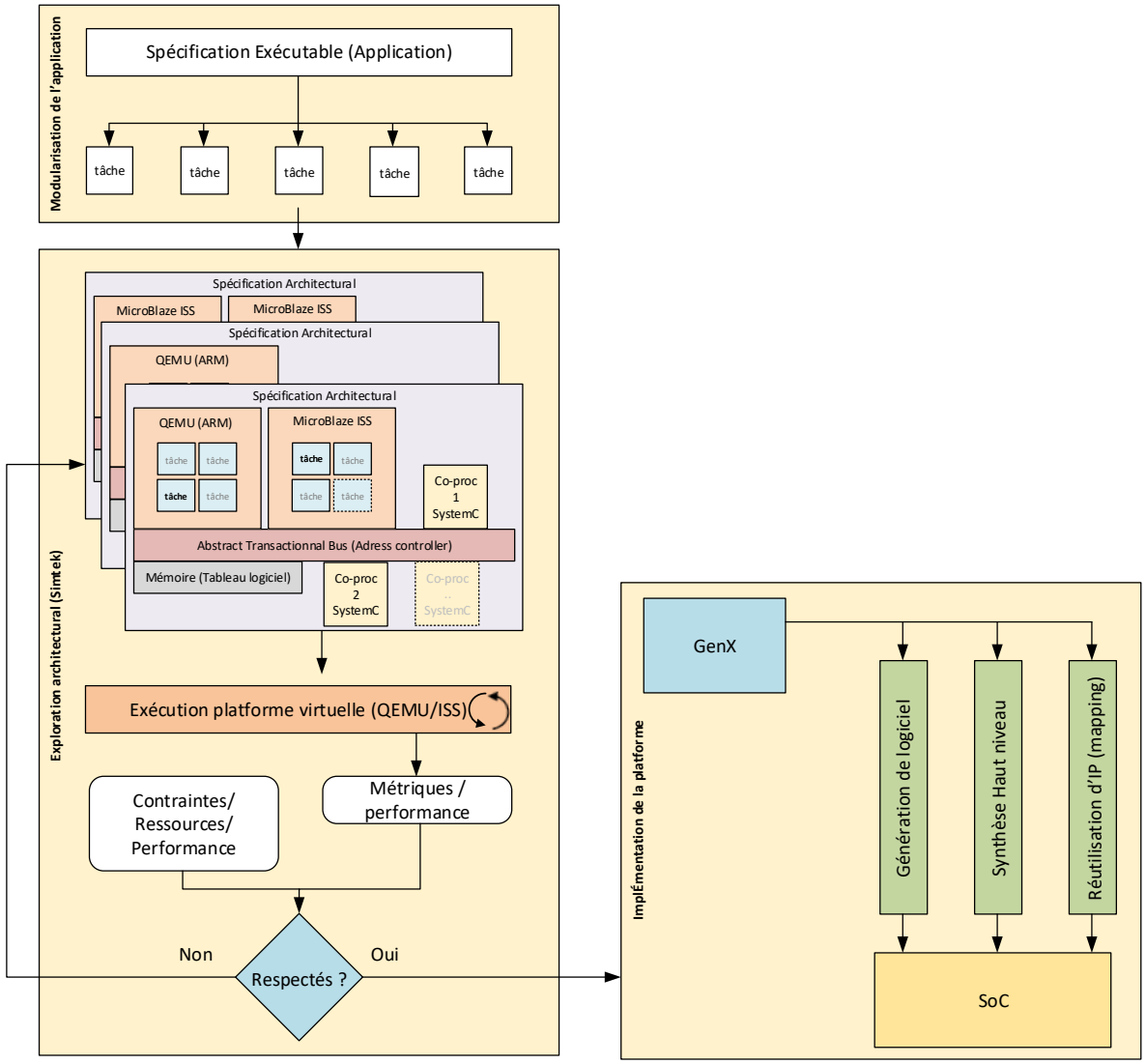



Figure 2.6 Flot de la platforme SpaceStudio

L'approche de haut niveau de SpaceStudio force par contre la séparation d'un système en modules fonctionnels. Cela a pour effet d'influencer la solution finale avant même de connaître le profil de performance de celle-ci. En effet, l'ajout de communication entre deux modules logiciel ou matériel augmente les délais comparativement à un seul module fonctionnel. Le système évalué n'est alors par le même que celui du début, car nous y avons ajouté de nouveaux délais. En pratique, la séparation doit se faire seulement aux endroits jugés propices à l'exploration matérielle suite à un profilage du modèle fonctionnel. Pour ce faire, nous

regardons la possibilité de développer itérativement les modules sous SpaceStudio à l'aide d'une spécification exécutable sous forme d'un système logiciel déjà existant.

## 2.4 Standard H.264

La norme H.264 a été définie par l'« International Telecommunications Union » (ITU), l'« International Standards Organisation » (ISO) et MPEG (Moving Picture Experts Group) comme successeur à la norme MPEG4. La norme conçue en 2003 a pour but d'offrir une efficacité de codage accrue et une robustesse aux environnements réseau. La norme riche de plus de 700 pages définit les différents algorithmes nécessaires afin de produire un décodeur. L'augmentation de la compression et de la qualité vient avec un coût en temps de calcul.

La figure 2.7 démontre sous forme de schéma bloc le flot traditionnel d'un encodeur sous la norme H.264. La section inférieure du schéma illustre le processus d'encodage alors que la section supérieure illustre la boucle de décodage et de reconstruction de l'image de référence pour l'encodage. L'encodeur utilise toujours en référence les images encodées/décodées afin de reproduire le processus d'un décodeur. L'utilisation des images décodées ayant subi une distorsion comme référence assure que l'erreur des prédictions à l'encodage ne divergera pas comparativement à un décodeur n'ayant pas l'image originale.

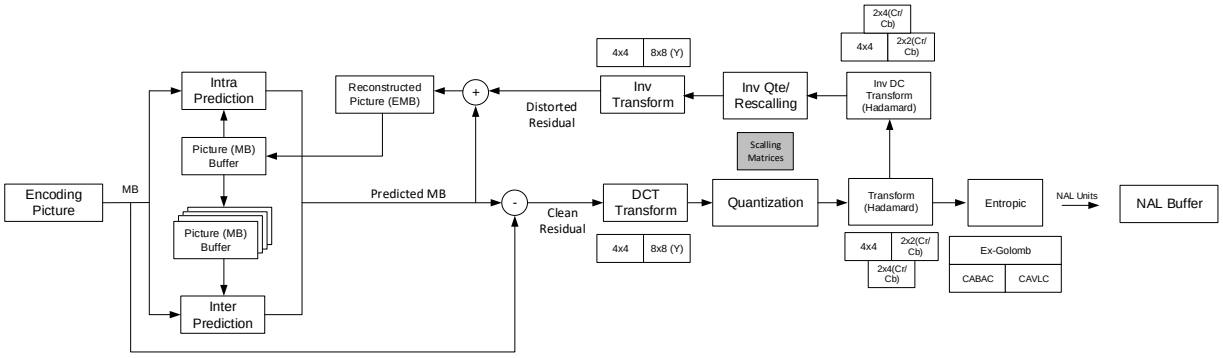



Figure 2.7 Schéma fonctionnel des étapes d'encodage et de décodage afin de traiter une trame

Dans cette section il sera question de survoler les diverses étapes encadrées par la norme et de détailler au besoin les algorithmes pertinents à notre recherche.

### 2.4.1 Division de l'image

On doit tout d'abord détailler le format des données en entrée. L'image à encoder doit être dans le format YCbCr. C'est-à-dire un canal de luminance Y (intensité lumineuse) et deux canaux de couleur Cb ( Chrominance bleue) et Cr (Chrominance rouge). L'isolation de la luminance dans un seul canal permet d'effectuer des changements sur l'image sans affecter significativement les variations d'intensité auxquelles l'œil humain est plus sensible. Ces trois canaux sont par la suite échantillonnés indépendamment.

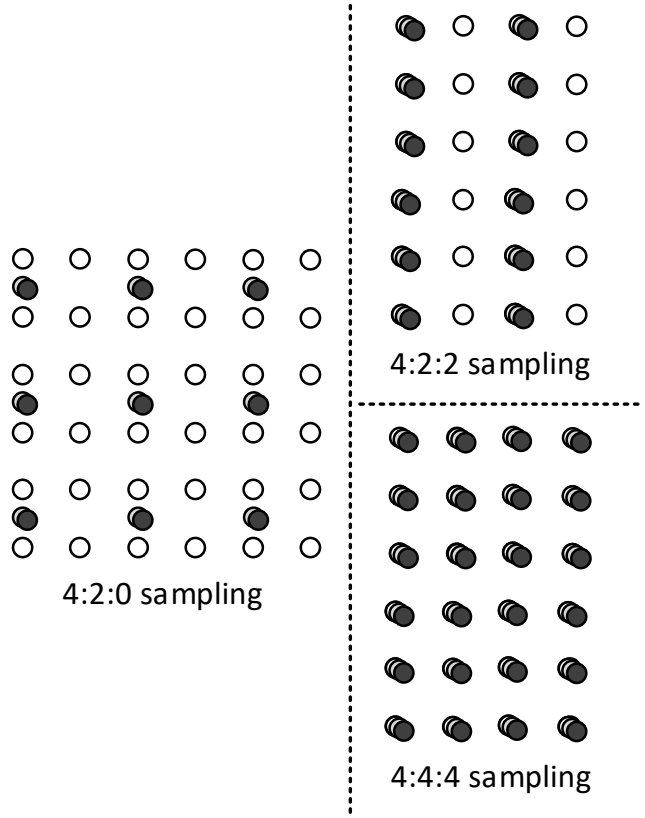



Figure 2.8 Échantillonnage supporté par la norme H.264 [39]

Le but est de réduire l'information en entrée afin de traiter moins de données. Comme mentionné, l'être humain est plus sensible aux intensités lumineuses qu'aux couleurs, il est donc

possible de réduire l'espace utilisé par celles-ci à l'aide d'un sous échantillonnage qui réduit la résolution des chrominances. La norme supporte les échantillonnages suivants :

— **4 :2 :0** : Pour quatre pixels de luminance (Y), la résolution horizontale est alors réduite de moitié.

— **4 :2 :2** : Pour quatre pixels de luminance (Y), la résolution verticale est alors réduite de moitié.

— **4 :4 :4** : Toutes les composantes de l'image sont conservées. Il s'agit d'un échantillonnage sans perte.

Une fois échantillonnée, l'image à traiter est séparée en « slice » et macrobloc (MB). Une « slice » est une portion de l'image pouvant être décodée indépendamment des autres. Une image peut avoir une ou plusieurs « slice » ayant chacune des options d'encodage différentes. À l'intérieur de celle-ci, l'image est séparée à nouveau en blocs de 16 par 16 pixels nommés macrobloc. Le macrobloc est la plus petite division de l'image. Il représente l'entrée attendue pour la plupart des algorithmes qui compose le standard H.264. La figure 2.9 illustre une configuration possible des « slice » composées de macroblocs.

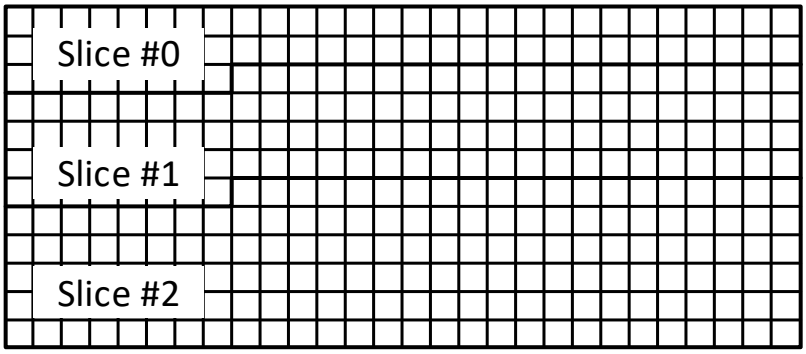



Figure 2.9 Séparation de l'image en slices et macroblocs [50]

### 2.4.2 Prédiction

Les gains offerts par la norme H.264 proviennent principalement de la performance de ses méthodes de prédictions. La prédiction sous H.264 fonctionne comme suit. Pour chaque macrobloc l'encodeur créer une prédiction de ce macrobloc en utilisant les données précédemment encodées. Cette prédiction est ensuite soustraite au macrobloc pour obtenir le résiduel.

L'efficacité et la précision du processus de prédiction ont un impact significatif sur l'efficacité de la compression. Plus la prédiction est précise, moins le résiduel contiendra de données et cela augmente l'efficacité de l'encodeur. C'est sans surprise qu'il s'agit de la zone la plus lourde d'un encodeur, car augmenter la précision de la prédiction demande une plus grande recherche et une augmentation de la complexité des algorithmes.

Donc afin d'avoir le plus petit résiduel on se doit de trouver la meilleure prédiction. Par contre, les délais engendrés par la recherche de celle-ci doivent être raisonnables en fonction des besoins de l'application de l'encodeur. Par exemple, pour produire un flot encodé en temps réel, chaque image doit être traitée rapidement pour produire le nombre d'images par seconde requis pour conserver une fluidité. Par contre, pour l'utilisation hors ligne comme les films Blu-Ray, l'encodeur peut prendre plus de temps pour trouver la meilleure prédiction.

Les algorithmes de prédiction de la norme H.264 peuvent être séparés en deux types. Prédiction spatiale (intra image) et prédiction temporelle (inter image).

### Prédiction spatiale

La prédiction spatiale tente d'exploiter la redondance d'information dans une image. L'idée est de produire à l'aide des macroblocs déjà encodés de l'image une interpolation produisant un macrobloc similaire à celui qui doit être encodé. La figure 2.10 illustre quelle partie de l'image peut agir à titre de référence.

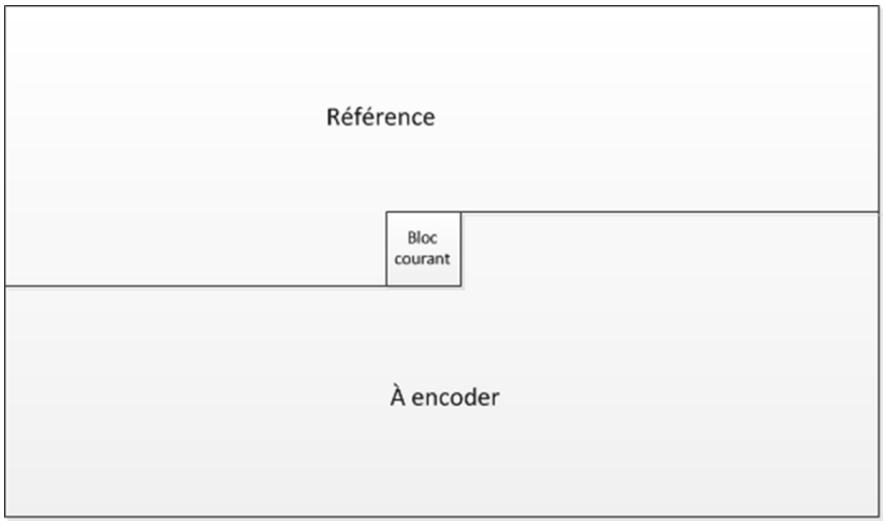



Figure 2.10 Partage des macroblocs de référence lors du processus de prédiction spatial

Afin de produire ce macrobloc à partir de ceux déjà encoder, la norme décrit neuf interpolations possibles. Celles-ci sont illustrées par la figure 2.11. Le produit de chaque interpolation est alors comparé au macrobloc à encoder à l'aide d'une somme de différences (Sum of Absolute Differences SAD). Plus la somme est grande, plus les données diffèrent. La plus petite somme est alors équivalente à la meilleure prédiction spatiale.

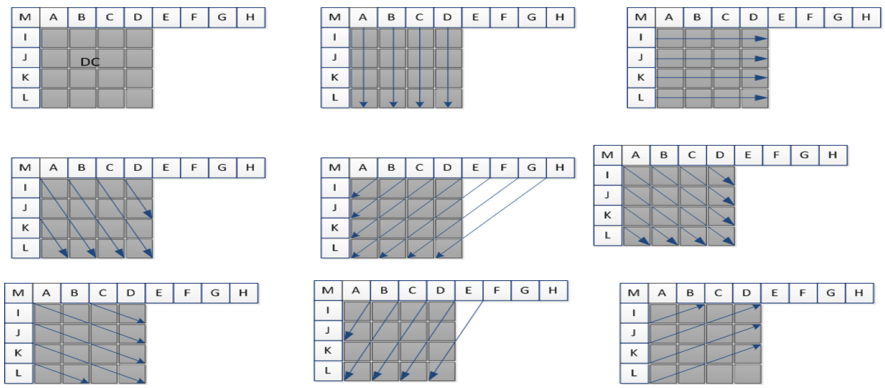



Figure 2.11 Méthodes d'interpolation spatiale possible sous la norme H.264

### Prédiction temporelle

La prédiction temporelle tente d'exploiter la redondance d'information dans le temps. C'est-à-dire que l'algorithme utilise les images passées et futures afin de trouver une référence qui minimise le résiduel. L'algorithme nomme ces images encodées de type P (passé) et de type B (bidirectionnel, passé et futur) illustré par la figure 2.12.

L'algorithme recherche dans une fenêtre parmi une suite d'images un macrobloc qui serait proche du macrobloc à encoder. Les références potentielles sont par la suite analysées au macrobloc à encoder toujours à l'aide de somme des différences. La plus petite somme est équivalente à la meilleure prédiction spatiale. Comme la recherche se fait sur une fenêtre plus grande que le macrobloc, la référence peut avoir une position différente dans l'image. Simplement penser à une suite d'image d'une personne qui se déplace. Ce déplacement est alors encodé sous forme de vecteur de mouvement. Ce déplacement est aussi illustré par la figure 2.12.

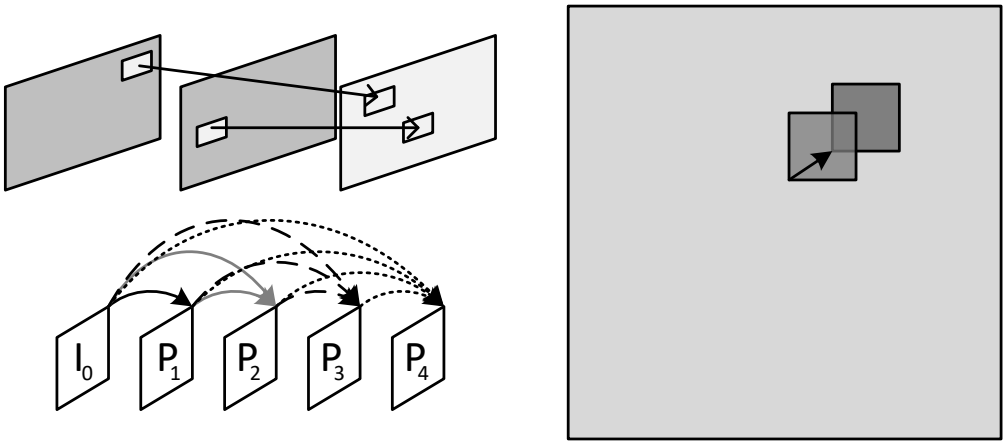



Figure 2.12 Prédiction temporelle et vecteur de déplacement [39]

L'algorithme de recherche supporte aussi le déplacement d'un demi et quart de pixel illustré par la figure 2.13. Ceci peut permettre un meilleur agencement afin de réduire le résiduel au prix d'un temps de calcul plus long. La prédiction aux sous pixel produit une nouvelle référence à l'aide de l'interpolation.

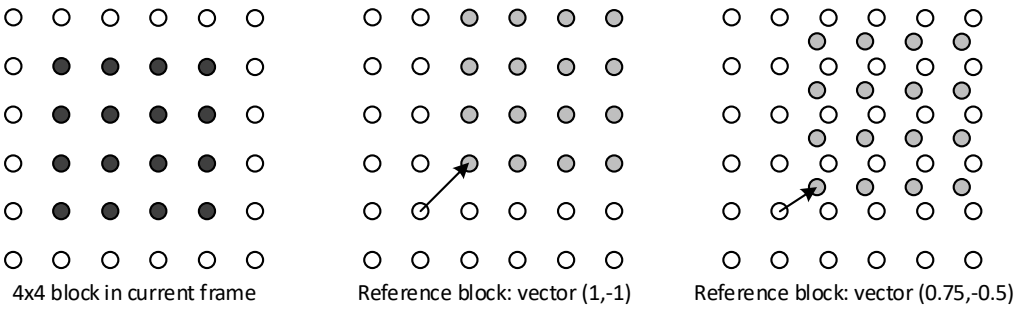



Figure 2.13 Prédiction au sous pixel [39]

Il est aussi possible d'utiliser la prédiction pondérer avec la prédiction temporelle. La pondération permet de mieux prédire un bloc lorsqu'il y a un effet de transition dans la suite d'images. Si les deux blocs sont très semblables et que seulement la luminosité est différente,

il est possible d'encoder un facteur qui pondère le bloc de référence pour le rendre similaire à celui que l'on tente d'estimer.

La prédiction temporelle nécessite une prédiction spatiale (image type I) comme trame initiale. Il peut y avoir plusieurs images de type P et B entre les images de type I. Le nombre de prédictions temporel entre les images de type I est configurable par l'algorithme.

### 2.4.3 Transformée

La transformée est un processus mathématique qui permet, entre autres, de traduire l'image dans le domaine fréquentiel. La transformée est détaillée par l'équation 2.1. Dans le domaine fréquentiel, les basses fréquences représentent l'uniformité alors que les hautes fréquences représentent les changements, discontinuité et détails de l'image. Il est alors possible d'opérer sur ces fréquences et, par exemple, d'éliminer les hautes fréquences qui représentent de l'information moins sensible à l'œil humain (filtre passe-bas). La transformer en elle-même est une opération complètement réversible et n'entraîne aucune perte sur les données. La norme définit une approximation entière de la transformée cosinus discret inverse de laquelle on peut déduire la transformée direct.

$$X_{ij} = \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} C_x C_y Y_{xy} \cos \frac{(2j+1)y\pi}{2N} \cos \frac{(2i+1)x\pi}{2N} \tag{2.1}$$

La transformée utilisé est similaire à celle utilisée dans les normes précédentes (JPEG, MPEG2, MPEG4) à la seule différence que celle utilisée sous H.264/AVC est entière. Une transformée entière permet d'éviter que l'incertitude apportée par les calculs à virgule flottante introduise une erreur dans le décodage d'une image. Étant donné que la spécification utilise les images décodées pour reconstruire les autres (images de type P et B), l'erreur pourrait alors s'additionner et produire des artefacts à l'image. De plus, cette estimation permet aussi d'accélérer les calculs.

### 2.4.4 Quantification

Comme discuté la transformée permet de départager l'information de l'image en hautes et basses fréquences. La quantification est l'opération qui introduit une perte d'information au détriment de la qualité d'image pour réduire l'entropie. La quantification est effectuée sur les résultats de la transformée. Sous H.264 elle permet l'élimination des petits détails (haute

fréquence) dont l'oeil humain est moins sensible. La norme fournit plusieurs matrices de quantification dépendant de la qualité recherchée. Plus la quantification est forte, plus les détails de l'image seront perdus et plus le résultat sera petit.

### 2.4.5 Codage entropique

Après avoir retiré les détails considérer superflus au résiduel, celui-ci passe dans un algorithme de codage entropique. Le codage entropique est une de compression sans perte. La norme H.264 utilise deux types d'encodage entropique. Soit la «Context Adaptive Variable Length Coding» (CAVLC) et «Context Adaptive Binary Arithmetic Coding» (CABAC).

L'encodage entropique est basé sur la fréquence des données (réduite par la quantification).

#### Context Adaptive Variable Length Coding (CAVLC)

Le CAVLC est basé sur le codage de Huffman utilisant les probabilités d'apparition des symboles. L'idée est d'associer aux symboles les plus fréquents le plus petit nombre de bit et vice versa. Les symboles constituant le résiduel sont encodés selon une table de références « look-up table ». Ces tables ne sont pas transmises et font partie du standard et chaque décodeur qui supporte le décodage de trame H.264 se doit d'avoir en mémoire ces tables. À l'aide de ce traitement, l'encodage final comporte moins de bits que les symboles originaux.

#### Context Adaptive Binary Arithmetic Coding (CABAC)

Le CABAC est un encodage entropique basé sur le codage arithmétique. Il utilise un algorithme probabiliste qui s'adapte en fonction des voisins du macrobloc à encoder/décoder. Ceci améliore le modèle de probabilité, car le mode de codage corrèle généralement fortement localement. Il produit un encodage de plus haute qualité au prix d'un plus grand temps de calcul. Il s'agit d'un encodage optionnel qui peut apporter une compression additionnelle de 5% à 15% comparativement à CAVLC. Il est le principal encodage utilisé pour les profils de haute qualité et est aussi le seul codage entropique utilisé dans HVEC, le successeur de l'encodage H.264.

### 2.4.6 Network Abstraction Layer (NAL)

Une NAL encapsule les « slices » afin de les transmettre soit par flot d'octets ou sous forme de fichier. Afin d'être en mesure de les transmettre, une NAL ajoute une petite quantité de données. Soit un identifiant de début et de fin de NAL et un en-tête contenant le type de

« slice ». De plus, les données des « slices » sont encapsulées dans un « raw byte sequence payload » (RBSP). Ces RBSP contiennent les résiduels et les vecteurs de mouvement pour tous les macroblocs de la « slice ».

Les NAL de type « instantaneous decoding refresh » (IDR) sont des images avec prédiction spatiale. Comme mentionnées lors de la présentation des méthodes de prédictions, les images de type P et B nécessitent une image de type I afin d'avoir une référence initiale. Les NAL de type IDR encapsulent ces images marquant ainsi une nouvelle référence pour un nouveau flot pour des images de type P et B.

Les NAL de type « sequence parameter sets » (SPS) et « picture parameter sets » (PPS) encapsulent les paramètres de configuration de l'encodage. Les NAL de type SPS s'appliquent sur une séquence vidéo alors que les NAL de type PPS s'applique sur une ou plusieurs images à l'intérieur d'une séquence vidéo. Un survol de la hiérarchie de l'encapsulation des données est illustré par la figure 2.14.

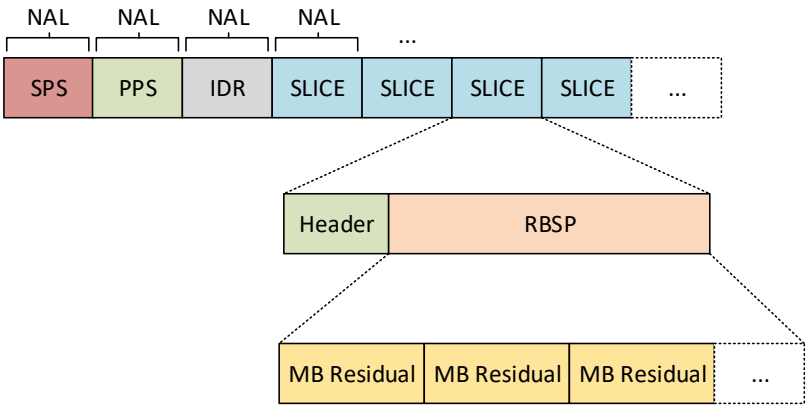



Figure 2.14 Encapsulation des données produite par un encodeur

### 2.4.7 Profils

Les profils contrôlent l'agressivité des algorithmes d'encodage et des outils utilisés par les encodeurs. Il s'agit de balancer entre la qualité, la rapidité de calcul et le débit. Les profils peuvent définir la résolution, le nombre d'images par seconde, le nombre de macroblocs par image, le nombre d'images de référence à conserver, le débit, etc. La norme définis plusieurs types de profils allant de l'application temps réel à l'encodage sans perte pour vidéo 4K.

Les profils sont des grades de conformité pour assurer la compatibilité entre les encodeurs et les décodeurs de toute sorte. Or, il n'est pas requis de suivre ces profils si nous avons des restrictions particulières tant que notre encodeur est jumelé à un décodeur ayant les mêmes options et restrictions.

## 2.5 Implémentations matérielles de la norme H.264

L'accélération d'encodage à l'aide de la norme H.264 est très importante surtout avec la récente augmentation de la consommation de contenu numérique sur les plateformes portable. Plusieurs articles ont été composés sur la conception de système sur puce pour l'accélération d'encodeur H.264 et une majeure partie se concentre sur le développement de module accélérant diverses sections d'un encodeur telles que l'IDCT, la quantification, l'encodage entropique ou l'estimation du mouvement. C'est le cas de l'article [48] qui met l'accent sur l'optimisation mémoire d'un module de prédiction aux sous pixel.

Dû à la complexité de la norme H.264, il y a un nombre limité d'implémentations complètes de la norme. De ceux-ci, nous avons sélectionné deux exemples. Soit les articles [7] et [14]. Le premier article propose la conception d'un encodeur sur FPGA alors que le second propose un système de type ASIC fait sur à l'aide de la technologie UMC 0.18um CMOS. Ces deux implémentations utilisent les langages de description matérielle comme VHDL et verilog et ne reflètent pas l'approche vue au niveau système préconisé par notre recherche.

Les auteurs de l'article [26] ont quant à eux choisi une avenue similaire à cette recherche. Ils ont utilisé une approche de haut niveau afin de développer un décodeur h.264 à l'aide d'une librairie C qu'ils ont par la suite synthétisé à l'aide d'un outil HLS. Bien qu'il ne s'agit pas d'un encodeur et que la complexité ne soit pas aussi élevée que notre approche, leurs résultats restent intéressants et s'inscrivent dans l'objectif de travailler à plus haute abstraction. Contrairement à nous, leur approche est fixe à la plateforme HLS et, comme que mentionné, l'application utilisée possède une plus petite complexité de calcul et n'utilise aucun principe de codesign ni d'exploration architectural.

L'approche au niveau système à base de spécification exécutable tel qu'illustré par les auteurs de l'article [26] et effectué par cette recherche est très récente et cela est reflété par le manque d'exemple dans la littérature.

# CHAPITRE 3 MÉTHODOLOGIE DE DÉVELOPPEMENT À L'AIDE D'UN MODEL SYSTÈME EXÉCUTABLE

Dans ce mémoire, nous chercherons à accélérer la création d'une spécification exécutable et à l'utiliser comme base du développement d'un système sur puce. Plus précisément, il est question de modifier et segmenter graduellement cette spécification en tâches logicielles et matériels afin d'atteindre les caractéristiques demandées par un client.

Bien que les méthodologies de codesign ne soient pas tout à fait récentes, l'approche, afin de produire un système sur puce à partir d'une spécification logiciel, élaborée dans ce mémoire est tout à fait nouvelle. Au moment de mettre au point celle-ci, et autant que nous sachions, aucun système pour le H264 n'avait encore été mis à terme avec cette approche dans la littérature. De plus, jusqu'à maintenant, les systèmes développés avec le logiciel SpaceStudio de manière à ce que chaque fonctionnalité soit segmentée en module et que le système soit un regroupement de ceux-ci. Un des obstacles à cette façon de faire est qu'elle peut nécessiter une grande réingénierie afin de se scinder en un groupe de modules compatible avec SpaceStudio. De plus, il n'est peut-être pas toujours nécessaire de séparer tout le système en morceau fonctionnel, car cela peut introduire des délais de communication qui ne sont pas présents dans la version originale du logiciel.

Nous voulons donc expérimenter la réutilisation du développement logiciel existant (librairie, applications logicielles, etc.) afin d'en produire un système matériel/logiciel sur puce, réduire le temps de développement et limiter la segmentation de celui-ci lorsque cela n'est pas nécessaire. Avant de présenter la méthodologie, nous devons tout d'abord discuter du système et des besoins du client.

## 3.1 Besoin de Grass Valley

Comme mentionné précédemment, Grass Valley produit une gamme de cartes de traitement vidéo. Certaines de ces cartes contiennent un sous-système qui produit des vidéos « thumbnail ». Grass Valley désire moderniser sa gamme de produits et améliorer les avantages offerts par ce sous-système.

Afin de respecter le coût de production, ce nouveau sous-système nommé « proxy » doit respecter plusieurs contraintes associées à la consommation de puissance et la surface d'utilisation (nombres de composantes). De plus, Grass Valley aimerait migrer vers un système offrant des images suivant un standard plus moderne, laissant de côté l'approche basse qualité présentement offerte.

### Résolution et qualité

Les spécifications de Grass Valley sont de sélectionner une résolution haute définition (HD), telle que 1280x720, tout en permettant également de prendre une résolution inférieure mise à l'échelle vers une résolution haute définition sans produire d'artefacts. Ces résolutions sont représentées dans la table suivante :

| 1280x720p HD | | |
|---|---|---|
| /1 | 1280 | 720 |
| /2 | 640 | 360 |
| /4 | 320 | 180 |
| /8 | 160 | 90 |
| /16 | 80 | 45 |

Nous avons déterminé que la résolution 320x180 est suffisante pour l'usage demandé par Grass Valley. En plus de simplifier l'implémentation initiale du système, une plus haute résolution aurait nécessité l'utilisation d'un profil de plus haute qualité ce qui, du même coup, aurait activé d'autres chemins de donnée dans l'encodeur et requis une plus grande bande passante.

### Bande passante

La bande passante représente la quantité de données produite en sortie par l'encodeur pour la compression des trames vidéo en entrée. La quantité de données produite d'un encodeur est fonction de l'agressivité de l'encodage. Plus l'encodeur compresse les trames vidéo, moins la bande passante sera grande et plus la qualité sera basse. L'efficacité de l'encodeur influence aussi la bande passante de même que la qualité de l'encodage.

Comme mentionné, le système « proxy » à développer sera un composant parmi plusieurs autres sur les cartes de Grass Valley. Il est donc nécessaire de limiter la bande passante utilisée. Il nous a été spécifié d'utiliser 250-300 Kbyte/sec avec un maximum de 1 Mbyte/sec.

Donc si l'on produit 30 images par seconde en sortie, ces 30 images devront comptabiliser au plus 1 mégaoctet au total.

#### Taux de rafraichissement

Le taux de rafraîchissement est le nombre d'images produites en sortie du décodeur à l'intérieur d'une seconde. Le nombre d'images par seconde à atteindre est de 8 à 15. Il s'agit d'une limite inférieure. Si le système développé est capable de produire plus d'images tout en respectant les autres requis, cela est aussi acceptable.

#### Architecture

L'encodeur ne doit traiter que l'image. Aucun flot sonore n'est pris en compte. Grass Valley préfère un système sur puce incluant un processeur général présynthétisé « softcore ». Ce dernier est un processeur pouvant être synthétisé sur de la logique programmable (FPGA). De plus, ceci est requis afin de produire une architecture évolutive telle que proposée initialement à Grass Valley.

#### Coût

Le coût final du système se calcule par deux métriques. La première métrique est la puissance nécessaire à laquelle Grass Valley alloue un budget de 1 watt. Le « proxy » se doit de limiter sa consommation de puissance, car sa fonctionnalité est une valeur ajoutée aux cartes de Grass Valley. De plus, il ne doit pas influencer l'état des composantes principales.

La seconde métrique est l'espace (surface/pièces). Présentement, leur système « thumbnail » leur coûte 5$ à produire. Ils aimeraient conserver ce budget pour le « proxy ». Par contre, ils sont ouverts à une légère augmentation, étant donné les coûts de développement d'un nouveau système de plus haute qualité.

### 3.2 Approche initiale

Initialement, l'implémentation du système pour Grass Valley était le coeur de cette maîtrise. Nous nous sommes dirigés vers une analyse de l'encodeur H.264 pour une implémentation RTL que l'on pourrait appeler traditionnel. Pour ce faire, nous avons récupéré la spécification officielle pour en modéliser le flot de contrôle et de données. Comme démontré lors du chapitre précédent, la norme est séparée en plusieurs groupes fonctionnels tels que les deux

modèles de prédiction, la transformée, la quantification, etc. Nous avons donc commencé avec la modélisation de chacun de ces groupes sous forme de module fonctionnel tenant compte en entrée les différentes options d'encodage offertes par la norme.

Après quelques mois de recherche et de modélisation, il était devenu évident que de développer un encodeur H.264 dans son entièreté à l'aide seulement de la spécification ne serait pas possible sur une période de deux ans. Également, il était devenu évident que la charge de développement et de test pour chacun de ces modules était au-delà de la charge possible pour un seul ingénieur. Nous avions donc besoin d'une approche pouvant produire un système sur puce rapidement. C'est à ce moment que nous avons décidé d'explorer l'utilisation du code « legacy » avec l'approche codesign proposée par SpaceStudio. Le code servant ici de spécification exécutable.

## 3.3 Développement niveau système à partir d'une spécification exécutable

Une spécification exécutable est une application implémentant les fonctionnalités du système à développer. Cette application sert à assurer que les spécifications énoncées concordent avec les besoins clients, mais aussi de modèle de référence pour valider l'exactitude de l'implémentation du circuit numérique.

Une spécification exécutable du système est habituellement exprimée en langage de haut niveau tel que Java, C/C++ ou SystemC. Par exemple, une application de compression, cryptage, décodage/encodage, traitement de signal, etc. Toute application logicielle se veut une spécification exécutable d'un système en soi. Si ce système doit être accéléré ou porté sur un SoC l'application agit à titre de modèle de référence.

Nous avons donc exploré la possibilité d'utiliser la spécification de la norme H.264 à l'intérieur de SpaceStudio pour le développement même du système « proxy » dans le but d'épargner en temps de développement (figure 3.1) et livrer plus rapidement le résultat à Grass Valley.

## 3.4 Méthodologie de développement expérimentale à l'aide de SpaceStudio

Comme mentionné lors de la présentation de l'outil SpaceStudio, l'approche à haut niveau de ce logiciel demande de mettre sous forme de modules fonctionnels (threads) le code d'entrée qui lui est généralement séquentiel. Cela a pour effet d'influencer la solution finale avant

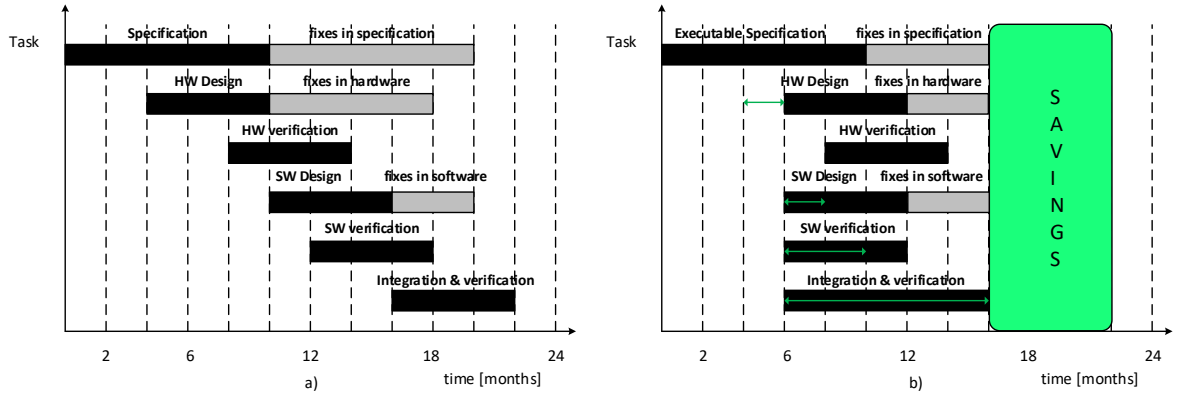



Figure 3.1 Graphique (a) représente le flot standard, Graphique (b) représente le flot ESL à l'aide d'une spécification exécutable [43]

même de connaître le profil de performance de celle-ci par l'ajout de communication entre ces divers modules. Ceci produit des délais de communication comparativement à un seul code séquentiel (i.e. module fonctionnel). Ces nouveaux délais changent le profil de performance du système.

Ce que l'on veut avec cette méthodologie est de faire la séparation seulement aux endroits jugés propices à l'exploration matérielle suite à un profilage du modèle. On veut donc une méthodologie de développement codesign où l'ont créé les modules itérativement sous SpaceStudio à l'aide d'une spécification exécutable. La figure 3.2 illustre le concept.

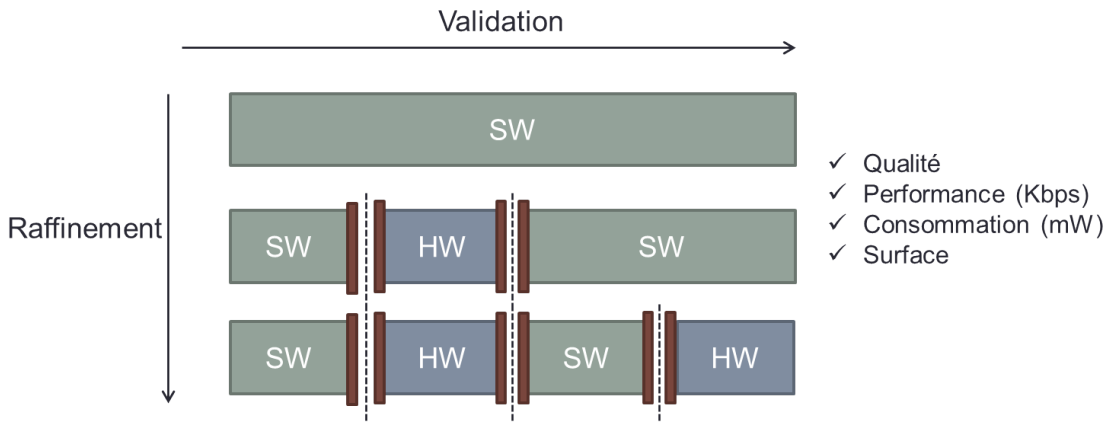



Figure 3.2 Flot de modularisation expérimental

Dans cette section il sera question de présenter en détail les diverses étapes empruntées pour l'élaboration de la méthodologie au niveau système ainsi que le développement du système « proxy » de Grass Valley.

### 3.4.1 Création du modèle fonctionnel

La première étape de cette méthodologie consiste en la conception de l'application qui agira comme spécification exécutable.

#### Librairie OpenH264

Afin de développer le modèle fonctionnel, il nous a été suggéré par Grass Valley de nous basé sur la librairie OpenH264 développée par CISCO, car ils étaient déjà familiers avec celle-ci.

CISCO offre gratuitement le code source dans le but de populariser l'encodeur H.264 pour l'api WebRTC. Cet api permet l'utilisation de communication temps réel aux fureteurs web et système mobile. La librairie OpenH264 est d'ailleurs utilisée par le fureteur Firefox depuis la version 33. En rendant leur librairie publique, CISCO a voulu offrir une implémentation commune afin d'assurer une interopérabilité entre les différents systèmes implémentant la norme. De plus, ceci permet aussi d'améliorer la qualité de l'encodeur à l'aide de la communauté d'utilisateurs et de développeurs.

Cette librairie, ayant principalement pour cible les plateformes portables, a été spécialement conçue pour les systèmes embarqués. De ce fait, le code de celle-ci peut être exécuté sur

processeur de type Intel x86 avec SSE, ARMv7 et AArch64 avec NEON ou toute autre architecture n'ayant pas d'opérations SIMD en utilisant les fonctions C/C++ alternatives.

La librairie est présentement contrainte à l'utilisation d'un profil de base hybride principalement dû à sa zone d'application (plateforme mobile). Ceci veut dire qu'elle supporte toutes les fonctionnalités pour rencontrer les restrictions de base, mais pas assez pour supporter un profile de plus hautes qualités. Concrètement, la librairie supporte seulement les trames I et P (aucune biprédiction B), l'encodage entropique CAVLC et CABAC et l'échantillonnage sous la forme YUV 4 :2 :0 planaire en entrée. La simplicité de ce profil aidera à l'implémentation initiale de la méthodologie en limitant la complexité du système.

La librairie est disponible sous deux formats. Soit en version binaire et source. La version binaire peut être liée à la compilation à une application afin que les frais de licence reliée à l'utilisation des algorithmes MPEG LA soient assurés par CISCO. Si les développeurs choisissent d'utiliser les sources de la librairie, ils doivent eux-mêmes assurer le paiement des royautés. Nous avons opté pour l'utilisation des sources, car nous avons besoin de sources afin d'être en mesure de modifier certaines sections pour y développer les modules matériels. Le code source est publié avec la licence BSD simplifiée.

**Configuration**

La librairie prend en entrée une suite de paramètre afin de configurer les divers algorithmes d'encodage utilisés. Il est possible de spécifier les dimensions en entrée et sortie des images, le taux d'images par seconde, la bande passante maximale, etc. Le profil d'encodage est alors inféré par les options activées et leurs valeurs. Afin de simplifier la logique de contrôle, nous avons fixé les paramètres d'encodage afin de nous concentrer sur un chemin précis de l'encodeur. Nous avons ajusté une suite de paramètres afin de respecter les requis de Grass Valley. Le tableau 3.1 affiche les principaux paramètres utilisés pour l'encodeur de référence. L'annexe A offre la liste complète des paramètres utilisés pour le développement du système.

Une fois la sélection des configurations faite, leur valeur ont été reproduite sous forme de valeurs constantes de compilation dans un fichier configuration du projet nommée « codec_app_def.h ». Ces valeurs sont donc prises en dure lors de la compilation. Le fichier de configuration a été fait de manière à ce qu'une grande partie des paramètres dynamique soit calculée à la compilation en fonction d'un sous ensemble de paramètres principaux. Par exemple, les paramètres suivants sont énoncés au début du fichier :

Tableau 3.1 Paramètre d'encodage du «proxy»

| Paramètre | Entrée | Sortie | Autre |
|---|---|---|---|
| Hauteur de l'image | 180 | 180 | - |
| Largeur de l'image | 320 | 320 | - |
| Taux d'image par seconde | 25 | 30 | - |
| Bande passante cible | - | - | 1000Kbps (1Mbps) |
| Codage entropique | - | - | CAVLC |
| Utilisation de processus parallèles | - | - | Désactivé |

```
#define SOC_ENCODER_INPUT_IMAGE_HEIGHT  180
#define SOC_ENCODER_INPUT_IMAGE_WIDTH   320
#define SOC_ENCODER_OUTPUT_IMAGE_WIDTH  320
#define SOC_ENCODER_OUTPUT_IMAGE_HEIGHT 180
```

Listing 3.1 Paramètre définie

De ces paramètres nous pouvons calculer la taille des trames à l'entrée. C'est ce paramètre qui est utilisé pour déclarer les tampons de l'application.

```
#define SOC_ENCODER_INPUT_FRAME_SIZE ((SOC_ENCODER_INPUT_IMAGE_HEIGHT *
   SOC_ENCODER_INPUT_IMAGE_WIDTH * 3)  >> 1)
```

Listing 3.2 Paramètre inféré à la compilation

L'encodeur « proxy » est développé selon des spécifications précises dictées par Grass Valley, mais il est possible de créer un nouvel encodeur avec différente spécification en modifiant ce fichier, compiler et synthétiser le SoC. Ce fichier a été développé de façon à offrir un peu de latitude si l'on veut modifier les paramètres pour produire un encodeur ayant un profil différent.

Le choix de fixer à la compilation la configuration de l'encodeur a été fait pour éviter le chargement de fichiers de configuration afin que le démarrage du système ne dépende pas d'un système de fichier. De plus, cela simplifie le code de l'encodeur et nous permet de transformer certains tampons mémoires variables en fonction des paramètres en tampon statique pour les modules matériels, car les allocations dynamiques ne sont pas supportées pour la définition de module matériel. Il y a aussi un gain à l'allocation statique pour les tâches logicielles étant

donné que l'allocation dynamique apporte un surcoût à l'exécution.

Restreindre le choix de la configuration aide aussi à réduire les chemins de contrôle si une option est statique. Il n'est alors pas nécessaire de porter tous les chemins de contrôles influencés par les différentes valeurs de celle-ci. Il s'agit d'une méthode pour réduire le code exécutable pour un système embarqué.

La libraire OpenH264 est complexe et a nécessité une longue analyse afin de bien déterminer son fonctionnement. L'analyse faite lors de la création de la spécification exécutable a été bénéfique pour la suite du développement. Le fonctionnement de la librairie sera détaillé au besoin pour justifier les choix architecturaux du SoC.

**Développement de l'encodeur**

Le développement initial de la spécification exécutable a été fait sous Windows 10 avec l'environnement de développement Visual Studio 2013. Afin de construire rapidement un encodeur à partir de la librairie OpenH264, nous avons utilisé l'application test offerte en exemple par la librairie. Les fichiers projet pour Visual Studio sont aussi fournis. À cette application nous avons effectué plusieurs modifications afin d'avoir un encodeur pour les besoins du développement en codesign.

Nous avons tout d'abord retiré la lecture des configurations par ligne de commande ou fichier de configuration comme mentionné dans la section précédente. Le fichier de configuration « codec_app_def.h » offre des structures constantes qui peuvent être assignées aux structures de l'encodeur pour assigner automatiquement la configuration définie à la compilation.

Par la suite, nous avons fusionné la librairie au programme de test afin de retirer les interfaces de librairie statique et lier le code directement à l'application, car certaines fonctions de la librairie sont susceptibles d'être accélérées par un module matériel. Cette librairie doit être en mesure de communiquer alors avec un module SpaceStudio ce qui ne peut pas se faire à l'intérieur d'un objet lié statiquement ou dynamiquement (librairie dll, lib ,etc.). Nous avons donc retiré toutes les classes d'interface telle que ISVCEncoder pour accéder à un objet encodeur et IWelsVP pour accéder aux algorithmes de traitement vidéo. Tous les fichiers de la librairie ont par la suite été ajoutés au projet de base afin d'en faire un exécutable monolithique.

La librairie contient différents chemins de traitement SIMD dépendant de la plateforme ciblée. Par exemple, certains algorithmes sont accélérés en assembleur en fonction des unités SIMD disponible (Neon, SSE, etc.). Ces optimisations sont désactivées pour notre implémentation, car la plateforme cible demandée par Grass Valley ne contient aucune unité SIMD. Nous avons aussi désactivé l'utilisation du parallélisme de la librairie pour simplifier l'implémentation initiale. Cette librairie possède des abstractions pour l'utilisation de parallélisme sous Windows et les systèmes d'exploitation du type POSIX utilisant « pthread ». Dans notre cas, nous prévoyons utiliser un processeur « softcore » ayant potentiellement un système d'exploitation autre tel que MicroC. La dépendance aux interfaces parallèles de Windows ou POSIX pose problème.

Suite à ces modifications, nous avons étoffé l'application pour qu'elle soit en mesure d'encoder des vidéos sous le format YUV et en produire un fichier h.264. Un fichier h.264 est un regroupement de NAL qui peut être mis à l'intérieur d'un conteneur vidéo tel que MP4 afin qu'il soit lisible par un lecteur vidéo comme VLC. Pour ce faire, nous avons construit un script sous Window utilisant le logiciel ffmpeg pour encapsuler le vidéo dans un conteneur vidéo de type MP4.

Nous avons par la suite développé un fichier « makefile » afin de compiler notre application sous Linux. Pour ce faire, nous avons utilisé une machine virtuelle « Virtual Box » exécutant le compilateur gcc 4.8.4 sous Mint Linux 17. Cette machine virtuelle a accès au même dépôt de code que la version Windows. Le développement de l'application sous Linux nous permet d'avoir accès à une grande liste d'application d'analyse de code qui n'est pas disponible sous Windows. De plus, une version Linux nous permettra de migrer plus facilement vers la plateforme virtuelle de SpaceStudio. Ce point sera approfondi à la prochaine section. La migration vers Linux s'est faite rapidement étant donné que la librairie est multiplateforme.

À ce point nous avons une application qui encode des trames de type YUV en NAL s'exécutant sur processeur x86 sous Windows et Linux. Afin d'être en mesure de développer un système sur puce à partir de celle-ci, on doit migrer l'application vers un module logiciel s'exécutant sur la plateforme virtuelle de SpaceStudio.

### 3.4.2 Migration de l'application vers SpaceStudio : Modèle fonctionnel

L'application a par la suite été portée vers le logiciel SpaceStudio sous format d'un module logiciel nommé *MainEncoderTask*. L'application test sert de boucle de traitement principale au module et cette boucle appelle les fonctions d'encodage provenant de la librairie que nous avons ajoutées comme fichier source externe. La migration de l'application s'est faite sous le mode ELIX de SpaceStudio. Celui-ci produit un exécutable n'ayant aucun concept de temps. L'exécutable est l'équivalent d'une application logiciel sous Windows, mais avec l'utilisation de la librairie de communication de SpaceStudio nommée SpaceLib. De ce fait, nous avons eu à modifier toutes les zones de code utilisé pour la journalisation (logging) afin que les sorties (printf, etc.) utilisent la fonction SpacePrint de SpaceStudio. L'utilisation de cette fonction permet de rediriger correctement le flot de messages dépendant du modèle du module (logiciel ou matériel).

### 3.4.3 Migration de l'application vers SpaceStudio : Modèle de simulation

La première itération du système se fera avec une architecture ARM. Bien que nous cherchons une implémentation complètement sans processeur physique, l'utilisation d'un processeur ARM nous permet d'utiliser Linux comme OS pour le module principal étant donné que celui-ci fonctionne déjà sous celui-ci, ceci ayant été validé lors de la conception de la spécification exécutable. Cela nous permet d'alléger la migration du code et nous permettra de nous concentrer sur la création des modules matériels plus rapidement. Le but est de transformer progressivement l'application en SoC et continuellement valider les changements. Lors d'une seconde itération, le code pourra être traduit complètement vers un système d'exploitation temps réel tel que MicroC (uC-II).

L'étape suivante a été de spécifier les mémoires disponibles dans l'architecture. Sous la plateforme de SpaceStudio, ces zones sont spécifiées par des périphériques mémoires de type « device ». Pour accéder à ces zones, les modules doivent utiliser les fonctions « DeviceRead » et « DeviceWrite » de la librairie SpaceLib. Or, le module d'encodage principal **MainEncoderTask** fonctionne avec une « heap », car l'encodeur utilise plusieurs allocations dynamiques pour ses structures de données et les trames. Pour la migration initiale, nous avons opté pour le statu quo et laissé les allocations mémoires telles quelles. La plateforme virtuelle ARM avec un système d'exploitation invité de type Linux permet l'utilisation des fonctions d'allocation dynamique (malloc, new, delete, etc.). La plateforme ARM utilisée est basée sur l'architecture Zynq-7000 de Xilinx et possède une mémoire locale de type DDR3.

Cette plateforme opère alors comme un ordinateur traditionnel et le système d'exploitation Linux à accès aux 512 Mb de mémoire sur laquelle réside sa « heap ».

Nous avons tout de même ajouté des périphériques mémoires à l'architecture de base. L'architecture initiale possède deux périphériques mémoires de type BRAM (bit Ram). Une première de 300 Mo contient les trames en entrée sous format YUV. La seconde BRAM possède une grandeur de 8Mo afin de contenir la sortie de l'encodeur soit les NAL. Une BRAM représente une unité de mémoire traditionnel ayant un temps d'accès plus court étant donné que celle-ci réside sur la fabrique FPGA.

La première mémoire est utilisée par un nouveau module que nous avons nommé **InputAdapter**. Ce module s'occupe d'isoler la réception de l'encodage d'une trame. Ce module permet l'abstraction de cette réception des données. Grass Valley n'a pas communiqué le protocole d'entrée pour ce système ce qui motive aussi l'inclusion de cet adaptateur. Le module **InputAdapter** charge en mémoire partagé une trame vidéo à encoder à partir de la BRAM qui sert de tampon de réception. Les modules logiciels sous SpaceStudio sont définis comme « thread » d'un même processus sous Linux. Elle partage donc par défaut le même espace virtuel. Afin de communiquer entre deux modules logiciel, il est possible de faire suivre simplement un pointeur vers une zone mémoire.

La seconde mémoire est utilisée par le module **MainEncoderTask**. Il se charge de mettre les trames encodées sous forme de NAL dans cette BRAM. Celle-ci agit comme tampon de sortie. À l'aide d'un périphérique virtuel de type « filesink » branché à cette mémoire, SpaceStudio nous permet de sortir les valeurs de cette mémoire vers notre système de fichier à la fin de l'exécution de la simulation. Ceci nous permet de visualiser les trames encodées à l'aide d'un lecteur média.

La plateforme Zynq incluant le processeur ARM, la DDR et l'ajout des deux BRAM nous offrent une configuration architecturale qui nous servira de base pour la suite du développement de notre système sur puce. L'architecture décrite est illustrée à la figure 3.3.

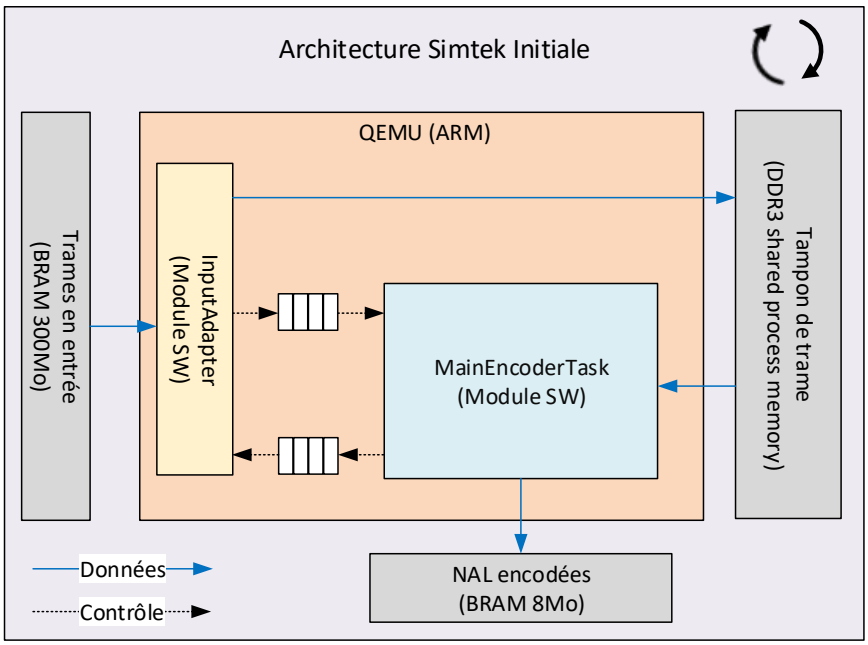



Figure 3.3 Architecture Simtek initiale

### 3.4.4 Implémentation du processus de validation

Le processus de validation se doit d'identifier toute modification engendrant une différence à la qualité de l'encodage. Il ne faut pas seulement se fier à la simple visualisation du fichier vidéo produit. On se doit de produire une méthodologie qui évalue la qualité de façon objective.

Les NAL produites par l'encodeur incluent un en-tête suivi du résiduel de macroblocs. Dans notre cas, chaque NAL encapsule une SLICE et celle-ci encapsule les résiduels complets pour une trame. Chaque NAL équivaut à l'encodage d'une trame. Si le mécanisme d'encodage est perturbé entre deux encodages, les résiduels de macrobloc seront différents. Il serait donc possible d'exécuter une somme de contrôle (checksum) sur chacune des NAL écrite en mémoire et comparer ce résultat avec celui d'une trame correctement encodée. S'il n'y a aucune différence entre les deux résultats cela sera indicatif de l'intégrité du processus d'encodage. Si le processus d'encodage a été modifié, les sommes de contrôle des NAL seront alors différentes.

La somme de contrôle utilisé pour valider l'encodage est basée sur l'algorithme « cyclic redundancy check » (CRC). Afin d'alléger la base de code du projet, nous avons recherché une implémentation simple d'un algorithme CRC. Les librairies comme Boost proposent une suite

d'algorithme de CRC, mais sont beaucoup trop complexes pour ce que nous cherchons. Nous avons opté pour une implémentation spécialisée pour les systèmes embarqués développés par Michael Barr [1]. Cette somme de contrôle est effectuée sur chaque octet d'une NAL.

La trame de référence proviendra du résultat d'encodage de notre modèle développé sous Windows. Celui-ci agira de « golden model ». Les sommes de contrôles des NAL de référence et de l'encodeur seront effectuées simultanément par le même algorithme de contrôle. À des fins pratiques, nous avons ajouté une nouvelle mémoire BRAM de 3Mo pour accueillir les NAL du « golden model » dans la plateforme virtuelle.

Pour valider l'algorithme, nous avons sélectionné trois types d'image de test. Une série d'images avec peu de mouvement afin de peu solliciter l'estimation de mouvement. Une série avec beaucoup de mouvement avec des couleurs uniforme afin de solliciter l'estimation de mouvement avec des recherches locales (itération basse). Et finalement, une série avec beaucoup de mouvement et des couleurs afin de solliciter l'estimation de mouvement à son maximum (itération maximale). Nous avons produit notre propre suite d'images afin d'obtenir les paramètres voulus en entrée (sous échantillonnage, résolution, etc.). Le but est de valider l'intégrité de l'encodage. Nous tenons pour acquis que la qualité de l'algorithme de la librairie a déjà été validée.

Dépandement de l'image sélectionnée à l'entrée, le « golden model » est lue au bon offset dans la BRAM de référence. La sélection du test est faite à la compilation. Il suffit d'activer la bonne image dans la configuration et recompiler le projet. La configuration de ces tests réside dans le fichier de configuration « codec_test_file_def.h » que l'on retrouve à l'annexe A.4. Ceci change seulement le module en entré **InoutAdapter** et le décalage de validation utilisé par **MainEncoderTask**.

Certains flots d'images solliciteront les algorithmes de prédiction temporelle telle que des images avec beaucoup de mouvement alors que d'autres solliciteront les algorithmes de prédiction spatiale telle que les images stables, dialogue entre deux personnes, etc.

Nous avons ajouté à la tâche principale des options de compilation afin d'activer ou désactivé la validation des données. Ceci nous permet de tester nos implémentations architecturales et désactiver la validation afin d'évaluer les performances de celle-ci. La figure 3.4 présente le résultat de validation effectuer sous la plateforme virtuelle. Ces informations sont affichées à

la console et écrites dans un journal d'exécution.

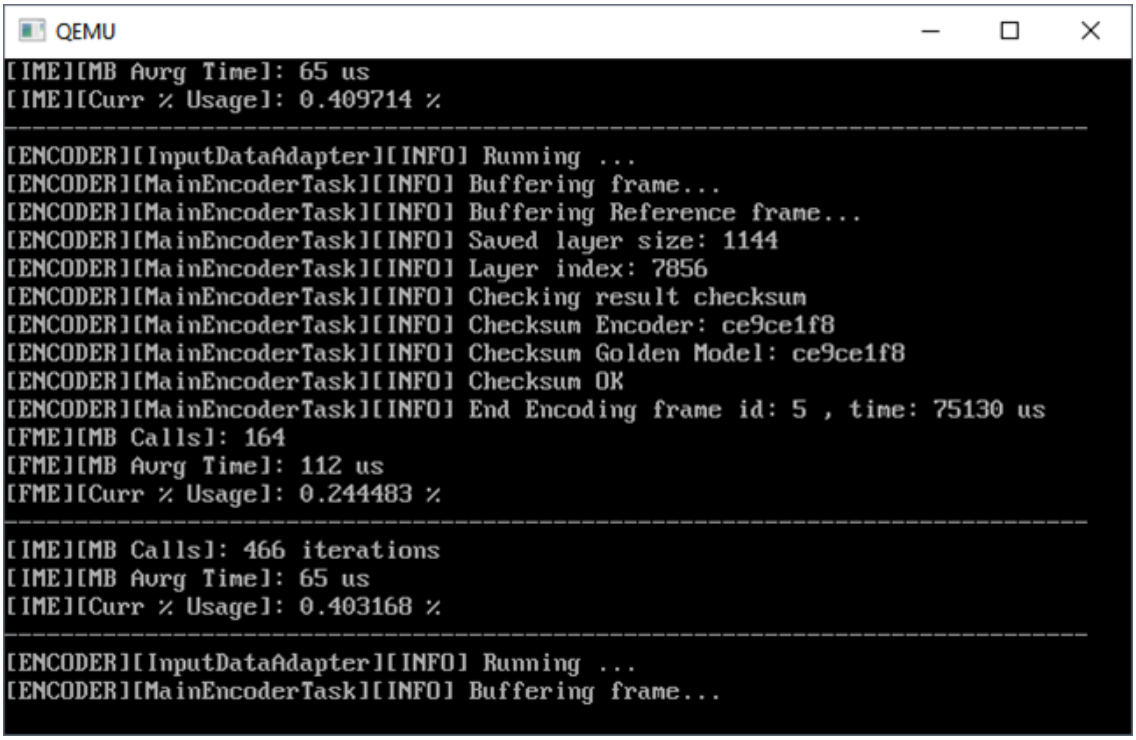



Figure 3.4 Exécution de l'encodeur et de la suite de tests sur la plateforme virtuelle de SpaceStudio

### 3.4.5 Profilage

Nous avons maintenant une application s'exécutant sur une machine virtuelle QEMU (ARM) sous SpaceStudio et ayant les facilités de validation. Nous sommes maintenant en mesure de procéder à l'évaluation des performances de l'application et découvrir les zones propices à l'accélération. Nous ne pouvons malheureusement pas évaluer les zones chaudes des fonctions à l'intérieur d'un module sous SpaceStudio. L'application possède un engin de monitorage, mais celui-ci évalue à gros grains chaque module indépendamment ainsi que tous les appels vers la SpaceLib (DeviceRead, DeviceWrite, etc.). Jusqu'à ce jour, les fonctions logiciel à l'intérieur d'un module ne sont pas évaluées. Nous devons donc nous tourner vers des logiciels externes à la plateforme virtuelle afin d'évaluer les performances des fonctions incluses dans la librairie OpenH264.

Lors de notre phase d'analyse, nous avons utilisé deux logiciels afin d'obtenir différentes métriques susceptibles de nous aider à faire un choix sur les accélérations possibles. Ces deux

logiciels ont été exécuter sur une plateforme de type x86 soit un processeur Intel Core i7 sous Linux Mint 17.

### Pareon

Le premier logiciel que nous avons utilisé se nomme Pareon de la compagnie Vector Fabric. Cet outil permet d'analyser un logiciel afin d'introduire des solutions de parallélisation. L'outil identifie les designs et les algorithmes de l'application qui sont prohibitifs à l'exécution parallèle et donne des suggestions afin de corriger celles-ci et offre des stratégies pour paralléliser le logiciel. L'outil guide le développeur à travers le code. Il permet aussi de produire des estimations de performance à l'ajout de parallélisme. Le but principal de l'outil est d'automatiser la tâche fastidieuse de l'analyse du code pour introduire du parallélisme.

Pour utiliser Pareon, l'application doit être compilée avec leur compilateur afin d'outiller le code. Par la suite, lorsque l'application est exécutée, la librairie de Pareon produit un fichier de données qui sera analysé par l'application.

La vue de l'outil offre plusieurs informations sur le code, dont le nombre d'appels, le temps d'une exécution, les délais mémoires et la couverture du code de chaque fonction et boucle. Il fournit aussi un pourcentage d'utilisation du temps processeur pour chaque appel de fonction. Nous avons extrait ces valeurs que nous avons transcrites dans le tableau 3.2. Seuls les résultats pour les fonctions principales sont affichés.

Malheureusement, la compagnie Vector Frabric a fermé ses portes en mai 2016 et l'avenir de cet outil est incertain.

### Valgrind

Le second logiciel utilisé est Valgring. Ce logiciel est un cadriciel pour implémenter des outils d'analyse, de déverminage (debugging) et de profilage. Le logiciel est gratuit et « open source » sous la licence GNU General Public License (GPL) version 2. Le logiciel vient avec une suite d'outils dont chacun peut donner des métriques différentes sur le code exécuté. Le plus connu de ces outils est « Memcheck ». Il se charge d'analyser une application C/C++ afin de découvrir toute utilisation erronée de la mémoire.

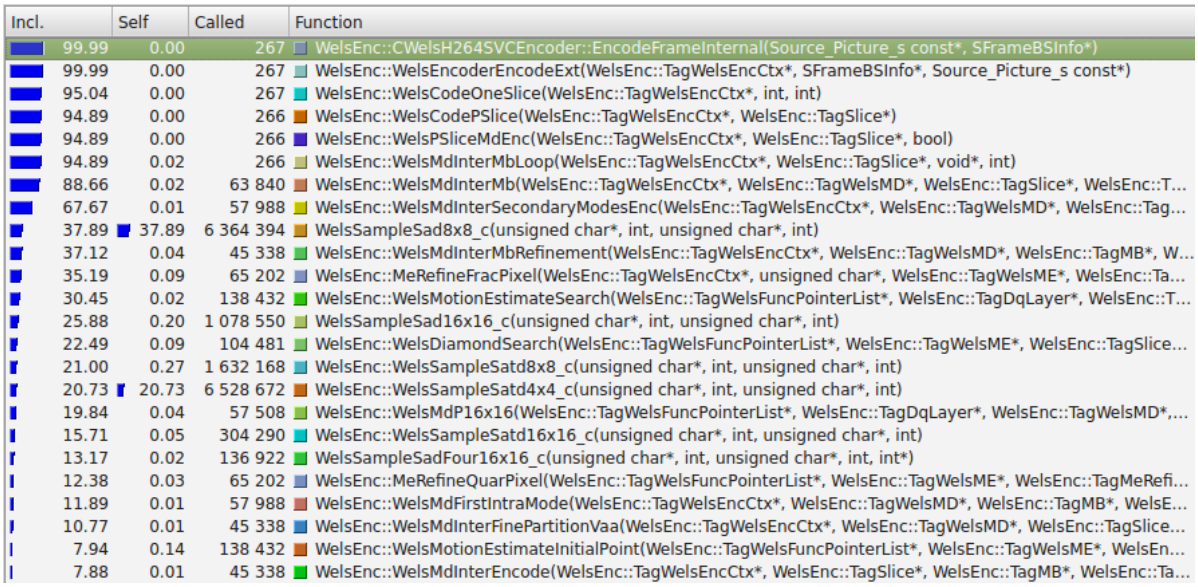
Tableau 3.2 Pourcentage d'utilisation du temps processeur selon Pareon

| Fonctions | % du temps CPU |
|---|---|
| WelsEnc : :WelsCodeOneSlice | **89%** |
| \| WelsEnc : :WelsMdInterMb | **84%** \| |
| \| \| WelsEnc : :WelsMdInterJudgePskip | **3.5%** \| \| |
| \| \| WelsEnc : :WelsMdP16x16 | **22.6%** \| \| |
| \| \| \| WelsEnc : :WelsMotionEstimationSearch | **22.4%** \| \| \| |
| \| \| \| \| WelsEnc : :WelsMotionEstimateInitialPoint | **5.5%** \| \| \| \| |
| \| \| \| \| WelsEnc : :WelsDiamondSearch | **17%** \| \| \| \| |
| \| \| \| \| \| WelsEnc : :WelsSampleSadFour16x16_c | **16.59%** \| \| \| \| |
| \| \| WelsEnc : :WelsMdInterSecondaryModeEnc | **57.88%** \| \| |
| \| \| \| WelsEnc : :WelsMdFirstIntraMode | **17.55%** \| \| \| |
| \| \| \| \| WelsEnc : :WelsMdI16x16 | **6.5%** \| \| \| \| |
| \| \| \| \| \| WelsEnc : :WelsSampleSad16x16_C | **4.19%** \| \| \| \| |
| \| \| \| \| WelsEnc : :WelsMdInterFinePartitionVaa | **3.7%** \| \| \| \| |
| \| \| \| \| \| WelsEnc : :WelsMdI4x4Fast | **3.5%** \| \| \| \| |
| \| \| \| \| WelsEnc : :WelsEncRecI16x16Y | **3.3%** \| \| \| \| |
| \| \| \| \| \| WelsEnc : :WelsDctMb | **1.28%** \| \| \| \| |
| \| \| \| \| WelsEnc : :WelsIMbChromaEncode | **2.42%** \| \| \| \| |
| \| \| \| WelsEnc : :WelsMdInterFinePartitionVaa | **5%** \| \| \| |
| \| \| \| WelsEnc : :WelsMdInterMbRefinement | **28.37%** \| \| \| |
| \| \| \| \| WelsEnc : :MeRefineFracPixel | **22.89%** \| \| \| \| |
| \| \| \| \| \| WelsEnc : :MeRefineQuarPixel | **8.7%** \| \| \| \| |
| \| \| \| WelsEnc : :WelsMdInterEncode | **7%** \| \| \| |
| \| \| \| \| welsEnc : :WelsInterMbEncode | **4.5%** \| \| \| \| |

Valgrind est un cadriciel d'analyse dynamique de binaire, c'est-à-dire que l'outil analyse le code à l'exécution de l'application. Son utilisation ne requiert pas de recompilation et n'a besoin d'aucune librairie instrumentée. Par contre, l'utilisation de Valgrin ralentit grandement l'exécution de l'application.

Ce qui nous intéresse de cet outil est sa possibilité de profilage. Dans cette suite d'outils, nous retrouvons Callgrind. Cet outil est principalement une extension de l'outil Cachegrind. Il permet de simuler les caches du processeur et ainsi prédire avec une grande précision les zones de code pouvant causer des erreurs de cache. Callgrind ajoute à cela les graphes d'appel de l'application. À l'aide de cet outil, nous sommes en mesure de visualiser les chemins utilisés dans le code ainsi que le nombre d'appels et le pourcentage d'utilisation du processeur de chaque fonction.

À la fin de l'exécution de l'application, Valgrind produit un fichier de donnée qui peut être visualisé à l'aide de l'application KCachegrind. La figure 3.5 affiche les résultats produits par Valgrind. Seuls les résultats pour les fonctions principales sont affichés.



| Incl. | Self | Called | Function |
|---|---|---|---|
| 99.99 | 0.00 | 267 | WelsEnc::CWelsH264SVCEncoder::EncodeFrameInternal(Source_Picture_s const*, SFrameBSInfo*) |
| 99.99 | 0.00 | 267 | WelsEnc::WelsEncoderEncodeExt(WelsEnc::TagWelsEncCtx*, SFrameBSInfo*, Source_Picture_s const*) |
| 95.04 | 0.00 | 267 | WelsEnc::WelsCodeOneSlice(WelsEnc::TagWelsEncCtx*, int, int) |
| 94.89 | 0.00 | 266 | WelsEnc::WelsCodePSlice(WelsEnc::TagWelsEncCtx*, WelsEnc::TagSlice*) |
| 94.89 | 0.00 | 266 | WelsEnc::WelsPSliceMdEnc(WelsEnc::TagWelsEncCtx*, WelsEnc::TagSlice*, bool) |
| 94.89 | 0.02 | 266 | WelsEnc::WelsMdInterMbLoop(WelsEnc::TagWelsEncCtx*, WelsEnc::TagSlice*, void*, int) |
| 88.66 | 0.02 | 63 840 | WelsEnc::WelsMdInterMb(WelsEnc::TagWelsEncCtx*, WelsEnc::TagWelsMD*, WelsEnc::TagSlice*, WelsEnc::T... |
| 67.67 | 0.01 | 57 988 | WelsEnc::WelsMdInterSecondaryModesEnc(WelsEnc::TagWelsEncCtx*, WelsEnc::TagWelsMD*, WelsEnc::Tag... |
| 37.89 | 37.89 | 6 364 394 | WelsSampleSad8x8_c(unsigned char*, int, unsigned char*, int) |
| 37.12 | 0.04 | 45 338 | WelsEnc::WelsMdInterMbRefinement(WelsEnc::TagWelsEncCtx*, WelsEnc::TagWelsMD*, WelsEnc::TagMB*, W... |
| 35.19 | 0.09 | 65 202 | WelsEnc::MeRefineFracPixel(WelsEnc::TagWelsEncCtx*, unsigned char*, WelsEnc::TagWelsME*, WelsEnc::Ta... |
| 30.45 | 0.02 | 138 432 | WelsEnc::WelsMotionEstimateSearch(WelsEnc::TagWelsFuncPointerList*, WelsEnc::TagDqLayer*, WelsEnc::T... |
| 25.88 | 0.20 | 1 078 550 | WelsSampleSad16x16_c(unsigned char*, int, unsigned char*, int) |
| 22.49 | 0.09 | 104 481 | WelsEnc::WelsDiamondSearch(WelsEnc::TagWelsFuncPointerList*, WelsEnc::TagWelsME*, WelsEnc::TagSlice... |
| 21.00 | 0.27 | 1 632 168 | WelsEnc::WelsSampleSatd8x8_c(unsigned char*, int, unsigned char*, int) |
| 20.73 | 20.73 | 6 528 672 | WelsEnc::WelsSampleSatd4x4_c(unsigned char*, int, unsigned char*, int) |
| 19.84 | 0.04 | 57 508 | WelsEnc::WelsMdP16x16(WelsEnc::TagWelsFuncPointerList*, WelsEnc::TagDqLayer*, WelsEnc::TagWelsMD*,... |
| 15.71 | 0.05 | 304 290 | WelsEnc::WelsSampleSatd16x16_c(unsigned char*, int, unsigned char*, int) |
| 13.17 | 0.02 | 136 922 | WelsSampleSadFour16x16_c(unsigned char*, int, unsigned char*, int, int*) |
| 12.38 | 0.03 | 65 202 | WelsEnc::MeRefineQuarPixel(WelsEnc::TagWelsFuncPointerList*, WelsEnc::TagWelsME*, WelsEnc::TagMeRefi... |
| 11.89 | 0.01 | 57 988 | WelsEnc::WelsMdFirstIntraMode(WelsEnc::TagWelsEncCtx*, WelsEnc::TagWelsMD*, WelsEnc::TagMB*, WelsE... |
| 10.77 | 0.01 | 45 338 | WelsEnc::WelsMdInterFinePartitionVaa(WelsEnc::TagWelsEncCtx*, WelsEnc::TagWelsMD*, WelsEnc::TagSlice... |
| 7.94 | 0.14 | 138 432 | WelsEnc::WelsMotionEstimateInitialPoint(WelsEnc::TagWelsFuncPointerList*, WelsEnc::TagWelsME*, WelsEn... |
| 7.88 | 0.01 | 45 338 | WelsEnc::WelsMdInterEncode(WelsEnc::TagWelsEncCtx*, WelsEnc::TagSlice*, WelsEnc::TagMB*, WelsEnc::Ta... |

Figure 3.5 Pourcentage d'utilisation du temps processeur des fonctions principales selon Valgrind

### 3.4.6 Déterminer les sections pertinentes

Une fois l'analyse de l'application effectuée on se doit de sélectionner la ou les zones propices à l'accélération. À l'aide des statistiques offertes par nos outils d'analyse, nous chercherons à déterminer les meilleures sections candidates à l'accélération et identifier les propriétés d'une telle zone.

Les deux outils d'analyse utilisés démontrent que l'application passe beaucoup de temps dans les fonctions SAD telles que **WelsSampleSad16x16_c()** et **WelsSampleSad8x8_c()**. Soit 16.59% et 28.36%. Une architecture de test incluant une version matérielle de la fonction **WelsSampleSad16x16_c()** a été développée par cinq élèves de 4iem année au BAC de la Polytechnique [22]. Les résultats de simulation et d'implémentation on permit d'identifier que cela ralentissait grandement le système à cause du taux de données à échanger entre le processeur et le coprocesseur. Ces transferts prenaient beaucoup plus de temps que l'exécution du SAD, car les données sur lesquelles doit être exécutée l'opération devaient être transférées vers le coprocesseur, celui-ci n'ayant pas accès à la mémoire centrale. La fi-

gure 3.6, que l'on retrouve dans leur rapport final, illustre ce gouffre de performance. La ligne « temps 0 » représente le délai requis uniquement pour les communications. On remarque très bien que cela est déjà beaucoup plus long que la version logiciel du SAD. Ceci démontre que lors de la création d'un coprocesseur, on ne peut se fier seulement au temps d'utilisation du processeur comme habituellement fait lors de l'optimisation par logiciel.

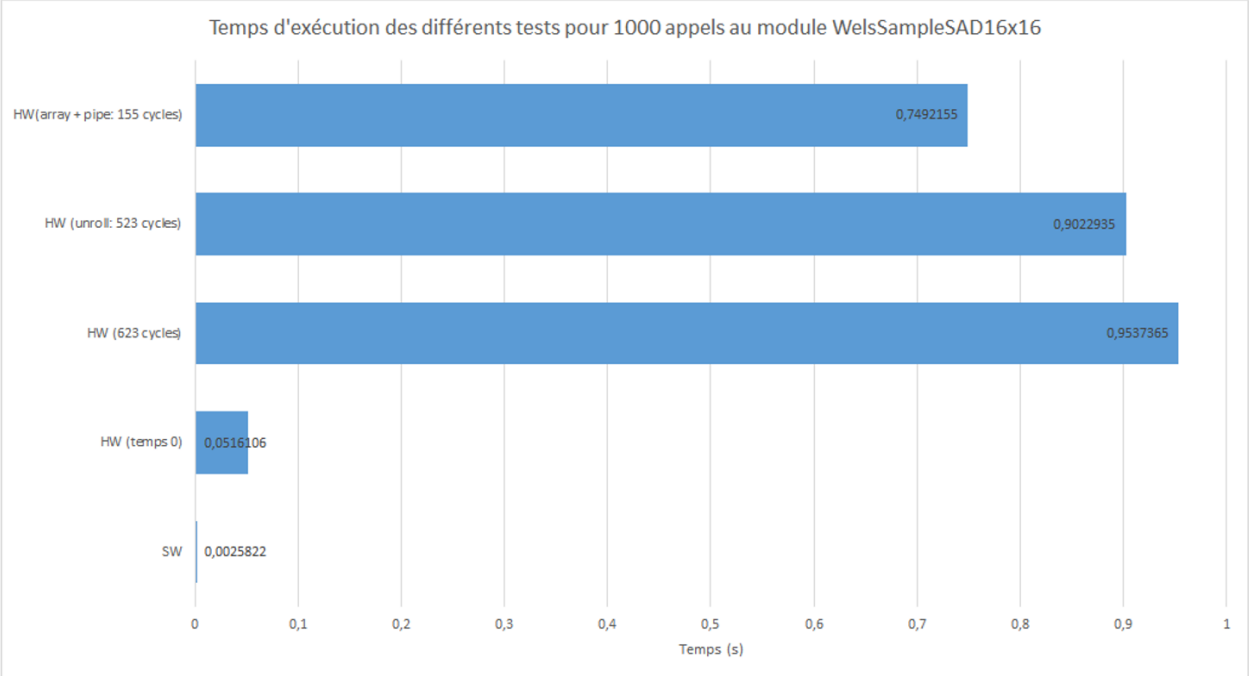



Figure 3.6 Temps d'exécution des différents tests en fonction des optimisations pour 1000 appels au module WelsSampleSad16x16

Le temps d'utilisation du processeur affiché par ces outils est cumulatif, c'est-à-dire qu'une fonction peut prendre quelques microsecondes d'exécution, mais être appelé très souvent et avoir un temps d'utilisation processeur élevé. Le pourcentage d'utilisation camoufle en fait deux métriques. Soit la fréquence d'appel et le temps d'exécution de la fonction. Pour le modèle processeur/coprocesseur sans mémoire unifiée, les fonctions candidates doivent avoir un temps d'exécution relativement haut avec un nombre d'appels relativement bas afin de pallier le surcoût des communications. Entre d'autres mots, il faut que le gain en temps de calcul soit supérieur au temps perdu en communications. Autrement, le gain obtenu par l'accélération sera absorbé par les communications.

Ceci dit, les analyses effectuées à l'aide de Valgrind et Pareon nous indiquent plusieurs fonc-

tions ayant des potentiels d'accélération. Nous nous sommes attardés à regarder celles ayant une grande quantité de calculs et d'opérations. De celles-ci, nous avons sélectionné les fonctions **welsMotionEstimateSearch()** et **welsMdInterMbRefinement()**. Ces fonctions font partie de l'algorithme d'estimation du mouvement (Motion Estimation, ME).

La fonction **welsMotionEstimateSearch()** va effectuer plus de 360 SAD lors de l'exécution du flot de recherche pour un macrobloc. La figure 3.7 illustre les opérations effectuées pour un appel. Elle comptabilise une utilisation du processeur à 30% pour le traitement de 267 trames.

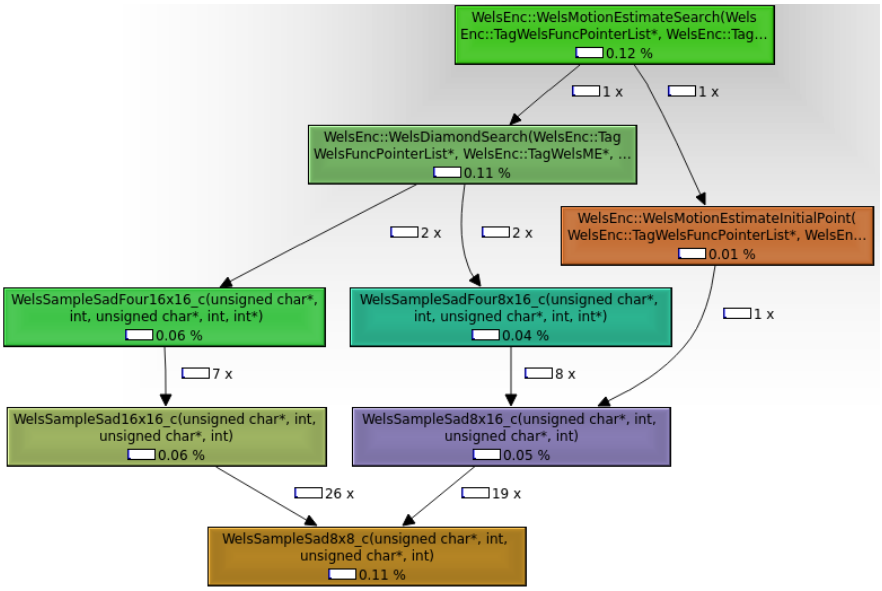



Figure 3.7 Graphe d'appels de la fonction welsMotionEstimateSearch() pour le traitement d'un macrobloc

La fonction **welsMdInterMbRefinement()** effectue plus de 1036 opérations de filtrage et 1057 SAD par macrobloc analysé. Les figures 3.8 et 3.9 illustrent les opérations effectuées pour un macrobloc. Elle comptabilise une utilisation du processeur à 37% pour le traitement de 267 trames vidéo.

Le nombre de SAD effectuer est calculé en fonction du nombre exécuté par **welsSampleSad8x8_c()**. Cette fonction produit huit SAD par appel.

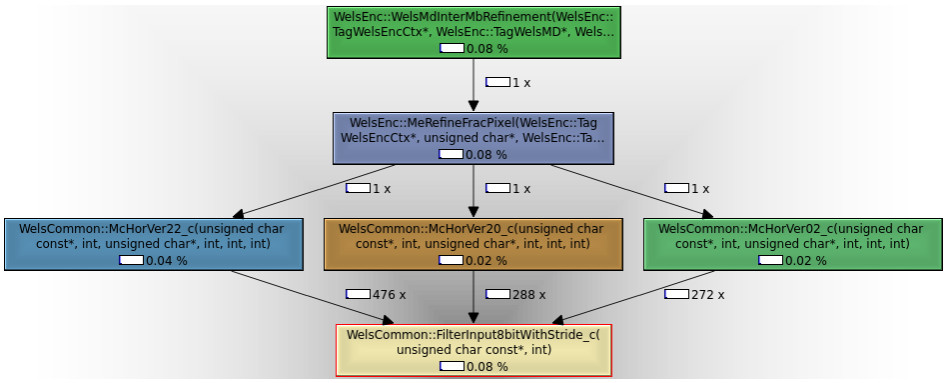



Figure 3.8 Graphe d'appel de la fonction welsMdInterMbRefinement() centré sur les filtres pour le traitement d'un macrobloc

### Analyse du temps d'exécution sous la plateforme virtuelle

Avant de nous lancer dans la création des modules, on se doit de vérifier les temps d'exécution de ces fonctions afin de calculer si le temps ajouté par les communications est plus grand que le temps d'exécution des fonctions. Si cela est le cas, il ne serait pas utile de nous lancer dans la modularisation de la fonction. Pour ce faire nous avons modifié la suite de tests pour extraire les statistiques d'exécution des fonctions cibles.

Afin de ne pas surcharger le flot de l'encodeur, nous avons opté pour partager des variables globales entre les diverses sections de code du module principal. Nous avons défini dans le fichier d'en-tête « *metric_variables.h* » trois variables pour chacune des fonctions. Une variable pour tenir le temps d'exécution moyen de la fonction, une pour le nombre d'appels à cette fonction pour traiter une trame et une dernière pour le temps cumulatif d'exécution de la fonction pour une trame.

Des sondes ont été mises avant et après l'appel de ces fonctions afin d'obtenir le temps d'exécution d'un appel. Par la duite, nous calculons le temps d'exécution moyen en divisant le temps cumulatif par le nombre d'appels de la fonction. Ces variables étant partagées, le module d'encodage est en mesure de calculer d'autres statistiques à partir de celles-ci et les réinitialiser à la fin du traitement d'une trame.

Les temps avant et après l'appel de la fonction sont pris avec la fonction **WelTimes()** de la librairie. Celle-ci a été modifiée par nous afin d'utiliser le périphérique **simulation_timer** lorsqu'utilisée sous Linux. Ce périphérique fournit à l'appelant le temps SystemC en pico-

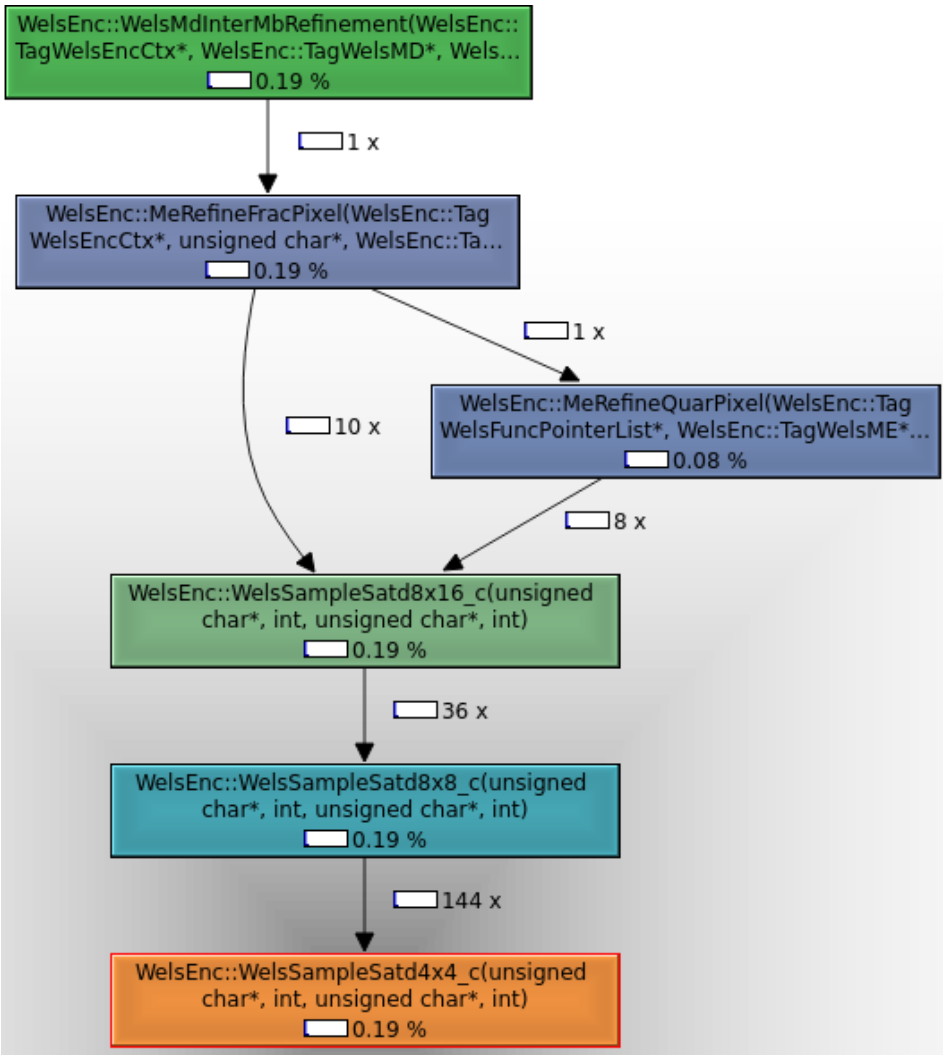



Figure 3.9 Graphe d'appel de la fonction welsMdInterMbRefinement() centré sur les SAD pour le traitement d'un macrobloc

seconde depuis le début de la simulation. Ceci permet d'avoir un temps plus stable que l'horloge interne de QEMU qui est moins précise. Lorsque le projet est synthétisé et implémenté, ce périphérique est remplacé par l'horloge interne de la carte cible. Dans les deux cas, la soustraction du temps de fin au temps de lancement de la fonction nous donne le temps d'exécution. Ceci ajoute le module de monitorage des performances **EncoderPerfMonitor** afin d'abstraire le périphérique et permettre l'interopérabilité entre la simulation et l'implémentation.

Dans le module d'encodage principal, nous mesurons aussi le temps d'exécution total pour l'encodage d'une trame. Nous utilisons cette valeur afin de déterminer le pourcentage du

temps d'exécution utilisé par chacune des fonctions. Afin d'avoir cette métrique, nous utilisons le temps moyen d'exécution de la fonction que l'on multiplie par le nombre d'appels. Ce résultat est ensuite divisé par le temps d'exécution total de la trame. Le module se charge aussi de maintenir les valeurs maximum et minimum pour chacune des métriques. Ces métriques sont affichées à la console et mises dans un journal. Le tout peut être désactivé à la compilation. Le listage 3.3 donne un exemple de résultat mis au journal après une exécution.

Les statistiques démontrent que nous avons un temps d'exécution de 128 microsecondes et 196 appels par trame donnant un usage de 37% du temps du processeur pour la fonction **welsMdInterMbRefinement()** et un temps d'exécution de 80 microsecondes et 607 appels par trame donnant un usage de 45.5% pour la fonction **welsMotionEstimateSearch()**. Afin de déterminer la latitude que nous avons pour transformer ces fonctions en coprocesseur matériel, on se doit maintenant d'analyser le surcoût des méthodes de communications sous SpaceStudio.

```
[ENCODER][MainEncoderTask][INFO] Execution done!
[WelsMdInterMbRefinement][NBFrames]: 267
-----------------------------------------------------------
[WelsMdInterMbRefinement][Time us MIN]: 108
[WelsMdInterMbRefinement][Time us MAX]: 128
[WelsMdInterMbRefinement][Usage MIN]: 0.121279 %
[WelsMdInterMbRefinement][Usage MAX]: 0.369318 %
[WelsMdInterMbRefinement][MB Calls MIN]: 44
[WelsMdInterMbRefinement][MB Calls MAX]: 196
-----------------------------------------------------------
[WelsMotionEstimateSearch][Time us MIN]: 60
[WelsMotionEstimateSearch][Time us MAX]: 80
[WelsMotionEstimateSearch][Usage MIN]: 0.138379 %
[WelsMotionEstimateSearch][Usage MAX]: 0.455655 %
[WelsMotionEstimateSearch][MB Calls MIN]: 46
[WelsMotionEstimateSearch][MB Calls MAX]: 607
-----------------------------------------------------------
[ENCODER][MainEncoderTask][INFO] Checksum encountered 0 errors
-----------------------------------------------------------
```

Listing 3.3 Statistiques écrites au journal après une exécution

**Analyse des temps de communication de SpaceStudio**

Nous avons développé une application test afin d'analyser deux des modes de communication sous SpaceStudio. Ces modes sont par file (fifo) et par mémoire interposées. Nous n'utiliserons pas la méthode DMA et par mémoire DDR étant donné que celles-ci sont limitées au processeur ARM. L'approche par mémoire interposé utilise les mémoires partagés BRAM afin de communiquer entre les modules, mais nécessite la synchronisation par fifo. La méthodologie est similaire à celle utilisée pour analyser le temps d'exécution des fonctions effectuer à la section précédente. Les sondes ont été mises avant et après les appels aux fonctions de communication de SpaceStudio. Les métriques sont prises sous la plateforme virtuelle de SpaceStudio tout comme l'analyse précédente.

Nous avons effectué deux types de test :

— Un envoi de trame complète sous forme d'une communication incluant 256 macroblocs.

— Un envoi d'une trame, un macrobloc à la fois, soit 256 communications.

Les résultats de cette analyse sont affichés dans la table 3.3. Nous avons effectué ces comparaisons pour les deux types de communication préalablement mentionnés, soit fifo et BRAM. La colonne ARM Linux représente le temps de communication pour un module logiciel sous une architecture ARM exécutant Linux.

Tableau 3.3 Analyse des temps de communication pour une trame

| **Tests effectués** | **Type Comm.** | **Direction** | **ARM Linux ($\mu s$)** | | |
|---|---|---|---|---|---|
| | | | min | max | **moy** |
| Trame Complète | FIFO (64) | Lecture | 34 | 89 | **56** |
| Trame Complète | BRAM | Lecture | 39 | 230 | **76** |
| MB par MB | FIFO(64) | Lecture | 3306 | 3525 | **3389** |
| MB par MB | BRAM | Lecture | 3148 | 3558 | **3197** |
| Trame Complète | FIFO (64) | Écriture | 40 | 84 | **54** |
| Trame Complète | BRAM | Écriture | 41 | 81 | **51** |
| MB par MB | FIFO (64) | Écriture | 3517 | 3549 | **3530** |
| MB par MB | BRAM | Écriture | 3489 | 3742 | **3547** |

De ces résultats on remarque que lire ou écrire une trame macrobloc par macrobloc est beaucoup plus coûteux que d'effectuer la même opération à l'aide d'une seule communication. Par la suite, nous avons analysé les communications pour un seul macrobloc, la composante la plus lue et écrit dans tout l'algorithme de l'encodeur. Les résultats sont illustrés à la

table 3.4. Finalement, les tables 3.3 et 3.4 démontrent que l'utilisation des communications par FIFO ne sont pas optimales comparativement à la mémoire interposée. L'utilisation des BRAM comme tunnel de communication s'avère l'option la plus rapide dans la plupart des cas, mais cela nécessite plus de ressources sur le FPGA.

Tableau 3.4 Analyse des temps de communication pour un macrobloc

| **Tests effectués** | **Type Comm.** | **Direction** | **ARM Linux ($\mu s$)** | | |
|---|---|---|---|---|---|
| | | | min | max | **moy** |
| Macrobloc | FIFO (64) | Lecture | 30 | 4333 | **67** |
| Macrobloc | BRAM | Lecture | 33 | 4334 | **71** |
| Macrobloc | FIFO(64) | Écriture | 15 | 93 | **52** |
| Macrobloc | BRAM | Écriture | 19 | 57 | **38** |

Les résultats de la section présente nous indiquent la latitude que nous avons pour la migration de nos fonctions en coprocesseur matériel. Afin de calculer le gain possible en fonction du temps de communication nous avons dérivé les équations suivantes. Le gain est calculé à l'aide de l'équation 3.1.

$$\begin{aligned} (t_{initial} - t_{accel}) - t_{comm} &= gain \\ gain > 0 &\rightarrow accélération \\ gain = 0 &\rightarrow statuquo \\ gain < 0 &\rightarrow décélération \end{aligned} \tag{3.1}$$

De cette équation nous pouvons déduire la relation entre le temps des communications et l'accélération afin d'obtenir un gain. Pour ce faire, nous devons avoir l'égalité illustrée par l'équation 3.2.

$$(t_{initial} - t_{accel}) > t_{comm} \tag{3.2}$$

Le temps initial auquel on soustrait le temps avec accélération doit être plus grand que le temps des communications. De cela, nous sommes en mesure de déterminer l'accélération minimale afin d'obtenir le statu quo et éliminer le délai ajouté par les communications. Ceci est illustré par l'équation 3.3.

$$(t_{initial} - t_{comm}) = t_{accel\_min} \tag{3.3}$$

Pour déterminer l'effort d'accélération requis pour nos modules matériels, nous avons donc calculé le gain minimal nécessaire pour les modules FME et IME. Si chaque macrobloc prend un maximum de 71 microsecondes à charger dans le pire des cas comme démontré précédemment par nos tests, cela laisse place à une accélération potentielle illustrée par l'équation 3.4.

$$\begin{aligned} FME : (128\mu s - 71\mu s) = 57\mu s \\ IME : (80\mu s - 71\mu s) = 9\mu s \end{aligned} \tag{3.4}$$

Pour welsMdInterMbRefinement, nous devons avoir un temps d'accélération plus petit que $57\mu s$ pour avoir un gain alors qu'il en faut un temps inférieur à $9\mu s$ pour welsMotionEstimateSearch. Ceci illustre que les communications prendront une portion respectable du temps d'exécution comparativement au temps de la fonction. Nous savons qu'il nous faudra une optimisation substantielle afin d'atteindre ces gains. Nous avons tout de même décidé de créer ces modules afin d'avoir un meilleur échantillon de notre méthodologie. Nous avons décidé de minimiser les opérations par fifo et prioriser les opérations par mémoire interposée.

### 3.4.7 Modularisation des zones chaudes du code

Afin d'isoler les modules progressivement, on se doit tout d'abord de déterminer leurs entrées/sorties afin de respecter la modélisation par processus de Kahn. Étant donné que l'encodeur travail avec des structures de pointeurs vers des zones mémoire et celles-ci qui sont passées par paramètre aux fonctions à modulariser, on se doit de déterminer où elles pointent et quel est le patron de consommation mémoire de l'encodeur. Lorsque le module sera passé du côté matériel, il sera isolé de l'espace d'adressage du processeur et ces pointeurs ne seront plus valides. Il est donc nécessaire de trouver une alternative pour fournir les données nécessaires aux modules matériels.

#### Analyse du flot de traitement d'un macroblock

Avant de décrire les opérations effectuées afin de produire des modules matériels à l'aide fonctions **welsMotionEstimateSearch()** et **welsMdInterMbRefinement()**, nous devons brièvement expliquer le flot de traitement d'un macrobloc sous la librairie OpenH264 pour mieux comprendre les modifications apportées lors de la migration.

La figure 3.5 illustre la hiérarchie des fonctions névralgique appelée lors du traitement d'un macrobloc. La boucle de traitement principale de l'algorithme **welsMdInterMbLoop()** boucle sur les 256 macroblocs d'une SLICE et se charge de préparer les structures de données pour les autres fonctions. L'algorithme fait tout d'abord une recherche temporelle (inter) pour un macrobloc entier (16x16) à l'aide de la fonction **welsMdP16x16()**. Celle-ci appel la fonction d'estimation du mouvement entier **welsMotionEstimateSearch()**. L'estimation du mouvement entier (IME) utilise l'algorithme de recherche **welsDiamondSearch()** basé sur la « Small Diamond search patern » (SDSP) [53]. Cette recherche demande l'accès à une fenêtre de macrobloc de référence. Cette fenêtre est paramétrée par la variable **ITERATIVE_TIMES** qui est assignée à 16 pixels par défaut. L'algorithme calcule une différence absolue (SAD) entre le macrobloc à encoder et 16 références dans une fenêtre de recherche de 9 macroblocs. La meilleure valeur est sauvegardée en mémoire dans la structure de donnée ME.

Par la suite, l'algorithme effectue une recherche spatiale (intra) dans la trame à encoder **welsMdFirstIntraMode()**. La recherche est effectuée sur des macroblocs de 16x16 à l'aide de la fonction **welsIdP16x16()**. Cette fonction calcule une différence absolue (SAD) entre le macrobloc et les neuf interpolations possibles. Celles-ci sont détaillées à la figure 2.11 de la section 2.4.2.

Suite à ces deux recherches, l'algorithme décide de poursuivre en fonction du plus petit différentiel. Si la prédiction spatiale est meilleure, celle-ci sera raffinée par la fonction **welsMdIntraFinePartitionVaa()**, autrement c'est la prédiction temporelle qui le sera. Dans notre cas, ce qui nous intéresse est le chemin suivi par le raffinement temporel (inter).

La première analyse effectuait une recherche pour un macrobloc de 16x16 pixels. Le raffinement cherche à découper le macrobloc en segment plus petit afin de voir s'il existe une meilleure correspondance avec l'image de référence. L'algorithme recherche alors pour les configurations 8x8, 8x16 et 16x8 au pixel près (IME) à nouveau à l'aide de la fonction **welsMotionEstimateSearch()**. Lors de recherche avec des macroblocs plus petits, les SAD sont additionnés. Par exemple, lors d'une recherche 8x8, quatre sous macrobloc sont évalués et chacun de leur SAD est additionné. Ce total est utilisé comme valeur pour déterminer si cette sous-recherche est meilleure. Pour leur part, les recherches 8x16 et 16x8 contiennent chacun deux sous-macroblocs.

Après avoir déterminé la meilleure configuration de macrobloc 16x16, 16x8, 8x16 ou 8x8, l'algorithme effectue une nouvelle recherche au demi-pixel et au quart de pixel (FME) à

Tableau 3.5 Flot de haut niveau du traitement d'un macrobloc

**[welsMdInterMbLoop]** [**Boucle SLICE MB**]
| **[welsMdInterMb]**
| | **[welsMdP16x16]**
| | | **[welsMotionEstimateSearch]**
| | | | **[welsMotionEstimateInitialPoint]**
| | | | **[welsDiamondSearch]** [**Boucle ITERATIVE_TIMES(16)**]
| | | | | **[welsSampleSadFour16x16]** [**Boucle 4**]
| | | | | | **[welsSampleSad16x16_c]**[**Boucle 4**]
| | | | | | | **[welsSampleSad8x8_c]** [**Boucle 8**]
| | **[welsMdInterSecondaryModesEnc]**
| | | **[welsMdFirstIntraMode]**
| | | | Si la prédiction I <P, alors on raffine la prédiction spatiale
| | | | **[welsMdIntraFinePartitionVaa]**
| | | | Si la prédiction I >P, alors on passe en raffinement temporel
| | | | **[welsMdInterFinePartitionVaa]**
| | | | | **[welsMdP8x8]** [**Boucle 4**]
| | | | | **[welsMdP16x8]** [**Boucle 2**]
| | | | | **[welsMdP8x16]** [**Boucle 2**]
| | | | | | **[welsMotionEstimateSearch] (IME)**
| | | | | | | **[welsDiamondSearch]**
| | | | | | **[welsMdInterMbRefinement] 1/2, 1/4 pixel (FME)**
| | | | | | **[welsMdInterEncode]**

l'aide de la fonction **welsMdInterMbRefinement()**. Cette fonction recherche un déplacement fractionnel de pixel. Cette fraction est faite par la pondération à l'aide d'un filtre sur le déplacement d'un demi et un quart de pixel afin de détecter les mouvements plus petit qu'un pixel. Cette opération produit un nouveau macrobloc de référence qui servira pour la comparaison et le calcul du résiduel.

Ces recherches nous indiquent une certaine zone mémoire que l'on doit rendre accessible aux futurs modules matériels. Soit une fenêtre de 9x9 macroblocs pour la recherche initiale d'un bloc de 16x16 pixels. De plus, on se doit d'avoir accès au macrobloc à encoder ainsi que faire suivre la référence finale interpolée par l'algorithme FME.

### 3.4.8 Suppression des pointeurs de fonctions

Lors de la modularisation des fonctions, on doit éliminer certaines constructions liées à la programmation de haut niveau, car celles-ci ne sont pas compatibles avec la synthèse. Dans notre cas, on doit éliminer les pointeurs de fonction.

Afin d'éliminer ces pointeurs, nous avons analysé la trace du code de la référence (golden model) sur Windows et noté les fonctions appelées à travers ces pointeurs. Nous avons utilisé l'application Vagrind avec l'outil Callgrind pour effectuer cette analyse. L'outil nous indique directement dans le code source de notre application quelles fonctions ont été appelées et le nombre d'appels. La figure 3.10 offre un exemple de l'analyse offerte par Valgrind pour la fonction **WelsDiamondSearch()**.

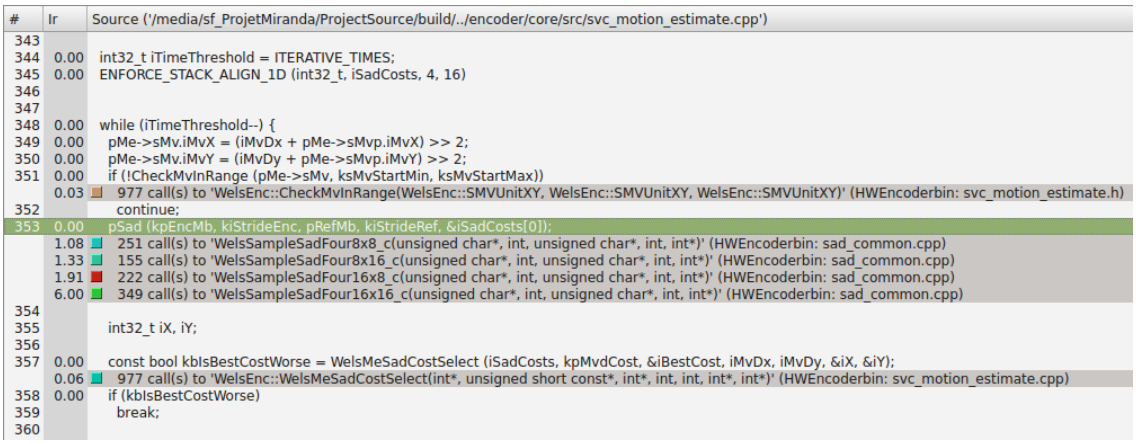



Figure 3.10 Liste des fonctions appelées à travers le pointeur pSad de la fonction WelsDiamondSearch()

Les pointeurs de fonction sont équivalents à un saut d'exécution dans un code. Il s'agit donc de passer l'exécution au segment de code à l'adresse indiquée par le pointeur. Cela est dynamique à l'exécution et le logiciel ne connaît pas toutes les possibilités de ce pointeur. Cette fonctionnalité n'a aucune équivalence en RTL et les outils HLS ne peuvent les synthétiser. Si on veut effectuer un traitement particulier selon une option en RTL on doit créer un multiplexeur (MUX) qui est similaire à un « switch case » dans un langage de haut niveau. Il faut donc transformer ces pointeurs de fonction en « switch case » pour rendre les chemins de contrôle explicite et permettre à la synthèse haut niveau de créer un MUX. Nous avons effectué ces modifications pour les fonctions **welsDiamondSearch()**, **MeRefineFracPixel()**, **MeRefineQuarPixel()** et **WelsMotionEstimateInitialPoint()**. Ces fonctions utilisent les fonctions de SAD dépendamment du bloc à analyser (8x8, 8x16,16x8, 16x16). Elles utilisent des tableaux de pointeur vers ces fonctions. Celles-ci étant toutes utiles pour les modules, leurs tableaux de pointeur ont été transformés en MUX (**SampleSadFourSelectionMux()**, **SampleSatdSelectionMux()** ) tel qu'illustré par le listage 3.4.

```
//============ Pointeur de fonction ============
PSample4SadCostFunc pSad = pFuncList->sSampleDealingFuncs.pfSample4Sad[pMe->uiBlockSize];
pSad (kpEncMb, kiStrideEnc, pRefMb, kiStrideRef, &iSadCosts[0]);

//============    Mux    ============
inline void EncoderIMEModule::SampleSadFourSelectionMux (uint8_t block_type, uint8_t* iSample1, int32_t
    iStride1, uint8_t* iSample2, int32_t iStride2, int32_t* pSad)
{
        switch(block_type)
        {
                case BLOCK_16x16:
                        WelsSampleSadFour16x16_c(iSample1, iStride1, iSample2, iStride2, pSad);
                        break;
                case BLOCK_16x8:
                        WelsSampleSadFour16x8_c(iSample1, iStride1, iSample2, iStride2, pSad);
                        break;
                case BLOCK_8x16:
                        WelsSampleSadFour8x16_c(iSample1, iStride1, iSample2, iStride2, pSad);
                        break;
                case BLOCK_8x8:
                        WelsSampleSadFour8x8_c(iSample1, iStride1, iSample2, iStride2, pSad);
                        break;
                case BLOCK_4x4:
                        WelsSampleSadFour4x4_c(iSample1, iStride1, iSample2, iStride2, pSad);
                        break;
        }
}
```

Listing 3.4 Convertion d'un tableau de pointeur de fonction en code conditionnel

Dans d'autres cas, nous avons fixé la fonction appelée sans prendre tous les cas en considération, car les options utilisées étaient fixes à la compilation. Aucun besoin de les déterminer en cours d'exécution. Autre cas, la fonction **welsMotionEstimateSearch()** qui à travers le pointeur *pfCalculateSatd* appele la fonction **NotCalculateSadCost()** représentant une fonction vide. Pour ce cas, nous avons retiré l'appel complètement. Même chose pour la fonction **welsMotionEstimateInitialPoint()** qui à travers le pointeur *pfCheckDirectionalMv* appelait la fonction vide **checkDirectionnalMvFalse()**. Finalement, la fonction **WelsMotionEstimateSearch()** utilise le pointeur *pfSearchMethod* afin de sélectionner la méthode de recherche pour la prédiction entière. Cet aiguillage dépend de la source vidéo. Étant donné que le type de notre source est fixe à **CAMERA_VIDEO_NON_REAL_TIME**, nous avons fait l'appel directement à la fonction **welsDiamonSearch()** associée au pointeur *pfSearchMethod* et retiré celui-ci. Ces types de modifications apportent par contre un problème contrairement au MUX. Si on change la configuration de l'encodeur pour une de plus haute qualité, l'appel direct des fonctions choisi ne sera plus valide, car ces pointeurs de fonctions changent alors de cible.

### 3.4.9 Abstraction mémoire

Un grand obstacle à la modularisation de n'importe quelle partie du système est l'utilisation de la mémoire partagée (heap). Lors du développement d'une application logiciel de type C/C++, l'allocation dynamique est courante. Dans ce modèle de haut niveau, l'accès mémoire est représenté par un déréférencement de pointeur. Les modules matériels n'ont pas accès à la mémoire centrale du processeur et ne partagent aucunement le même adressage virtuel. Si nous voulons respecter la modélisation par processus Kahn, ces opérations mémoires doivent être contrôlées par SpaceStudio et nous devons aussi traduire les pointeurs mémoires en adresse valide pour les modules matériels.

Le flot complet de l'encodage utilise des pointeurs vers l'espace d'adressage de la mémoire centrale pour réduire la quantité de données qui voyage entre ses différentes sections. La librairie a été développée avec le principe de mémoire unifié. Sous un même processus/processeur, les pointeurs peuvent être partagés. Relocalisé ces accès mémoire vers un périphérique mémoire virtuel contrôlé par SpaceStudio est irréalisable dans un court laps de temps. Le processus serait long et propice aux erreurs, car ceci consiste à changer tous les déréférencements de pointeurs qui représentent une lecture par la fonction **DeviceRead()** et ceux qui représentent une écriture par la fonction **DeviceWrite()**. Ceci est sans compter le changement d'adressage. Au moment de réaliser ce travail, SpaceStudio n'avait encore aucune facilité pour accomplir cette tâche. Nous avons donc concentré nos efforts sur les fonctions à modulariser. Ceci veut aussi dire que nous ne pouvons pas introduire une mémoire globale partagée entre les modules matériels et logiciels. Il est donc nécessaire de faire véhiculer les données nécessaires au traitement vers les modules matériels.

#### Mémoire partagé

Afin de produire une mémoire accessible autant par le logiciel que par le matériel pour partager les données requises à l'encodage, nous avons ajouté deux BRAM au projet. La première de grandeur de 161ko pour recevoir la trame à encoder et la seconde de grandeur de 900Ko pour recevoir la ou les trames de références. L'encodeur reçoit des trames de dimension 320x180 pixels, mais leurs dimensions sont modifiées à deux reprises. La première afin de respecter l'alignement en mémoire, le module principal ajoute alors 12 lignes à l'image pour une dimension de 320x192. Par la suite, celle-ci est déplacée dans un tampon plus large afin d'agrandir l'image. L'algorithme ajoute des bordures pour permettre la détection de mouvement aux limites de l'image. La librairie ajoute 64 lignes et colonnes à l'image telle

qu'illustrée par la figure 3.12. La dimension finale d'une image en mémoire est de 384x256 pixels pour la luminance (Y) et 192x128 pour les chrominances (CbCr). Le total nécessaire en mémoire est de 144Ko.

Le traitement des macroblocs débute à la fonction **welsMdInterMbLoop()**, il est donc nécessaire de charger dans les BRAM les macroblocs à encoder ainsi que les références avant l'appel de cette fonction. On doit donc déterminer à quel moment l'image de références et les images à encoder sont prêtes. Nous avons analysé le flot d'une trame en entrée et avons déterminé les structures et fonctions pertinentes. Ce flot est illustré par la figure 3.11.

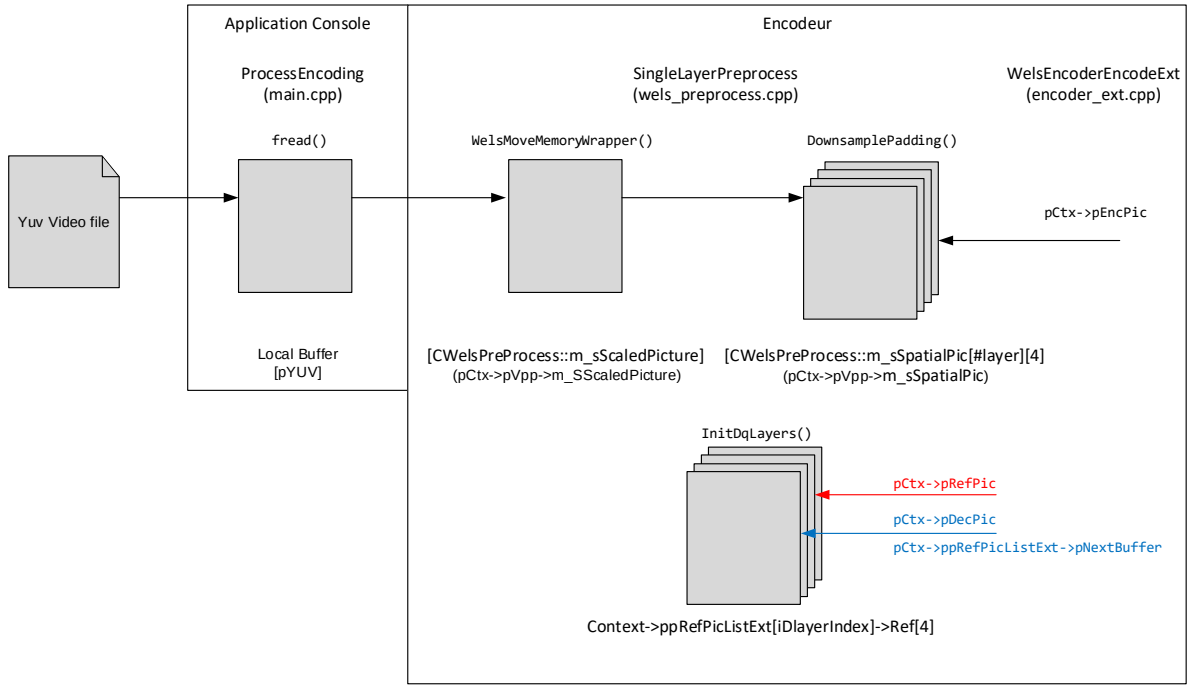



Figure 3.11 Flot d'entrée des images et pointeurs vers les tampons de l'encodeur

Le pointeur *pEncPic* représente l'adresse de la trame à encoder. Cette trame réside dans le tampon *m_sSpatialPic* et ceux-ci sont fixes après la mise à l'échelle faite par la fonction **DownsamplePadding()** à l'intérieur de la fonction **SingleLayerPreprocess()**. Cette structure et ce pointeur sont mis à jour à chaque nouvelle trame à encoder lors de l'appel de la fonction d'encodage **WelsEncoderEncodeExt()**. L'image à encoder doit donc être envoyée du côté matériel après la fonction **DownsamplePadding()**. Nous avons donc créé une

fonction de transfert nommée **sendFrameToHWFrameBuffer()** que nous avons branchée dans la fonction **SingleLayerPreprocess()** à la suite de l'appel à la fonction **DownsamplePadding()**.

La fonction **sendFrameToHWFrameBuffer()** envoie la trame vers le tampon matériel. Elle utilise la structure *SPicture* de la librairie afin : 1) d'obtenir l'adresse mémoire où réside l'image et 2) de calculer sa grandeur afin de faire le transfère vers la BRAM. Elle reçoit en paramètre le pointeur *pDstPic* qui pointe vers la structure *SPicture* dans le tampon *m_pSpatialPic*. Cette structure contient un tableau de pointeurs donnant accès aux différentes couches de l'image telle qu'illustrée par la figure 3.12. Pour calculer la grandeur de l'image, nous utilisons ces différents pointeurs. La soustraction *pData[1] - pData[0]* nous donne la grandeur de la composante Y alors que la soustraction *pData[2] - pData[1]* nous donne la grandeur de la composante Cb. Sachant que les deux chrominances sont de même taille, ce résultat est multiplié par deux et additionné à la taille de la composante Y. Ceci nous donne la grandeur totale de la trame. Ce calcul inclut une partie du « padding » qui est aussi transféré. Nous avons opté pour cette approche plus simple pour la première itération du système pour ne pas modifier les calculs de déplacement en mémoire pour faciliter la migration.

La même approche a été utilisée pour les images de références. Nous avons analysé le flot de ces images afin de trouver le meilleur endroit pour les transférer vers la mémoire partagée. Lorsqu'une image est encodée, celle-ci est placée dans la liste des images de référence. Il y a donc une trame de référence à la fin du traitement de la fonction boucle **WelsEncoderEncodeExt()**.

Nous avons mentionné quelques fois qu'un encodeur H.264 décode le résultat de l'encodage au fur et à la mesure afin de servir de référence. Or, pour la librairie OpenH264, le résultat du décodage d'un macrobloc est mis dans le tampon *ppRefPicListExt* par le pointeur *pDecPic*. Comme on peut voir sur la figure 3.11 ce pointeur ainsi que le pointeur de référence *pRefPic* pointe vers le même tampon mémoire. Ce tampon est aussi mis à jour au fur et à la mesure de l'encodage d'une trame. Une nouvelle trame de référence est disponible à la fin de l'exécution de la fonction d'encodage **WelsEncoderEncodeExt()**. Les pointeurs *pRefPic* et *pDecPic* sont mis à jour suite à la fonction **WelsBuildRefList()** par la fonction **PrefetchReferencePicture()** et sont fixe au cours de l'encodage de la trame.

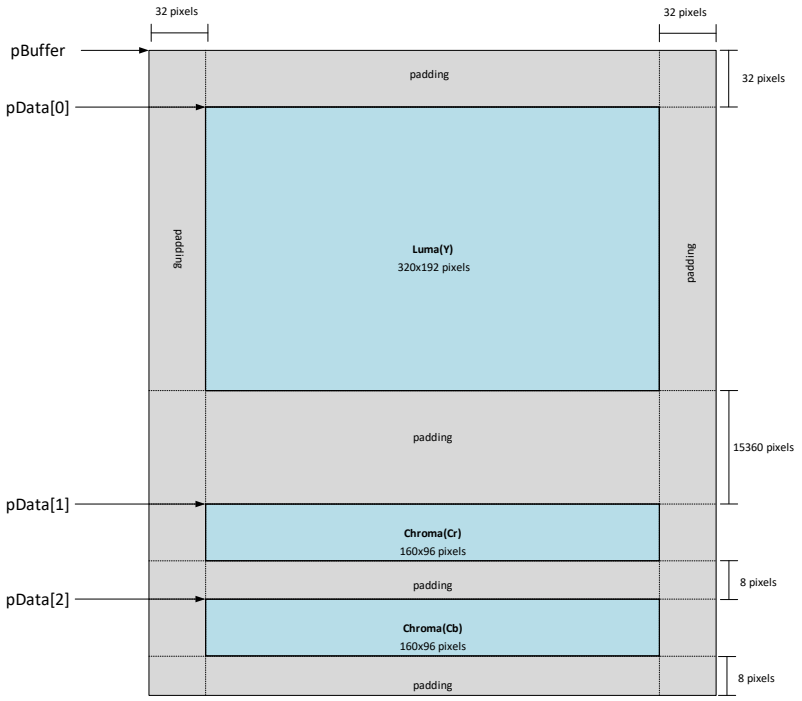



Figure 3.12 Disposition d'une trame YUV en mémoire par la structure *SPicture* d'OpenH264

Nous avons donc créé la fonction **updateHWReferenceFrameBuffer()** et ajouter le transfère à la fonction **PrefetchReferencePicture()**. Tout comme pour la trame à encoder, la fonction reçoit en paramètre le pointeur *pDstPic* qui pointe vers la structure *SPicture* dans le tampon *ppRefPicListExt*. À l'aide de celle-ci, nous calculons la grandeur de l'image et exécutons le transfert vers la mémoire partagée.

### Cache de travail

Comme mentionné à la section 3.4.7, les modules matériels doivent avoir accès à une zone minimum de 9x9 macroblocs de référence ainsi que le macrobloc à encoder. Par contre, nous avons rencontré un problème avec l'alignement en mémoire. En effet, lors d'une recherche, l'algorithme effectue des déplacements un pixel à la fois dans la zone de recherche. Ceci demande des lectures à des adresses non multiples de quatre. Il s'agit d'accès dit non aligné. La plupart des processeurs ont une restriction sur l'alignement des accès mémoire, mais certain plus complexe comme l'architecture x86 ou ARM permettent les accès non alignés. Ceci explique pourquoi la librairie n'a aucune difficulté à s'exécuter sur les cibles prévues par les développeurs. Par contre, SpaceStudio voulant être agnostique de la plateforme cible avec sa librairie de communication n'autorise pas les accès non alignés. Il ne fut donc pas

possible d'utiliser une zone mémoire externe afin d'effectuer les recherches de référence sans implémenter un algorithme d'alignement. Nous avons alors opté pour effectuer ces recherches sur une mémoire cache (scratchpad) interne au module. Les modules matériels se chargent alors de mettre dans leur mémoire locale les données nécessaires pour leur recherche au pixel près.

Nous avons tout d'abord créé une nouvelle fonction à l'intérieur de chacun des modules qui se charge d'aller dans la mémoire partagée et récupérer le macrobloc à encoder. Comme ces modules sont appelés pour des recherches de diverses dimensions (16x16, 8x16, 16x8, 8x8), la fonction se charge de passer les bons paramètres à la fonction **MoveFrameBufferMemory()** selon le type de bloc. Les caches de travail sont alors utilisées pour tous les types de dimension.

```
void  EncoderIMEModule::MoveFrameBufferMemory (uint8_t* pDstY, int32_t
   iDstStrideY, uint32_t pSrcY, int32_t iSrcStrideY, int32_t iWidth,
   int32_t iHeight )
{
        eSpaceStatus m_CommStatus;
        int32_t   j;

        for (j = iHeight; j; j--) {
                m_CommStatus = DeviceRead(HW_INPUT_FRAMEBUFFER_ID, pSrcY,
                   pDstY, iWidth);
                pDstY += iDstStrideY;
                pSrcY += iSrcStrideY;
        }

}
```

Listing 3.5 Fonction générique de transfert de la mémoire partagée vers la cache de travail

Par la suite, nous avons créé une autre fonction à l'intérieur de chacun des modules qui se charge d'aller en mémoire partagé et charger la référence dans le « scratchpad ». Tout comme pour le macrobloc à encoder, ces modules sont appelés pour des recherches de dimensions diverses (16x16, 8x16, 16x8, 8x8). Cette même fonction utilise à son tour la fonction de transfert générique **MoveFrameBufferMemory()**. L'architecture mémoire finale est illustrée à la figure 3.13.

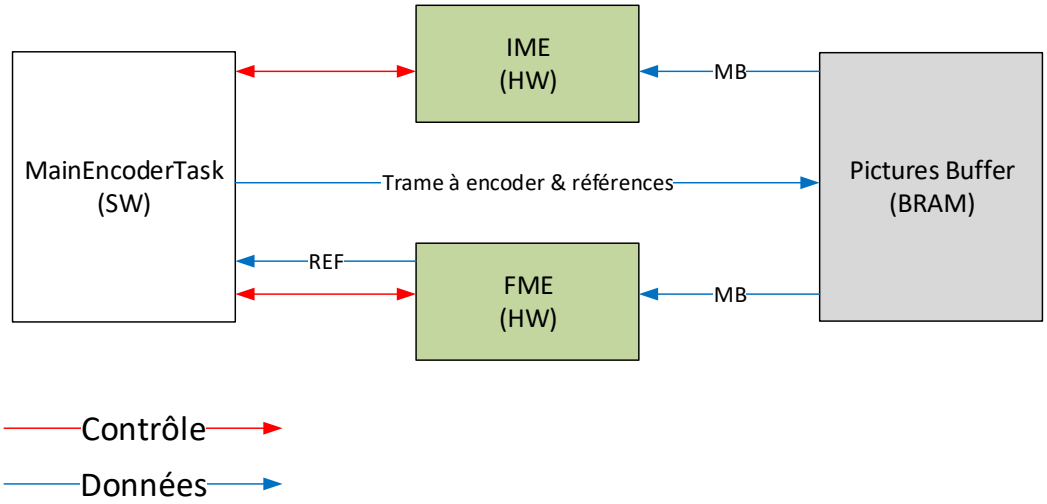



Figure 3.13 Schéma bloc de l'architecture mémoire

### 3.4.10 Migration des modules d'estimation du mouvement

#### Module IME

Nous avons créé le module **EncoderIMEModule** dans SpaceStudio et commencé la migration de la fonctionnalité de l'estimation du mouvement entier. Afin d'intégrer l'algorithme dans notre module, nous avons tout d'abord analysé les entrées et sorties de ses différentes fonctions. On connaît déjà les besoins mémoires en lien avec les macroblocs, mais nous devons isoler les autres données. Comme la librairie utilise des passages par référence à l'aide de pointeur, les entrées/sorties des fonctions ne sont pas tous explicites. On doit alors analyser l'utilisation de ces pointeurs.

L'analyse a deux objectifs. Le premier est de réduire les données passées entre le module principal et le module matériel, car les structures de l'encodeur sont très volumineuses et il serait plus pratique de passer seulement les paramètres requis et retourné les valeurs nécessaires. Le second est que les modules ne partagent pas le même espace d'adressage. Il faut donc traduire en index local au cache de travail des modules matériels, puisqu'on ne peut pas prendre les valeurs telles quelles.

Nous avons complètement ratissé toutes les fonctions utilisées et, après plusieurs semaines de travail, nous avons déterminé les entrées et sorties minimales du module. Nous avons

réduit les structures initiales en plus petite structure spécialisée pour le module IME. La nouvelle structure nommée *IMEModuleInputComStruct* contient les informations nécessaires pour traiter une recherche. Ces informations incluent la position du macrobloc à encoder en mémoire principale (*me_EncYOffset*), le type de macrobloc 8x8, 16x8, 8x16 ou 16x16 (*me_BlockSize*) et cinq vecteurs de mouvement comme références initiales.

Toutes les fonctions de la librairie OpenH264 utilisées par le module ayant besoin de modification pour recevoir la nouvelle structure ont été transférées à l'intérieur du module. C'est le cas des fonctions **MotionEstimateSearch()**, **WelsMotionEstimateInitialPoint()** et **WelsDiamondSearch()**. Les autres ont été laissés dans leur fichier respectif tel que les fonctions de SAD. Nous voulons garder le code de la librairie intacte le plus possible afin de limiter le travail de migration et la réutilisation. Nous avons ajouté le préfixe HW à toutes les fonctions transférées à l'intérieur du module matériel.

Dans ce qui suit, nous présentons le flot de traitement du module IME illustré à la figure 3.14.

Le module se met tout d'abord en attente des paramètres nécessaire pour la fonction **MotionEstimateSearch()** en provenance du module **MainEncoderTask**. Ceci est fait par un appel **ModuleRead()** de la librairie SpaceLib. Les paramètres sont écrits dans le registre d'entrée de type *IMEModuleInputComStruct*. Le module tient pour acquis que les paramètres ont été traduits pour son espace d'adressage. Cette traduction est détaillée à la prochaine section (section 3.4.11).

Une fois les paramètres reçus, le module charge les données requises par l'algorithme de recherche dans les caches internes du module à l'aide de la fonction **loadLumaIntoScratchPad()**. Comme discuté préalablement, le module IME exécute une recherche entière de pixel afin de trouver une référence proche du macrobloc à encoder. En premier lieu, nous avons défini une cache pour contenir le macrobloc à encoder. Cette cache porte le nom *sp_encode_mb_luma* et possède 256 octets. Étant donné que l'estimation du mouvement se fait uniquement sur la luminance (Y), nous n'avons qu'à analyser ces 256 pixels sans tenir compte des chrominances rouge et bleu. Nous avons aussi définie deux caches pour les références nommées *sp_ref_mb_luma_1* et *sp_ref_mb_luma_2* ayant chacune une taille de 326 octets. Les références contiennent deux lignes et deux colonnes de plus afin de contenir l'information de bord de ces macroblocs. Afin de simplifier le design initial du module, nous n'avons pas créé de mémoire cache pour la fenêtre de 9x9 macroblocs.

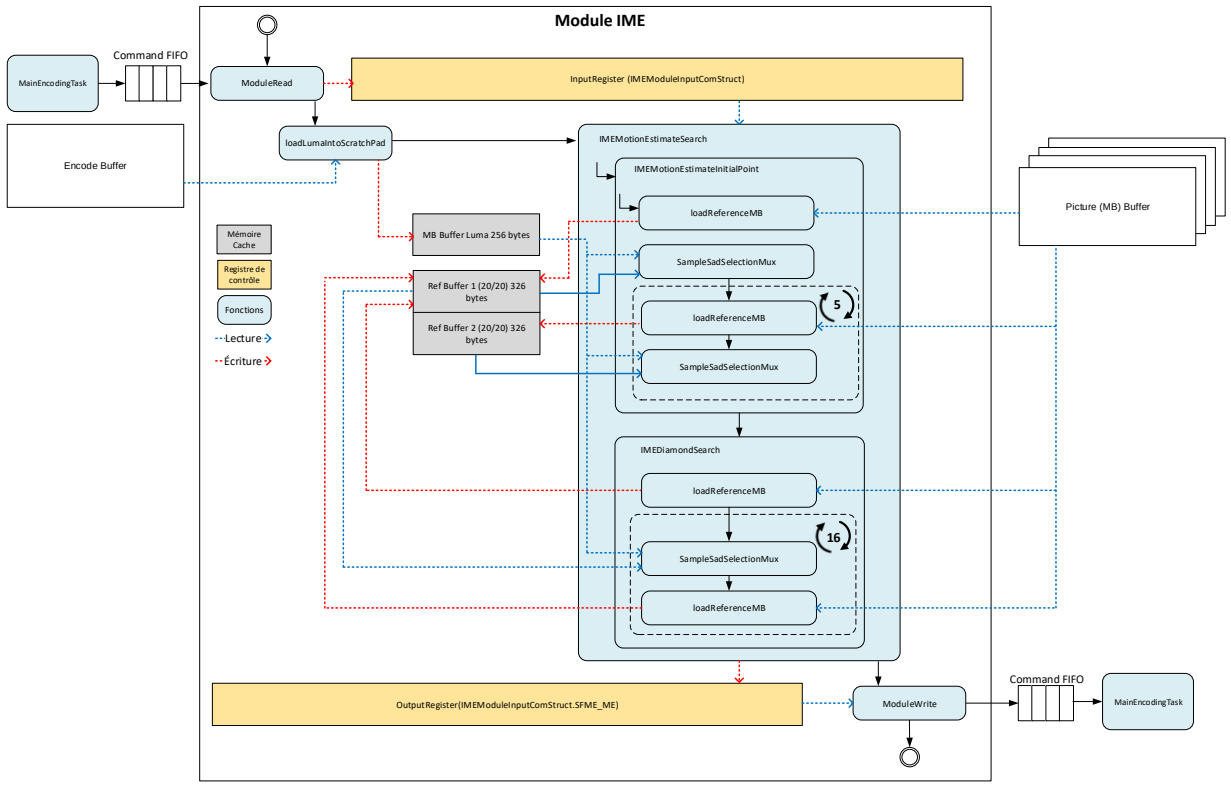



Figure 3.14 Schéma des opérations mémoire du module IME

Pour la première étape de la recherche IME, le module tente de trouver le meilleur point de départ pour effectuer la recherche. L'algorithme de recherche initiale est fourni par la fonction **WelsMotionEstimateInitialPoint()**. Comme mentionnés, nous avons en entrée les vecteurs de mouvement précédemment estimé pour les macroblocs voisins. L'idée est que si le mouvement est uniforme dans toute la trame, il est fort probable de trouver une référence pour notre macrobloc dans la même direction que ses voisins. Cette supposition permet de détecter plus rapidement une référence pour économiser sur le temps de calcul. Le voisinage d'un macrobloc est illustré à la figure 3.15. Ce point de recherche initial n'est cependant pas garanti d'être uniforme. Certains des macroblocs voisins peuvent avoir des vecteurs de mouvements qui pointent dans diverses directions. Ceci rend difficile le chargement en mémoire des références selon une fenêtre de recherche. Nous avons alors décidé de charger ces premières références, un macrobloc à la fois.

Le module charge une première référence en cache à l'aide de la fonction **loadReferen-**

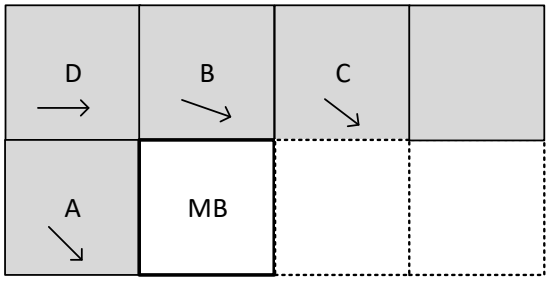



Figure 3.15 Vecteurs de mouvements dans le voisinage du macrobloc à encoder

**ceMB()** dans une des caches de référence. Un SAD est effectué à l'aide du MUX **SampleSadSelectionMux()** entre le bloc à encoder et la référence. Le bon type de SAD est effectué selon le type du bloc dans le registre de contrôle. Le résultat est conservé pour cette référence. Une seconde référence est chargée dans la seconde cache de référence. Un SAD est effectué entre cette nouvelle référence et le bloc à encoder. Les deux références sont par la suite comparées et celle ayant le plus petit SAD est conservée en cache. Le module charge alors une autre référence dans la cache du bloc discrédité. Cette comparaison est effectuée autant de fois qu'il y a de vecteurs de mouvement voisin. La meilleure estimation est conservée dans le registre de contrôle et celle-ci sert d'entrée pour la seconde étape de la recherche.

Une fois la zone de recherche initiale trouver, le module lance la deuxième étape de recherche afin de trouver une meilleure référence à l'aide de la fonction **IMEDiamondSearch()**. L'algorithme effectue une recherche dans quatre directions. Soit vers le bas, le haut, la gauche et la droite à l'aide des lignes et colonnes de bord amené par le module. Ceci permet de déterminer une nouvelle référence et une nouvelle direction de recherche. Les fonctions **WelsSampleSadFour** font ces quatre SAD (dans les quatre directions) en un appel. La meilleure direction est alors explorée par l'algorithme de recherche **DiamondSearch**. L'algorithme effectue aux maximum 16 déplacements ou jusqu'à ce que la meilleure direction ait un SAD plus élevé que la dernière référence. Tout au long des recherches, le vecteur de déplacement et l'adresse de la référence sont mis à jour dans le registre de contrôle.

Une fois la référence trouvée par le module, le détail de celles-ci dans le registre de contrôle est retourné au module principal par une **ModuleWrite()**.

Le module consomme un total de 908 octets de mémoire cache.

### Module FME

Nous avons créé le module **EncoderFMEModule** dans SpaceStudio et entrepris la migration de la fonctionnalité de l'estimation du mouvement fractionnelle. Afin d'intégrer cet algorithme un peu plus complexe dans notre module, nous avons à nouveau analysé les entrées et sorties de ses différentes fonctions. Comme l'algorithme utilise les mêmes structures que le module IME, il n'a pas été nécessaire de refaire une analyse en profondeur. Nous avons seulement ciblé les nouveaux paramètres utilisés en entrée et sortie.

Nous avons par la suie déterminé les entrées et sorties minimales du module. Nous avons réduit les structures initiales en plus petites structures spécialisées pour le module FME. La nouvelle structure nommée *FMEModuleInputComStruct* est un peu plus volumineuse que celle du module IME. Ceci est nécessaire, car l'algorithme de prédiction fractionnel utilise plus de paramètres afin de produire la prédiction finale. En plus des informations sur les composantes de luminance (Y) similaire au module IME, nous avons aussi besoin des informations des chrominances telles que la grandeur des composantes à encoder et leur référence ainsi que leur position en mémoire partagé (pointeur). L'algorithme est exécuté sur un ou plusieurs macroblocs tout dépendants de leur gardeur. La structure en entrée reflète cette nécessité ayant 9 sous-structures pour chaque combinaison de macroblocs possibles (quatre 8x8, deux 16x8 ou 8x16 et un 16x16). Par contre, le module n'utilise qu'un des groupes de macroblocs par exécution. Ceci veut dire qu'on pourrait seulement conserver quatre sous-structures au maximum au lieu de neuf.

En plus de ces valeurs, l'algorithme a besoin du type de paramètre *curMb_uiLumaQp*. Ce paramètre représente la qualité d'encodage entropique utilisé. Ce paramètre est utilisé afin de déterminer le coût d'encodage d'un vecteur de déplacement. Si le vecteur est peu commun, le coût du symbole sera plus élevé. Ce coût est alors comptabilisé avec le SAD de l'estimation afin de déterminer la meilleure référence. Le coût est calculé à l'aide d'une table de valeurs de 108 Ko en mémoire. Étant donné que le module FME utilise ces valeurs, il aurait été nécessaire que celle-ci soit accessible en matériel. Dans les deux cas ceci prendrait 108 Ko d'espace mémoire, une taille prohibitive. Étant donné que nous sommes limités dans notre utilisation de la mémoire locale sous forme de BRAM, nous avions à trouver une solution alternative. Après analyse de la table de valeurs, nous avons observé que la table était symétrique. C'est-à-dire que la table a un centre et les valeurs de chaque côté de ce centre sont

identiques. Il serait alors possible de réduire de moitié la taille de la table soit 54 Ko. Par contre, à travers cette observation, nous avons trouvé l'équation utilisée pour produire ces valeurs. Nous avons effectué le compromis et calculé en matériel les valeurs au besoin plutôt que de les entreposer en mémoire. Nous avons donc recréé cette équation en matériel. Ceci rend le calcul peu coûteux comparativement à l'approche logicielle. Cette nouvelle fonction **getMvdCost()** est partagée entre le module IME et FME dans le fichier nommé « *mvcost.h* ». L'appel se fait par l'intermédiaire de la macro **FME_COST_MVD** qui remplace la macro **COST_MVD** originale dans les modules.

Nous avons aussi créé une structure en sortie nommée *FMEModuleOutputComStruct* étant donné que nous avons plusieurs paramètres à retourner au module principal. Cette structure renvoie les vecteurs de déplacement pour tous les macroblocs calculés (jusqu'à 4 pour les macroblocs de grandeur 8x8), leur cache (détail de leurs voisins) ainsi que le coût final de la prédiction (*sadCost0*).

Les fonctions du module FME ayant besoin de modification pour recevoir les nouvelles structures ont été transférées à l'intérieur du module [**InitMeRefinePointer()** , **MdInterMbRefinement()**, **MeRefineFracPixel()**, **MeRefineQuarPixel()**, **UpdateMotionInfo()**, **PredMv()**, **PredInter8x16Mv()**, **PredInter16x8Mv()**]. Les autres ont été laissés dans leur fichier respectif [**McHorVer02_c()**, **McHorVer20_c()**, **PixelAvg_c()**, **WelsCopy*_c()**]. Tout comme pour le module IME, nous voulons garder le code de la librairie intacte le plus possible afin de limiter le travail de migration et la réutilisation du code. Certaines fonctions [**WelsSampleSatd*()**] ont été transférées afin de retirer les dépendances aux fichiers de librairies tels que Windows.h. Nous avons aussi transféré les fonctions qui doivent lire et écrire en mémoire [**McChroma_c()**, **McCopy_c()**, **McChromaWithFragMv_c()**, **McCopyWidthEq2_c()**, **McCopyImpl_c()**] afin d'utiliser la librairie SpaceLib et les appels **DeviceRead()** et **DeviceWrite()** pour faire ces opérations dans la mémoire partagée à partir du module matériel. Nous avons ajouté le préfixe HW à toutes les fonctions transférées à l'intérieur du module.

Dans ce qui suit, nous allons maintenant discuter du flot de traitement du module FME tel qu'illustré à la figure 3.16.

Le module se met tout d'abord en attente des paramètres nécessaires pour la fonction **HWMdInterMbRefinement()** en provenance du module **MainEncoderTask**. Ceci est fait par

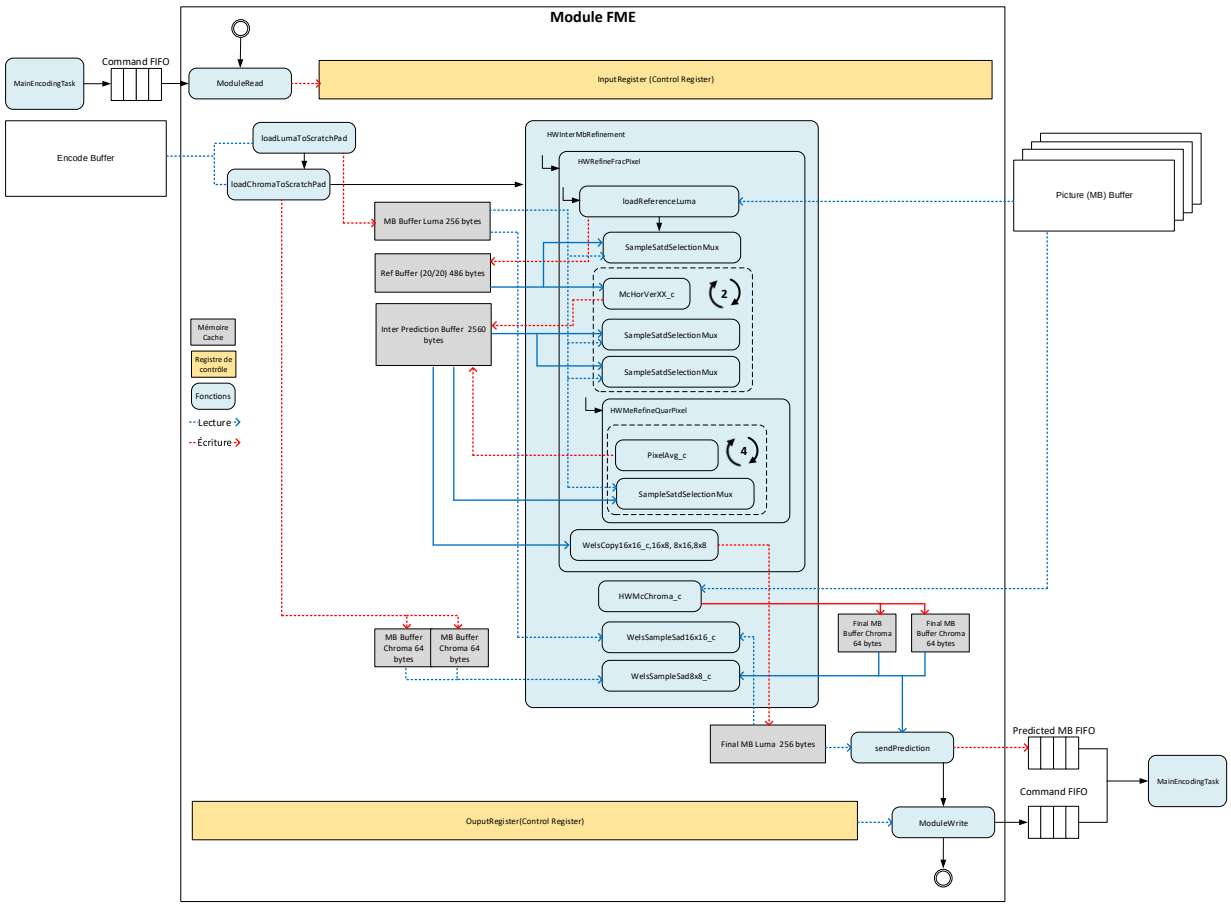



Figure 3.16 Schéma des opérations mémoires du module FME

un appel ModuleRead de la librairie SpaceLib. Les paramètres sont écrits dans le registre d'entrée de type *FMEModuleInputComStruct*. Le module tient pour acquis que les paramètres ont été traduits pour son espace d'adressage. Cette traduction est détaillée à la prochaine section (Section 3.4.11).

Une fois les paramètres reçus, le module charge les données à analyser par l'algorithme de recherche dans ses caches internes à l'aide de la fonction **loadLumaIntoScratchPad()** et **loadChromaBIntoScratchPad()**. Nous avons défini une cache pour contenir le macrobloc à encoder. Cette cache porte le nom *sp_encode_mb_luma* ayant 256 octets de taille. Dans le cas de l'estimation de mouvement aux sous pixel, nous avons aussi besoin des chrominances bleu et rouge dans le cas où le bloc analysé est de dimension 16x16. Nous avons donc définie deux caches de 64 octets nommés *sp_encode_mb_cb* et *sp_encode_mb_cr*. Compte

tenu du sous-échantillonnage (4 :2 :0), nous avons seulement le quart des valeurs de chrominance comparativement à la luminance. Les chrominances sont utilisées pour calculer le SAD complet d'un bloc afin de savoir si l'encodeur doit encoder ou passer le macrobloc. Dans la norme H.264, l'encodeur peut ne pas encoder un bloc si le coût SAD de toute prédiction est plus élevé que l'interpolation de ses voisins. C'est alors au décodeur de créer le macrobloc.

Après avoir chargé en mémoire locale les données, le module appel la fonction **HWMdInterMbRefinement()**. Cette fonction se charge en premier lieu de mettre à jour les divers paramètres ainsi qu'aiguiller le flot de données vers les bons tampons de travail. Par la suite, la fonction **HWMeRefineFracPixel()** est appelée afin d'effectuer une recherche au demi-pixel. Cette fonction charge tout d'abord la composante de luminance du macrobloc de référence. Nous avons donc défini une cache pour la référence nommée *sp_ref_mb_luma* ayant une taille de 486 octets. Le module effectue un premier SAD afin de comparer la référence trouver par l'IME et le bloc à encoder. Ce résultat servira de base. Par la suite, le module doit créer une interpolation au demi-pixel. Le tampon de 486 octets dédié aux références contient trois lignes et trois colonnes de plus afin de contenir l'information de bord de ces macroblocs. Le filtre d'interpolation, représenté par les fonctions **McHorVer02_c()** et **McHorVer20_c()**, a besoin de six pixels dont trois se retrouve à l'extérieur du macrobloc. Ce filtre est effectué à l'horizontale et vertical sur tous les pixels du macrobloc. Le filtre donne deux interpolations pour chaque orientation. Un déplacement vers la gauche, vers la droite, vers le haut et vers le bas. Par la suite la fonction **HWMeRefineFracPixel()** effectue le SAD sur les quatre interpolations (gauche, droit, bas, haut) représentant le demi-déplacement et analyse le résultat de chacun. Finalement, elle détermine la meilleure estimation tout en sauvegardant la direction du déplacement. La figure 3.17 illustre le processus de filtrage pour un déplacement à l'horizontale, alors que le listage 3.6 représente la pondération utilisée.

```
const uint32_t R3   = p1 + p6;
const uint32_t R2   = p2 + p5;
const uint32_t R1   = p3 + p4;
return (R3 - ((R2 << 2) + R2) + (R1 << 4) + (R1 << 2));
```

Listing 3.6 Interpolation d'un demi-pixel

Par la suite, le module procède au traitement au quart de pixel à l'aide de la fonction **HWMeRefineQuarPixel()**. Utilisant le résultat trouvé précédemment, elle calcule une interpolation au quart dans les quatre directions (haut, bas, gauche, droite) à l'aide de la fonction filtre **PixelAvg_c()**. Ce filtre utilise les deux prédictions horizontales ou verticales

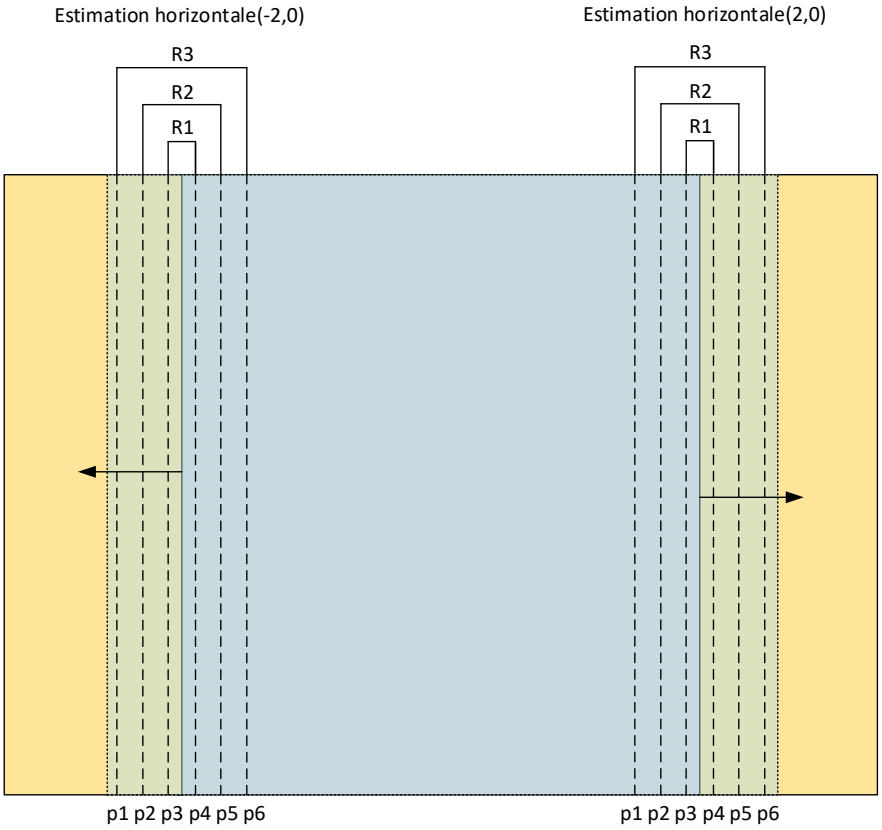



Figure 3.17 Exemple d'interpolation au demi-pixel pour les déplacements gauche et droite en bordure du macrobloc. Le même algorithme est utilisé pour les déplacements vers le haut et le bas

du demi-pixel. Afin de construire une interpolation d'un macrobloc de référence et véhiculer le raffinement entre les diverses fonctions du module, l'algorithme nécessite une zone de travail. Nous avons donc ajouté les tampons *pBufferInterPredMe* ayant quatre espaces de 640 octets. Cette espace est nécessaire afin de maintenir en mémoire les interpolations horizontales et verticales pour les estimations au demi-pixel ainsi que conserver deux valeurs d'interpolation au quart de pixel. L'algorithme évalue les quatre directions du déplacement et les deux tampons permettent de comparer deux estimations au quart de pixel afin de déterminer la meilleure progressivement. Le tampon est utilisé pour tous les types de macrobloc et doit maintenir à son maximum quatre interpolations de bloc 8x8. Le tampon provient directement de l'implémentation d'OpenH264 et celui-ci est un peu plus volumineux que nécessaire. Nous ne l'avons pas modifié pour la première version du module afin de ne pas introduire d'erreurS aux algorithmes de prédiction, car celle-ci est implémentée avec des index de déplacement fixe à travers le tampon. Afin de compacter l'utilisation mémoire, les dépla-

cements devront être dynamiques en fonction du type de bloc traité (8x8, 8x16, 16x8, 16x16).

Après le calcul d'interpolation, le module appel une des fonctions **WelsCopy*()**. Une de ces fonctions, dépendamment du type de bloc, copie la meilleure estimation déterminée par la fonction **HWMeRefineFracPixel()** vers un tampon de référence finale. Le module FME doit absolument produire le macrobloc en sortie, car l'algorithme ne fait pas seulement une simple copie de macrobloc de référence, mais créé un nouveau macrobloc de référence. Les fonctions d'estimation au demi et quart de pixel produisent un macrobloc unique par interpolation selon la fenêtre de recherche. Nous avons alors créé les tampons mémoires *sp_final_mb_luma* de 256 octets pour contenir la luminance et *sp_final_mb_chroma* de 128 octets contenant les deux chrominances.

Par la suite, la fonction **HWMcChroma_c()**, qui se charge de copier les chrominances du bloc de référence, est appelé par la fonction **HWMdInterMbRefinement()**. Pour se faire, elle utilise les vecteurs de déplacement prédit par la fonction **HWMeRefineFracPixel()** et copie les valeurs de chrominance du tampon de trame où pointent ces vecteurs vers le tampon *sp_final_mb_chroma*. Aucune interpolation de ces valeurs n'est effectuée par l'algorithme. Par la suite, un dernier SAD est fait entre la nouvelle référence (Y), les blocs de chrominance copiés et le bloc à encoder (YCbCr). Ce résultat et les vecteurs de déplacement sous forme de structure *FMEModuleOutputComStruct* sont retournés au module principal par un appel à la fonction **ModuleWrite()**. Finalement, la nouvelle référence interpolée est aussi fournie au module par fifo à l'aide de l'appel à un second **ModuleWrite()**.

Le module consomme un total de 3.8 Ko de mémoire cache.

### 3.4.11 Insertion des modules au flot de traitement

Afin d'être en mesure d'appeler les modules matériels IME et FME au lieu des fonctions de traitement logiciel, nous avons eu à contourner une limitation de la librairie de communication de SpaceStudio. Les fonctions de communications comme **ModuelRead()** et **ModuleWrite()** se doivent d'être appelées à l'intérieur d'un module SpaceStudio. Elles ne peuvent être utilisées dans le code externe comme c'est le cas avec la librairie OpenH264. Nous avons alors défini de nouvelles fonctions sous le module **MainEncoderTask** qui prennent les mêmes paramètres que les fonctions logiciels. Les fonctions de transfert **HWIMEModule()** et **HWFMEModule()**.

Afin que la librairie ait accès à ces fonctions, on doit être en mesure d'accéder à l'objet qui représente le module principal. Nous avons alors créé un patron de type singleton à l'intérieur du module et avons ajouté ses fichiers d'en-tête aux fichiers « .cpp » où se trouve l'appel aux fonctions **WelsMotionEstimateSearch()** et **WelsMdInterMbRefinement()**. Par la suite, la communication aux modules matériels se fait par l'appel de sa fonction respective. Le listage 3.7 illustre l'appel vers le module FME inséré dans les fichiers de la librairie OpenH264 associé au module principal. L'utilisation de pointeur dans ces cas n'est pas problématique étant donné que le module principal restera toujours logiciel. Nous avons utilisé ce même patron pour le transfert des trames vers la mémoire partagée mentionnée à la section 3.4.9.

```
#ifdef CONF_FME_SOFTWARE
        WelsMdInterMbRefinement (pEncCtx, pWelsMd, pCurMb, pMbCache);
#else
        MainEncoderTask::getInstance("SINGLETON")->HWFMEModule(pEncCtx->
            pCurDqLayer,  pWelsMd, pCurMb, pMbCache);
#endif
```

Listing 3.7 Exemple de l'appel du module matériel FME à partir de la librairie

Les fonctions de transfert développé pour l'appel des modules ont aussi une seconde fonctionnalité. Elles ont la nécessité de traduire les valeurs des structures de l'espace d'adressage global vers l'espace d'adressage local des modules matériels et transférer ces valeurs dans les structures spécialisées pour chacun des modules.

Pour le module IME et FME, tous les pointeurs faisant référence à de la mémoire RAM (*pData*, *pbuffer*, etc.) ont été traduits vers l'espace d'adressage des modules matériels. C'est le cas du pointeur *pRefMb* qui représente la trame à encoder et *pEncMb* la trame de référence. Ce dont nous avons besoin afin de retrouver les bons macroblocs dans les mémoires partagés est de calculer le déplacement dans l'image étant donné que toutes les images sont écrites à l'adresse 0x0 dans les tampons partagés. Par exemple, afin de calculer la référence on prend le pointeur du macrobloc de référence *pMe->pRefMb* et on le soustrait à la racine du tampon mémoire où il réside *pCurDqLayer->pRefPic->pBuffer*. La même approche est utilisée pour le macrobloc à encoder. Tout peut être traduit de cette façon en utilisant la racine de leur tampon mémoire.

Pour le reste les donner ont été copier à l'aide de la fonction **memcpy** vers les structures définie pour chaque module. Afin de simplifier le processus de transfert de données, la structure *ME*, qui contient les détails du macrobloc spécialisé par les IME et FME, et sa version originale du module principal ont été alignées l'une avec l'autre. C'est-à-dire que les paramètres utilisés par la structure *ME* réduite ont été regroupés et ordonnés de la même façon sous la structure *ME* originale. Ceci permet alors d'effectuer un **memcpy** de la grandeur de la mini-structure pour copier tous les paramètres nécessaire. Toutefois, dans le cas du module FME, nous avons neuf structures *ME* additionnelles à copier. Or, il est impossible de les copier en une seule opération, car la structure *ME* du module principal possède plus d'information que celle des modules matériels. Ceci causerait une corruption de donnée. Par conséquent, nous avons créé deux fonctions : 1) la fonction **mapSFME_MVComponentUnitStruct()** afin de transférer vers le module FME et 2) la fonction **mapSWelsMEStruct()** transférée vers la structure *ME* de la librairie OpenH264. Pour les paramètres sans aucune structure, on utilise la simple assignation vers la structure d'entrée du module.

## 3.5 Architecture finales du système

La figure 3.18 représente l'architecture finale du système. Les sections bleues illustrent les modules logiciels alors que les vertes représentent les modules matériels. Les zones grises représentent les mémoires du système ayant été ajouté afin de faire le lien entre le matériel et le logiciel (CPU/mémoire principal). La section « FrameBuffer » fait référence aux fonctions **sendFrameToHWFrameBuffer()** et **updateHWReferenceFrameBuffer()** développée afin d'assurer une cohérence entre la mémoire des modules et la mémoire centrale. Les mémoires et fonctions au banc de test y sont aussi illustrées.

Ceci complète l'atteinte des points 1) (Démonstration de la faisabilité de développer un encodeur H.264 à partir d'une référence logicielle utilisée en entrée d'une d'un processus de codesign logiciel/matériel) et 2) (proposition d'étapes d'une exploration architecturale afin de transférer un code logiciel vers une implémentation matérielle) des contributions mentionnées à la section 1.3.

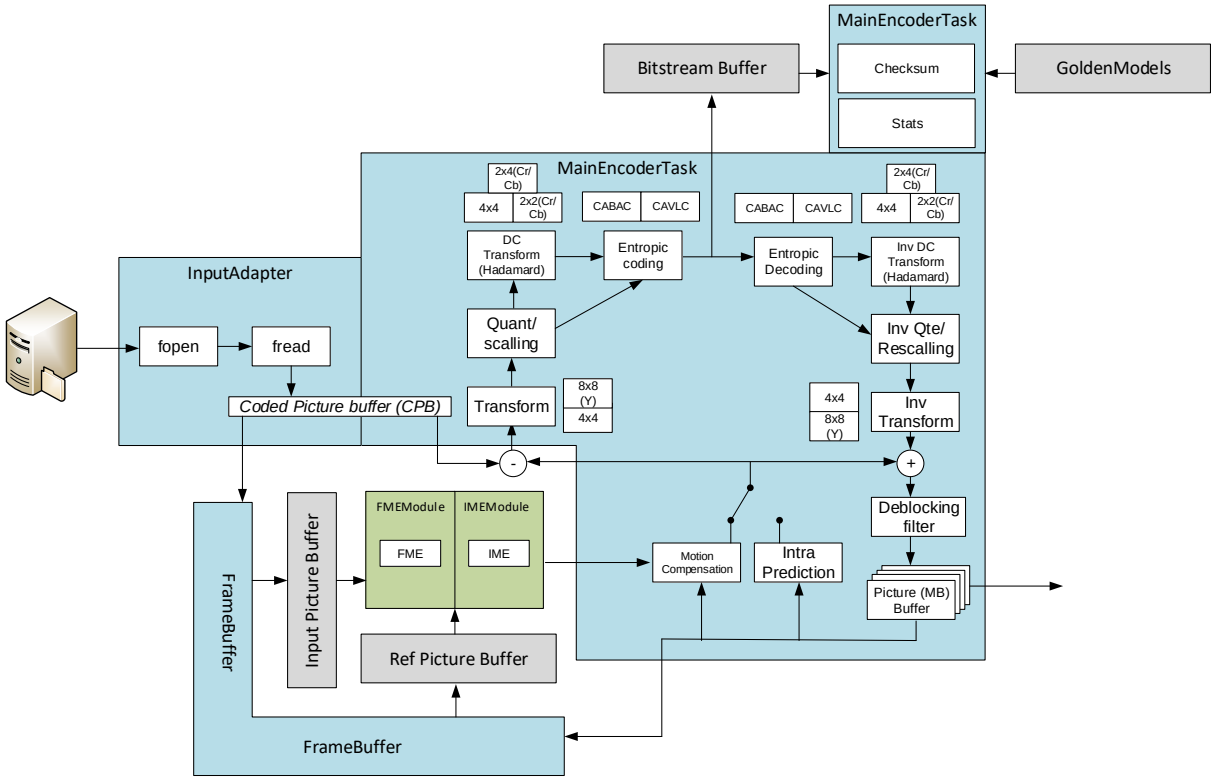



Figure 3.18 Architecture finale du système

# CHAPITRE 4 RÉSULTATS THÉORIQUES ET EXPÉRIMENTAUX

Ce chapitre présentera les performances de l'encodeur s'exécutant sur les trois plateformes développées. Soit le GoldenModel, la plateforme ARM logiciel et matériel. Ces résultats seront suivis d'une discussion pour ensuite terminer avec une discussion sur les limitations de la solution proposée.

## 4.1 Plateformes

Toutes ces plateformes ont été développées à l'aide de la version 2.8 de SpaceStudio.

### Golden Model (x86)

Le modèle de référence (GoldenModel) a été développé et exécuté sur une plateforme Intel Core i7 4700HQ ayant quatre coeurs de 2.4Ghz et ayant 8Go DDR3 de mémoire vive. Celle-ci utilisant le système d'exploitation Windows 10.

### ARM Cortex A9 (SW)

La configuration purement logicielle de notre système a été exécutée sur la plateforme virtuelle Zynq. Celle-ci comprend un processeur ARM Cortex A9 à doubles coeurs cadencés à 666Mhz et ayant 512 Mo de mémoire vive. Cette plateforme utilise Linux Digilent 3.10.

### ARM Cortex A9 (HW)

La configuration purement logicielle de notre système a été exécutée sur la plateforme virtuelle Zynq. Celle-ci comprenant un processeur ARM Cortex A9 à doubles coeurs cadencés à 666Mhz et ayant 512Mo de mémoire vive. Cette plateforme utilise Linux Digilent 3.10. À celle-ci nous avons joint nos coprocesseurs spécialisés IME et FME.

## 4.2 Résultats

Les tables suivantes représentent les performances d'encodage pour les trois plateformes développées exécuter sur trois types de vidéo conçu afin de solliciter les algorithmes d'estimation du mouvement de façon différente. Les colonnes min, max et moy représentent le temps en microseconde pour un appel du module. Les colonnes « usage » (min et max) représentent le

temps d'utilisation d'un module en fonction du temps pour encoder une trame complète. La colonne « calls » représente le nombre d'appels effectués vers ce module pour l'encodage d'une trame complète. Le temps d'exécution des modules matériel n'est pas comptabilisé dans ces résultats. Nous considérons leurs exécutions prenant un seul cycle lors des simulations. Afin qu'ils aient un temps représentatif, on doit les synthétiser et rétroanoté la simulation.

La table 4.1 représente les performances des encodeurs pour l'encodage d'une vidéo ayant des scènes statiques et des couleurs uniformes. Il s'agit d'une scène de discussion entre deux personnes avec quelques changements de point de vue. Cette vidéo sollicite légèrement les algorithmes d'estimation du mouvement.

Tableau 4.1 Encodage d'une vidéo ayant des scènes statique et de couleur uniforme

| Système | fps | FME ($\mu s$) | | | | | | IME ($\mu s$) | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | min | max | moy | usage min | usage max | calls | min | max | moy | usage min | usage max | calls |
| Golden Model (x86) | 145.44 | 55 | 123 | 65 | 2.69% | 54.15% | 43 | 1 | 34 | 5 | 0.05% | 35.19% | 57 |
| ARM Cortex A9 (SW) | 41.76 | 1 | 622 | 157 | 0.15% | 66.10% | 43 | 1 | 287 | 66 | 0.21% | 58.70% | 57 |
| ARM Cortex A9 (HW) | 32.72 | 13 | 667 | 199 | 1.26% | 59.26% | 43 | 1 | 687 | 155 | 0.12% | 54.43% | 57 |

La table 4.2 représente les performances des encodeurs pour l'encodage d'une vidéo ayant des scènes dynamiques et des couleurs uniformes. Il s'agit d'une scène d'un jeu vidéo ou le personnage esquive rapidement les attaques d'ennemie. Cette vidéo sollicite beaucoup les algorithmes d'estimation du mouvement, mais dus à l'uniformité des couleurs des macroblocs, une équivalence peut être rapidement trouvée.

Tableau 4.2 Encodage d'une vidéo ayant des scènes dynamiques et de couleur uniforme

| Système | fps | FME ($\mu s$) | | | | | | IME ($\mu s$) | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | min | max | moy | usage min | usage max | calls | min | max | moy | usage min | usage max | calls |
| Golden Model (x86) | 38.26 | 61 | 84 | 76 | 22.49% | 51.73% | 152 | 4 | 14 | 6 | 2.57% | 21.60% | 420 |
| ARM Cortex A9 (SW) | 10.68 | 56 | 242 | 156 | 9.30% | 44.17% | 152 | 11 | 123 | 57 | 2.66% | 51.86% | 423 |
| ARM Cortex A9 (HW) | 6.84 | 93 | 299 | 205 | 7.38% | 41.46% | 152 | 11 | 211 | 156 | 18.62% | 59.79% | 421 |

La table 4.3 représente les performances des encodeurs pour l'encodage d'une vidéo ayant des scènes dynamiques et des couleurs variées. Il s'agit d'une scène d'une personne qui court dans un marché. Cette vidéo sollicite beaucoup les algorithmes d'estimation du mouvement, car elle contient beaucoup de transition et les couleurs variées forcent les algorithmes d'analyse aux sous pixel à trouver la meilleure estimation.

Ceci complète l'atteinte du point 3) (Expérimentation du développement d'un SoC encodeur

Tableau 4.3 Encodage d'une vidéo ayant des scènes dynamiques et de couleur variées

| Système | fps | FME ($\mu s$) | | | | | | IME ($\mu s$) | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | min | max | moy | usage min | usage max | calls | min | max | moy | usage min | usage max | calls |
| Golden Model (x86) | 43.51 | 57 | 110 | 76 | 1.56% | 52.73% | 130 | 3 | 33 | 6 | 2.55% | 34.76% | 439 |
| ARM Cortex A9 (SW) | 11.22 | 32 | 285 | 155 | 1% | 58.26% | 130 | 2 | 185 | 58 | 0.40% | 55.89% | 441 |
| ARM Cortex A9 (HW) | 7.09 | 100 | 348 | 200 | 1% | 53.47% | 130 | 61 | 249 | 155 | 9.48% | 69.00% | 439 |

H.264 logiciel/matériel s'exécutant sur plateforme virtuelle du FPGA Zynq de Xilinx. Cette même plateforme virtuelle servant à faire la validation de l'exploration architecturale) des contributions mentionnées à la section 1.3.

## 4.3 Discussion

Tout d'abord, on remarque évidemment que l'encodeur développé afin de servir de modèle de référence est en moyenne 3.5x plus rapide que les encodeurs développés sur ARM. Ceci est tout à fait normal lorsque l'on considère que le processeur de la plateforme de référence possède le double des cœurs et une cadence 3.5x plus rapide. On peut aussi remarquer que le module FME semble un peu plus lourd à exécuter sur la plateforme x86 que sur ARM pour la seconde vidéo. Nous expliquons mal ce phénomène. L'algorithme FME exige l'interpolation de valeurs par la lecture des pixels voisins au macrobloc et cela peut être lourd sur la cache ou le système de prédiction de branchement du processeur x86, mais cela devrait être visible sur tous les tests et pas uniquement pour la seconde vidéo.

Le nombre d'appels aux algorithmes IME et FME est fortement similaire entre les diverses plateformes. Nous avons une variation de 2 à 3 appels pour le module IME lors de l'encodage des vidéos dynamiques qui peut être relié à la façon dont les opérations sont effectuées sur les processeurs x86 et ARM. Ceci n'influence pas les appels au module FME, car ce n'est pas tous les macroblocs analysés par l'IME qui nécessite un raffinement. Le résultat des encodages sur le processeur ARM respecte celui du modèle de référence. Ceci élimine la possibilité d'une différence d'implémentation.

Par la suite, on remarque que les coprocesseurs matériels développés semblent ralentir le système plutôt que l'accélérer. Pour l'encodage de la première vidéo, on remarque une perte de 9 images par seconde alors que pour les deux autres on remarque une perte de 4 images par secondes comparativement à la version complètement logicielle. Le module FME prend en moyenne $50\mu s$ de plus par macrobloc que la version logiciel sur ARM. Alors que le module IME pour sa part prend $100\mu s$ de plus. On remarque que ces délais sont stables pour les trois

types de vidéo et ne dépendent pas du type de macrobloc analysé par l'algorithme. Ceci est signe que nous avons augmenté le coût des communications. À ce stade, l'unique différence entre les modèles matériels et logiciel est l'ajout des communications, car l'exécution des modules matériels est faite sans annotations temporelles étant donné que nous ne les avons pas synthétisées. Les résultats comptabilisés nous indiquent ainsi que nos communications sont sous-optimales et que celles-ci ralentissent notre système. Elles causent des délais d'exécution plus longs que la version logicielle et absorbent ainsi tout le gain de la migration des algorithmes vers des modules matériel.

Afin de situer nos performances, voici le résultat d'encodage de deux systèmes développés à la main à l'aide de langage HDL. Les auteurs de l'article [14] on produit un encodeur de type ASIC capable d'encoder des images de résolutions 640x480 et 1280x720 à 30 images par seconde (fps) en utilisant 581 mw. Les auteurs de l'article [7] ont quant à eux développé un encodeur sur FPGA capable de traiter 115 Mpixel/sec. C'est-à-dire capable d'encoder à 1080i et 720p à 30 images par seconde.

Il est évident que nous ne pouvons nous comparer directement à ces résultats. Par contre, la force de notre approche est le taux de productivité. Il existe peu d'implémentation complète de la norme H.264 à l'aide de méthodologie de haut niveau, mais l'article [26] précédemment mentionné peut se comparer à notre travail malgré qu'ils implémentent un décodeur. Le décodeur produit au moyen d'une approche à l'aide d'un outil d'HLS sans aucune accélération introduite manuelle est capable de décoder 100 images par seconde en format QCIF et 5 image par secondes en format VGA tel qu'illustré par les figures 4.1 et 4.2 tirées de l'article [26]. La ligne bleue représente leur implémentation matérielle alors que la ligne CPU représente leur logiciel de références sur processeur. Comparativement à nous, nous obtenons 7 à 30 images par secondes pour le format 320x180. Ceci nous situe relativement dans les mêmes eaux si l'on considère que nous n'avons appliqué aucune accélération et qu'un encodeur possède une plus grande complexité de calcul. Nous croyons réaliste la possibilité d'atteindre des performances semblable dans le futur.

Ceci dit, il est aussi intéressant de comparer leur méthodologie à la nôtre. Lors de leur profilage, ils se sont limités à ce que HLS pouvait leur donner comme information et ont optimisé jusqu'à l'obtention de gain de performance. Pour notre part, nous avons utilisé Valgrin et Pareon nous donnant un meilleur aperçu des optimisations possible. Dans leurs travaux futurs, ils ont même suggéré l'utilisation d'un outil qui peut générer un graphe

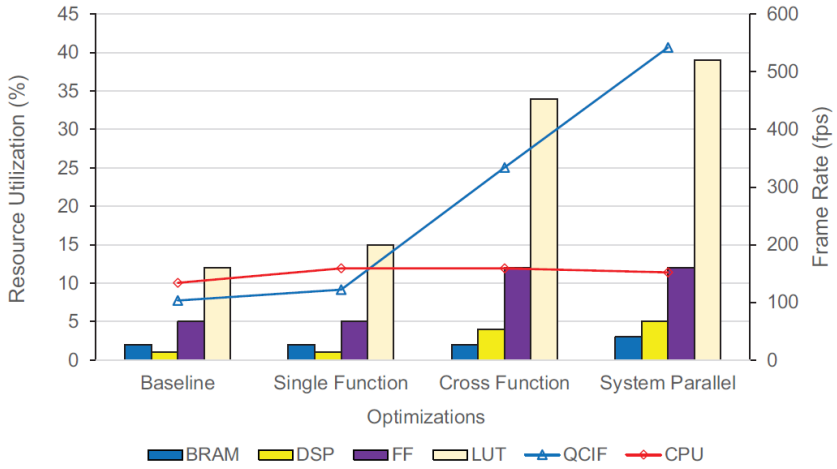



Figure 4.1 Article[26], décodage des trames de résolution QCIF (176x144)

d'appel. De plus, leur approche cible l'exploration de microarchiotecture pour un système complètement matériel. Ce que nous apportons est l'aspect macroarchoitecture à l'aide du codesign ne tenant pas pour acquis qu'une approche complètement matérielle est toujours souhaitable. Les optimisations HLS mentionnées par les auteurs pourraient être intégrées à la partie microarchitecture de notre méthodologie après le partitionnement effectué.

## 4.4 Limitations de la solution proposée

Suite aux résultats de notre système, il est nécessaire de faire le point sur les diverses limitations de cette première implémentation qui a tout de même nécessité un bon effort de travail. Cette section sera donc dédiée au point 4) de nos contributions élaborées à la section 1.3. Soit l'identification des difficultés rencontrées lors du développement de l'encodeur et de la méthodologie.

### 4.4.1 Système H.264

Une des premières limitations du système développé à l'aide de la méthodologie est le doublage des trames à encoder et de référence. Nous avons dans la mémoire DDR du processeur ARM sous Linux les trames, mais devons les partager aux modules matériels. Or ceux-ci n'ont pas accès à cette zone mémoire. Ceci est causé par l'utilisation de « heap » à travers les appels système tels que « malloc » et que cette zone protégée par le système d'exploitation n'est pas accessible. Étant donnée la quantité de manipulation mémoire à l'intérieur de l'encodeur, tout transférer ces opérations vers la mémoire partagée en utilisant les appels de

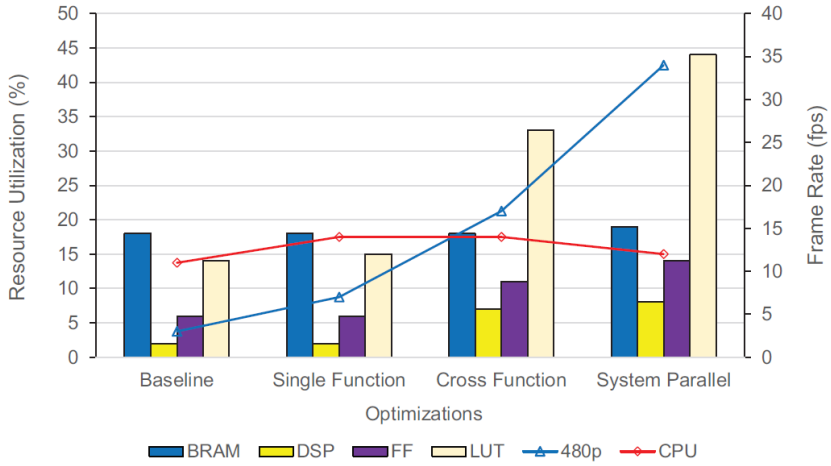



Figure 4.2 Article[26], décodage des trames de résolution 640x480

la SpaceLib serait une tâche colossale et propice aux erreurs lorsque fait manuellement. Une autre limitation due à l'architecture mémoire est le chargement des mémoires partagé. Celui-ci se fait à chaque début d'encodage de nouvelle trame et prend plusieurs microsecondes. Soit de $55\mu s$ à $71\mu s$ tout dépendants le module et le type d'image. Cette architecture cause alors des délais mémoires qui ne seraient pas présents dans un modèle de mémoire unifiée.

Une seconde limitation est reliée à l'architecture de base utilisée. L'architecture ARM Cortex A9 et Linux sur laquelle nous avons développé n'est pas l'architecture cible. Le processeur est non représentatif du design final souhaité par GrassValley. Nous avons utilisé celle-ci afin de nous éviter la migration de toute la librairie vers le système d'exploitation temps réel microC (uC).

Une autre limitation est que le système ne traite présentement aucunement les macroblocs de façon parallèle. Nous avons complètement désactivé le parallélisme de la librairie lors de la migration vers SpaceStudio pour éliminer les dépendances aux librairies tierces de parallélisation (pthread, Win32). Le flot implémenté à l'aide de la mémoire partagée ne tient aucunement compte d'une possibilité de parallélisme. Les modules matériels ont été conçus pour travailler de façons atomiques. Il ne partage aucune donnée entre chaque appel. Ils travaillent de façon unitaire sur leur macrobloc et peuvent être dupliqués afin que le système soit en mesure de travailler sur plusieurs macroblocs en parallèle. Afin d'exploiter cette possibilité de parallélisme, il serait par contre nécessaire de modifier le flot de données de l'application afin de tenir compte du partage des tampons mémoires entre plusieurs coprocesseurs.

L'utilisation de la librairie OpenH264 a aussi apporté son lot de problèmes. Tout d'abord, le temps de familiarisation fut très long principalement à cause du manque d'information sur celle-ci et de l'absence de commentaires dans le code source. Comprendre une section de l'algorithme demandait de lire attentivement le code et de l'exécuter plusieurs fois. De plus, la nomenclature ambivalente des symboles (**update_P8x16_motion_info** vs **UpdateP16x8MotionInfo**) et certaines optimisations précoces effectuées afin d'accélérer l'application rendait la lecture plus compliquée que nécessaire. Certaines décisions nous ont aussi amenés à nous questionner sur la maturité de la librairie. Une de ces aberrations fut l'utilisation de la table de coût (MvdCost) qui contenait le double des données nécessaire dans le but d'accélérer un calcul. Celui-ci étant relativement triviale pour un processeur moderne.

Finalement nous ne nous sommes pas concentrés sur le flot de données. Les opérations de lecture mémoire et de mise à jour des mémoires locales des modules matériels sont sous-optimales. Dans le cadre du module IME, nous chargeons cinq macroblocs afin de déterminer la zone de recherche, mais nous n'utilisons pas le principe de fenêtre de recherche (discuté à la section 3.4.7 ) pour la suite. Chaque déplacement charge un nouveau macrobloc de 256 pixels. L'utilisation de fenêtre de recherche permettrait d'amortir les délais de lectures mémoires par la réutilisation des blocs voisins pour les prochains macroblocs.

### 4.4.2 Méthodologie

Comme première approche, la méthodologie reste à très haut niveau et ne couvre pas les détails d'une implémentation matérielle. Nous avons souligné les embûches liées à la migration d'un code logiciel vers une plateforme matérielle sans entrer dans le détail de l'implémentation du flot de données qui indique les mémoires et chemins à utiliser. Or, afin de limiter l'utilisation mémoire et accélérer les transferts de donnée, on doit modifier le logiciel pour refléter ces limitations.

La migration d'une section de code vers de la logique numérique est aussi une transition d'idiome de développement. Comme vu à l'aide de la libraire OpenH264, le développement logiciel n'a pas à se soucier de la localité de la mémoire. Les appels systèmes fournissent les ressources au développeur et les divers niveaux de mémoire des systèmes modernes s'assure que les données soit à quelques cycles de disponibilité. Beaucoup de logique est dédiée à cette abstraction de localité dans les systèmes moderne (cpu, MMU, cache). Or, lorsqu'on transfère un code en matériel, ces abstractions tombent et nous devons gérer la localité manuellement

et le flot de données se doit d'être altéré. Ce problème a déjà été identifié par les fabricants et ils offrent maintenant des bus spécialisés ayant une cohérence de cache avec la mémoire centrale. Dans le cas du Zynq ce bus se nomme ACP. Cette piste n'a pas été explorée étant donné que la plateforme demander par Grass Valley ne devait pas inclure une puce de type Zynq.

**Profilage**

En ce qui concerne la première étape du profilage, nous avons convenu que notre approche n'a pas été optimale. En effet, exécuter les outils d'analyse Valgrin et Pareon à l'extérieur de la plateforme virtuelle n'est pas la solution idéale. Nous cherchons à évaluer les performances des algorithmes sur la plateforme cible, dans notre cas sous un processeur ARM Cortex A9, mais l'analyse sur un processeur d'un autre type peut offrir des performances différentes et peut influencer les décisions sur les modules à créer.

Lors de notre seconde passe de profilage effectué sous la plateforme virtuelle à l'aide de nos sondes personnalisées, nous avons effectué nos recherches sur une architecture très différente de celle initialement planifiée. Les temps calculer pour l'exécution des fonctions ne sont pas représentatifs du processeur final. Le processeur ARM est très puissant et analyser le temps d'exécution sur celui-ci ajoute un biais à notre analyse. En effet, une fonction de l'ordre des microsecondes ne laisse pas beaucoup de latitude si l'on veut modulariser. Or, ces délais peuvent être plus longs avec une unité de contrôle différente rendant la modularisation plus inintéressante. Par contre, le temps des communications peut aussi changer et ainsi rendre le gain similaire au processeur ARM. Il serait tout de même préférable de profiler l'application sur une plateforme virtuelle cible afin de réduire l'incertitude.

En ce qui concerne la plateforme virtuelle utilisée, nous y avons identifié une anomalie en lien avec le calcul des performances des moyens de communication. En effet, les métriques accumulées lors de la phase d'exécution sur la plateforme virtuelle ne correspondent pas aux tests de communication faits préalablement. Lorsque nous exécutons Linux sous la plateforme QEMU, il semble y avoir un surcoût considérable des communications par fifo. Ces délais de plusieurs microsecondes rendent la création des modules matériels pratiquement inutile lorsque celui-ci s'exécute en dessous de la milliseconde en logiciel. Les communications prennent alors plus de temps que l'exécution de l'algorithme en format logiciel.

SpaceStudio nous offre une option de monitorage très pratique lorsqu'on utilise les opérations

de la SpaceLib. Ceci nous indique, entre autres, les délais de communication entre les divers modules et les périphériques. Par contre, nous n'avons pas d'information sur l'exécution interne des modules. Si l'on s'aperçoit qu'un module est lent, nous n'avons aucun moyen d'identifier la zone problématique. Les outils de profilage peuvent nous indiquer diverses métriques sur le temps d'exécution, le flot de contrôle, le nombre d'appels d'une fonction, la couverture, etc. À l'aide de ces métriques, nous sommes en mesure de mieux identifier les zones sollicitées du code.

**Modularisation d'un modèle système**

Lorsque nous voulons modulariser une zone de code d'un système, on se doit bien sûr de la modifier. Par contre, si nous voulons expérimenter plusieurs groupements de fonctions afin de produire différents modules matériels, ces modifications doivent être supprimées et le code original restauré. Sous la version 2.8 de SpaceStudio, il n'est pas possible d'évaluer plusieurs segmentations différentes sous un même projet. Il est cependant possible de produire plusieurs projets pour tester différentes modularisations, mais le code commun de ces projets doit résider ailleurs. Afin d'expérimenter différentes architectures, on se doit d'être en mesure de revenir dans l'état initial du système. Ceci permet la construction de modules seulement aux endroits voulus et permet de ne pas impacter la performance du système par l'ajout de communication. Que le modèle de base soit toujours disponible, et ce malgré toutes les diverses modifications imposées par l'implémentation d'une architecture. Ces limitations nous ont empêchés d'itérer sur diverse modularisation tout en conservent le système initial. Ceci fait référence à notre approche mentionnée à la section 3.4. Afin de valider une architecture SW/HW pour les modules FME et IME sans y avoir de communication, nous avons contourné le problème en ajoutant un appel conditionnel des modules matériels. Si la configuration inclut la macro HW, nous appelons le coprocesseur autrement, la section logiciel de la librairie est appelée.

Comme mentionnées à la section 3, diverses fonctions de rappel (callback) ont dû être développées afin de permettre au code de la librairie d'appeler les fonctions de la librairie SpaceLib. Ceci a occasionné une légère offuscation du code et la complexification de l'architecture. Ces appels de fonctions ont aussi eu un impact sur la performance de l'encodeur.

Un des plus gros problèmes rencontrés fut les modifications à apporter au flot de données. Lorsqu'on transfère un module vers le matériel, nous avons besoin de connaître les opérations mémoire en entrée/sortie (consommation) et les liens mémoire (identifier le lien producteur

consommateur) qu'il a ou aura avec le ou les modules logiciels, car SpaceStudio ne modélise par un environnement mémoire uniforme. Les liens producteurs consommateurs entre les zones de code sont invisibles aux outils de profilage traditionnels. Si l'on veut modifier l'architecture mémoire pour évaluer une nouvelle architecture logiciel/matériel, connaître ces liens est essentiels pour effectuer rapidement les opérations. Sans ce détail, la migration d'un module est beaucoup trop longue, car on doit trouver et modifier manuellement tous les échanges de données entre logiciel et matériel.

## 4.5 Réflexion sur l'automatisation du processus

Nous avons vu qu'une simple copie d'un algorithme logicielle vers le matériel introduit d'énormes latences et qu'il est nécessaire de modifier le flot de données afin de tenir compte de l'ajout de communication. L'ajout de tampon et la modification du code se font avec la connaissance de la consommation des données par l'application. Suite à cela, nous nous sommes demandé s'il était réellement possible d'automatiser ce processus de migration du flot de données par l'analyse de code et réduire le facteur de spécialisation nécessaire afin qu'un développeur soit en mesure de créer des architectures optimisées. Nous voyons quatre étapes afin d'y arriver.

Nous devons tous d'abord extraire le flot de données de l'application. Tout au long de cette recherche, nous avons mentionné la nécessité d'avoir celui-ci afin de comprendre la consommation de donnée du logiciel et ainsi effectuer les décisions architecturales. Faire ressortir ce flot manuellement est par contre trop coûteux. Pour le faire automatiquement, nous devons évaluer les données nécessaires afin de qualifier le flot de donné. Ces informations peuvent, entre autres, être les zones mémoires utilisées, les opérations de lecture et d'écriture mémoire, la section du code de haut niveau associé à ces opérations, la quantité de données véhiculées et le temps de vie d'une donnée. On retrouve ces informations par une analyse dynamique et statique sur la consommation mémoire de l'application.

Une fois ces données recueillies, on doit modéliser le flot de données. C'est-à-dire interpréter les données sur les opérations mémoires afin de développer un modèle abstrait de la consommation de donnée de l'application. Le modèle se doit de représenter, entre autres, les liens producteurs consommateurs entre les sections du code (fonctions, classes, modules, etc.), la bande passante, les patrons de consommations (lignes, carrée, sauts, etc.), etc. À cette étape, un logiciel de développement tel que SpaceStudio aurait l'information nécessaire pour per-

mettre à un développeur d'avoir accès au flot de données à un plus haut niveau et facilement effectuer les changements sur celui-ci. Grâce au modèle de haut niveau et ses liens avec le code applicatif, le logiciel de développement utilisé par le développeur serait en mesure de faire lui-même les changements au code réduisant ainsi le temps entre deux architectures mémoires. Le modèle servant de patron au compilateur pour produire le code nécessaire. Ceci permettrait la création de modèles mémoires pour logiciel lié à des modèles mémoires pour matériel développé manuellement.

La prochaine étape serait une recherche de modèle fait par un groupe de développeur afin d'identifier les modèles mémoires matériels optimaux pour les modèles mémoires logiciels. Ceci nous permettrait de classifier un modèle avec un type d'algorithme (convolution, compression, etc.) avec leur approche matériel optimisé. Au fur et à mesure de la construction de ces modèles, une base de données pourrait être chargée de ces modèles classifiés. Il serait possible à ce point de produire des gabarits de modèle permettant la modification optimale du code application si l'auteur connaît le type d'algorithme ou la classification du modèle mémoire logiciel.

Finalement, à l'aide d'une telle base de données, un algorithme d'intelligence artificielle pourrait être entraîné à différencier les modèles et produire d'elle-même les modifications au code lors de la compilation.

Nous croyons qu'il est possible d'automatiser ce type de transformation, mais nous avons à effectuer les recherches nécessaires pour compléter ces étapes. Le chapitre suivant tentera de démontrer la faisabilité d'une telle modélisation.

# CHAPITRE 5 AMÉLIORATION DU PROCESSUS

Dans ce mémoire, nous avons présenté une méthodologie de développement de système sur puce à l'aide d'une spécification exécutable. Bien que l'approche proposée par la méthodologie et les chemins pour y arriver aient été explicites, l'application de ceux-ci s'avère plus compliquée. La différence idiomatique entre le domaine logiciel et matériel rend la transition entre ces deux mondes très complexes. En effet, on se doit de modifier le flot de données d'une application afin que celui-ci soit optimal dans le domaine matériel. Or, nous n'avons pas les outils nécessaires afin de rendre cette étape plus rapide et simple. Cette transition doit ainsi s'effectuer manuellement. Le temps de développement nécessaire à la modification du code afin de refaire les chemins de données des modules nous ont causé des délais et ont produit une première itération plus lente que le modèle purement logiciel. Afin d'améliorer la méthodologie, le chemin de données doit nécessiter une attention plus importante pour augmenter la performance et la qualité des modules matériels.

Dans cette section, nous discuterons des suggestions et des avenues possibles afin de fournir de meilleurs outils aux développeurs sous SpaceStudio afin de rendre la méthodologie de développement inintéressante pour le futur. Certaines de ces discussions seront accompagnées de preuve de concept et de prototypage pouvant servir de contributions importantes pour le développement d'outils ESL en général, et plus particulièrement SpaceStudio.

## 5.1 Profilage

En ce qui concerne les outils de profilage disponibles, SpaceStudio offre actuellement un engin de monitorage, mais afin de réduire l'incertitude du profilage et la dépendance aux outils tierce, il serait préférable d'inclure un outil de profilage plus développé qui peut aussi être exécuté sur la plateforme virtuelle de SpaceStudio. Ceci permettrait entre autres d'y inclure le monitorage des délais reliés aux fonctions internes des modules s'exécutant sur cette plateforme. En effet, lors de notre recherche nous avons développé un outil afin d'obtenir les temps d'exécution de différentes sections du code applicatif et des modules matériels. Lors de l'analyse initiale des délais de communication, cet outil s'est avéré erroné et nous avons obtenu de mauvais résultats. Ces erreurs étaient dues au fait qu'il n'y a pas d'outils intégrés à SpaceStudio pour effectuer ces métriques et les méthodes de temps utilisées (Linux sous QEMU) ne sont pas fiables. Une autre avenue a été explorée afin d'utiliser une modélisation de périphérique nommé **simulation_timer**. Celui-ci utilise le temps SystemC lors de

simulation et utilise le chronomètre système lors de l'implémentation sur carte. Malgré tout, cela n'est que fonctionnel sous Linux. Il est primordial d'offrir des outils de chronométrage et monitorage multiplateforme afin de permettre une exploration architecturale sans restriction sur le système d'exploitation.

L'outil pourrait aussi offrir l'arbre d'appel d'un module comme celui que nous avons produit à l'aide du logiciel Valgrin. Cet outil est lui-même à source ouvert et il serait possible d'inclure ce cadriciel à SpaceStudio et ainsi développer un outil de profilage statique spécialisé pour le domaine.

## 5.2 Flot de données

Le développement du flot de données entre l'approche de conception logicielle et l'approche de conception matérielle est très différent. On ne peut faire un bloc matériel ayant de bonnes performances en utilisant directement un flot de données logiciel. Comme vues par nos tests à la section 2.4, les performances d'un module sont directement reliées aux communications effectuées. L'approche de type logiciel prend en compte que la mémoire est locale, que les opérations sont directement effectuées sur celle-ci et que les communications sont amorties par les modèles de cache sophistiqués. Dans le cas d'un module matériel, la mémoire n'est pas toujours partagée ou rapidement accessible. Dans ce cas, il existe des approches telles que « ligne buffer » et « scarcth pad » afin d'offrir des mémoires locales rapides et alléger le coût des communications.

Ceci dit, il y a peu d'avantages si l'on doit modifier la totalité de l'algorithme logiciel (traitement par macrobloc vers un traitement par pixel en continu (stream), mémoire local, ligne à délai, etc.), afin de créer un module matériel à l'aide d'une spécification exécutable. Ceci rend encore plus difficile le design au niveau système à partir d'une spécification logiciel. Un outil du type de SpaceStudio doit soit permettre les modifications de flot plus facilement ou les prendre en charge automatiquement lors de la transformation d'un module logiciel vers le domaine matériel.

Une grande partie du développement de l'encodeur a été utilisé afin de « plier » le code logiciel vers un comportement matériel et les séparer du module principal. Il a été nécessaire d'analyser l'utilisation mémoire du code à transférer afin de retrouver le modèle producteur/-consommateur et ainsi construire le code permettant le déplacement des données.

### 5.2.1 Adressage mémoire

Il existe deux types d'adressage possible pour un système comportant des coprocesseurs. Le premier consiste à isoler les coprocesseurs de façon à ce que chacun ait son propre espace d'adressage et mémoire centrale. Le second consiste en un adressage unifié où tous les coprocesseurs partagent la même mémoire centrale et le même adressage physique. Une adresse pour un coprocesseur correspond au même emplacement pour tous les autres coprocesseurs. La seconde approche est plus coûteuse en matériel, car elle nécessite plus de logique afin d'assurer la cohérence entre les divers coprocesseurs. Dans le contexte de notre méthodologie, l'adressage unifié nous permettrait d'utiliser les pointeurs du domaine logiciel à l'intérieur de la logique des modules matériels sans quelconque modification. La logique de cohérence se chargerait de réduire la barrière entre les idiomes logiciels et matériels.

Le choix de l'adressage mémoire se veut un choix architectural offert au concepteur, car, bien que l'approche UMA permet de simplifier le port de logiciel vers le matériel, celui-ci peut ne pas être nécessaire pour certaines architectures. De plus, ce type d'adressage est limité par la technologie. Par exemple, les puces ARM offrent la cohérence de cache par le bus « Accelerator Coherency Port » (ACP) et IBM offre la même approche avec leur processeur POWER8 à l'aide du « Coherent Accelerator Processor Interface » (CAPI). Le choix d'utiliser une technologie UMA force une plateforme cible et celles-ci devront être modélisées par SpaceStudio afin de permettre une exploration architecturale comprenant un type d'adressage unifié.

Présentement SpaceStudio 2.8 modélise uniquement la première approche. Les modules s'exécutant sous Linux ne partagent pas le même espace d'adressage que les modules matériels et il n'existe aucune facilité pour le faire SpaceStudio 2.8. Ces modules sont alors contraints d'utiliser des mémoires locales et les facilités de la librairie SpaceLib (ModuleRead/DeviceWrite et DeviceRead/Write) pour communiquer avec les modules matériels. Ceci nous oblige donc à transformer tous les accès mémoire du logiciel de référence par des appels vers la librairie. Dans un langage comme le C/C++, ces accès prennent la forme de déréférencements de pointeur. Pour un petit module, on peut remplacer ceux-ci par un appel de fonction plutôt rapidement. On doit par contre manuellement trouver tous les allocations et déréférencements de pointeur, identifier les adresses mémoires utilisées, lier ces opérations aux autres sections de code dépendantes de ces opérations, etc. Dans le cas d'une très grande librairie comme nous avons avec OpenH264, transformer un tel système est pratiquement impossible. Nous n'avons donc pas été en mesure de modifier le code complet de la librairie afin d'utiliser les abstractions de SpaceStudio pour optimiser le flot de données. Les sections 5.2.2 et 5.2.3

proposent des solutions afin de rendre cette approche plus simple.

### 5.2.2 Analyse statique

Une première recommandation est l'ajout d'une étape à l'analyseur de communication Clang, déjà utilisé sous SpaceStudio, afin de recueillir les accès mémoire (*ptr, memcopy, etc.) et les demandes de mémoire (malloc, new, etc ). Ceci permettrait, lors de création de modules matériels utilisant des données partagées ou de la mémoire dynamique, de modifier les accès mémoires pour les remplacer par l'abstraction de la SpaceLib de SpaceStudio (DeviceRead/-Write). Cette approche permettrait alors la migration automatique d'un module en format matériel. Les allocations mémoires devraient être transformées en mémoire statique alors que les accès mémoire comme les déréférencements de pointeur seraient transformés en Device-Read/DeviceWrite.

Les accès mémoire sous le langage C/C++ sont habituellement interprétés par un OS. Ces accès sont implicites aux constructions du langage. Par exemple, le listage 5.1 représente l'exemple idéal de lecture et écriture de mémoire à travers un pointeur, mais les constructions peuvent être plus complexes comme démontré par le listage 5.2.

```
function_read(int* pData, int data)
{
        data = *pData; // lecture memoire
        data = pData[0]; // lecture memoire
}
function_write(int* pData, int data)
{
        *pData = data; // ecriture memoire
         pData[0] = data; // ecriture memoire
}
```

Listing 5.1 Exemple d'opération mémoire sous le langage C/C++

```
function_write(int* pData, int data)
{
        pData[offset1 + offset2 << 1] = data;
        *(pData + 34 * someOffset) = data
}
```

Listing 5.2 Exemple d'opération mémoire complexe sous le langage C/C++

Ces opérations mémoire devraient être abstraites par leurs appels respectifs vers la librairie d'abstraction SpaceLib comme démontré par le listage 5.3.

```
function_read(int* pData, int data)
{
        DeviceRead(RAM\_ID, pData, data, 4); : lecture memoire
}
function_write(int* pData, int data)
{
        DeviceWrite(RAM\_ID, pData, data, 4); : ecriture memoire
}
```

Listing 5.3 Exemple d'abstraction mémoire à l'aide de la librairie SpaceLib sous le langage C/C++

Cette approche serait parfaite pour créer une zone mémoire unifiée pour les modules matériels et logiciels. Chaque allocation, lecture et écriture serait faite dans la zone unifiée définie par une architecture sous SpaceStudio. Celle-ci peut être configurée par l'utilisateur pour être un périphérique du type DDR ou BRAM. De plus, lorsque l'application utilise les allocations dynamiques à profusion (création de tampons, modification des grandeurs des tampons, etc.) il serait nécessaire de les abstraire à l'aide de déclarations statiques pour refléter la réalité du domaine matériel.

Cependant, l'analyse statique à l'aide de Clang ne donne pas toute l'information nécessaire. Lors d'une allocation mémoire, les paramètres d'allocation ne peuvent qu'être déterminés à l'exécution. Par exemple, le tampon mémoire pour contenir une image de l'application d'encodage. Il en va de même pour les adresses mémoires. Celles-ci peuvent être modifiées et calculées à l'exécution. Une analyse statique ne peut donc pas déterminer préalablement où

se fait la lecture/écriture mémoire. Il est donc impossible d'associer un appel à la librairie Spacelib par une communication vers un module ou périphérique précis.

Cette approche offre déjà une facilité comparativement à une approche complètement manuelle. Il suffit par la suite de guider l'utilisateur pour remplir le reste de l'information. Par contre, ceci implique que le développeur possède une bonne connaissance du flot de donnée.

### 5.2.3 Analyse dynamique (PIN)

Le profilage présenté jusqu'à maintenant est principalement dirigé vers l'analyse statique du chemin de contrôle et de données. À l'aide de Clang on peut statiquement identifier les opérations mémoires d'un code C/C++, mais les informations importantes telles que la location et la quantité de données lue ou écrite ne peuvent pas toujours être déterminées statiquement. Tout au long du chapitre 3, nous avons mentionné à plusieurs reprises que nous avons limité les modifications au flot de données afin de simplifier la migration vers le matériel. La difficulté d'identifier rapidement le flot de données était une de ces raisons. Les liens mémoires entre les diverses sections du code sont invisibles aux outils de profilage statique lors de l'utilisation de mémoire dynamique. De plus, une donnée peut porter plusieurs noms durant son traitement i.e. : ptrData, ptrImage, etc. Afin d'avoir un meilleur bénéfice à l'ajout de modules matériels, nous devons être en mesure de modifier facilement le flot de données. Sans l'aide d'un outil, cette tâche est pratiquement impossible avec une application à grande échelle. Pour faire ressortir ce flot, il est nécessaire d'effectuer un profilage dynamique.

Afin de complètement qualifier une opération mémoire, nous avons besoin de plusieurs paramètres. Tout d'abord, nous avons besoin de la location (adresse mémoire) où est effectuée l'opération mémoire. Cette donnée est primordiale pour lier deux opérations mémoire de deux sections de code différent. En effet, si une fonction A écrit à une adresse 0x0001 et qu'une autre fonction B lie à l'adresse 0x0001 nous avons un lien producteur (fonction A) et consommateur (fonction B). Nous devons bien sûr tenir compte de la préséance des opérations. Si la fonction B est appelée avant la fonction A, on ne peut pas conclure un lien producteur/consommateur entre ces deux opérations. L'analyse statique du flot d'appels reste donc nécessaire pour établir la préséance des opérations.

Nous devons ensuite identifier le type d'opération (lecture, écriture). Ceci nous permettra de connaître le sens du flot des données et aussi d'identifier le lien producteur/consommateur.

Par la suite, on se doit de connaître la quantité d'octets lue ou écrite par ces opérations. Connaître la quantité de données véhiculées nous permet de mieux choisir le type de lien matériel qui pourrait être nécessaire. Si l'on doit véhiculer beaucoup de données, un lien DMA peut être avantageux. Inversement, ajouter un lien DMA à de petits transferts peut ralentir l'implémentation à cause du coût d'initialisation du module DMA.

Finalement, il faut pouvoir identifier l'instruction C/C++ qui effectue l'opération. Il est primordial de retrouver dans le code applicatif la ligne associée à l'opération mémoire afin de discerner ce code si l'on doit le transférer.

Afin de démontrer la possibilité d'offrir un outil d'analyse du code dynamique, nous avons identifié l'outil PIN d'Intel [27]. Afin d'utiliser PIN, on doit développer un programme sous forme d'une librairie dynamique appelée « tool » à l'aide duquel on identifie les instructions qui nous intéresse. Chaque instruction peut être inspectée par notre outil et il est alors possible de récupérer les informations que l'on juge pertinentes. Dans ce qui suit, nous démontrons un prototype réalisé avec PIN.

Pour récupérer l'information voulue, nous avons développé la structure **FunctionStruct** afin de contenir l'information des instructions d'une fonction et la structure **InstructionStruct** pour contenir l'information d'une instruction mémoire. La structure **InstructionStruct** contient l'information préalablement discutée soit l'adresse mémoire, l'opération, le lien vers le fichier source. Ces structures sont illustrées par le listage 5.4.

Lors du lancement de l'outil, celui-ci exécute une fonction d'initialisation. Cette fonction se nomme **ImageLoad**. Cette fonction est appelée après le chargement, mais avant l'exécution du binaire à analyser. Elle permet de voyager dans l'image de l'exécutable et de dynamiquement introduire du code. Dans cette fonction, nous devons indiquer quelles instructions nous intéressent en y accrochant une fonction de rappel (callback). Dans notre cas, nous avons voyagé à travers toutes les instructions de type RTN qui indique l'appel d'une fonction. Parmi toutes les fonctions de notre application, nous sommes uniquement intéressés aux opérations mémoire liées à la fonction **WelsMdInterMbRefinement**.

```
enum memory_operation {NIL, READ, WRITE, RW };
string memory_operation_str[] = { "NIL", "READ", "WRITE", "RW" };

// Instructions
typedef struct InstructionStruct
{
        string _symbole;
        string _mnemonic;
        ADDRINT _address;
        memory_operation _mem_op;
        BOOL _is_stack_operation;
        VOID * _writeAddr;
        INT32 _writeSize;
        UINT64 _value;
        INT32 _source_column;
        INT32 _source_line;
        string _source_file;
} Inst_Stats;

// Fonctions
typedef struct FunctionStruct
{
        string _name;
        string _image;
        ADDRINT _address;
        ADDRINT _stack_ptr;
        RTN _rtn;
        std::map<ADDRINT, Inst_Stats*> inst_list;
        struct FunctionStruct* _child_func;
        struct FunctionStruct* _parent_func;
} Func_Stats;
```

Listing 5.4 Structures développées pour maintenir en mémoire les informations des instructions d'une fonction

Une fois que nous avons localisé cette fonction, nous itérons à travers ses instructions et nous récupérons leurs informations. C'est à ce moment que nous pouvons récupérer l'adresse, le

symbole, la mnémonique et la location de l'instruction dans le fichier source. Pour récupérer le reste de l'information dynamique des instructions mémoires, nous y accrochons la fonction de rappel « RecordMem » avec un pointeur vers la structure **InstructionStruct** instancié pour cette instruction. Ce processus est illustré au listing 5.5.

```
for (INS ins = RTN_InsHead(rtn); INS_Valid(ins); ins = INS_Next(ins))
{

        Inst_Stats* _inst = new Inst_Stats();
        _inst->_address = INS_Address(ins);
        _inst->_symbole = INS_Disassemble(ins);
        _inst->_mnemonic = INS_Mnemonic(ins);
        _inst->_mem_op = memory_operation::NIL;
        func_list[_func_stat->_address]->inst_list[inst_index] = _inst;

        PIN_GetSourceLocation(_inst->_address, &(_inst->_source_column)
            , &(_inst->_source_line), &(_inst->_source_file));

        if (INS_IsMemoryRead(ins) && INS_IsStandardMemop(ins))
        {
                INS_InsertPredicatedCall(
                ins, IPOINT_BEFORE, (AFUNPTR)RecordMem,
                IARG_INST_PTR,
                IARG_UINT32, memory_operation::READ,
                IARG_MEMORYREAD_EA,
                IARG_MEMORYREAD_SIZE,
                IARG_BOOL, INS_IsPrefetch(ins),
                IARG_PTR, _func_stat,
                IARG_PTR, _inst,
                IARG_BOOL, INS_IsStackRead(ins),
                IARG_END);
        }

        [...]
}
```

Listing 5.5 Itération à travers les instructions mémoires d'une fonction et identification des opérations mémoires

Une fois l'initialisation faite, l'application à analyser est lancée. Lorsque celle-ci rencontre une instruction mémoire de la fonction **WelsMdInterMbRefinement**, la fonction d'analyse est appelée. À l'intérieur de celle-ci, il est alors possible de récupérer l'adresse mémoire, la quantité de données affectées par l'opération et le type d'opération. Comme illustré par le listage 5.6 la fonction se contente de seulement inscrire pour le moment cette information dans un fichier de journalisation. Le détail complet de l'implémentation de cet outil est joint à l'annexe B.

```
ins_stat->_writeAddr = addr;
ins_stat->_writeSize = size;
ins_stat->_mem_op = mem_op;
ins_stat->_is_stack_operation = is_stack_operation;

TraceFile << ip << ": " << memory_operation_str[ins_stat->
   _mem_op] << " " << ins_stat->_symbole << " " << " Stack: "
   << is_stack_operation
<< setw(2 + 2 * sizeof(ADDRINT)) << ins_stat->_writeAddr << " "
<< dec << setw(2) << ins_stat->_writeSize << " "
<< hex << setw(2 + 2 * sizeof(ADDRINT))
<< " Source line : " << ins_stat->_source_line << " Source
   Column: " << ins_stat->_source_column << " Source file: " <<
    ins_stat->_source_file;
```

Listing 5.6 Corps de la fonction «RecordMem»

Le listing 5.7 illustre le résultat d'une exécution de l'outil développé. De ce listing on remarque l'identification du type d'opération (READ,WRITE), les mnémoniques x86 de l'opération, l'adresse mémoire utilisée, la quantité de données et la ligne du code source associé. On peut ainsi associer ces lectures au code source qui est illustré par le listing 5.8. L'opération de la ligne **00F9C4E9** de la sortie de PIN peut être associée à la lecture de la pile du pointeur vers **SPicData.pRefMb**. La ligne suivante **00F9C4EC** est la lecture en mémoire à l'adresse **03A9824C** à partir de ce pointeur. L'opération d'écriture se fait sur la pile pour enregistrer la valeur du pointeur dans la variable **pRefCb**.

```
00F9C4E9: READ mov ecx, dword ptr [ebp+0x14]  Stack: 0x1
0059D790  4  Source line : 0x5a1 Source Column: 0x8
Source file: d:\documents\maitrise\projetmiranda\projectsource\encoder\
core\src\svc_base_layer_md.cpp

00F9C4EC: READ mov edx, dword ptr [ecx+eax*1+0x188]  Stack: 0
03A9824C  4 Source line : 0x5a1 Source Column: 0xb
Source file: d:\documents\maitrise\projetmiranda\projectsource\encoder\
core\src\svc_base_layer_md.cpp

00F9C4F3: WRITE mov dword ptr [ebp-0xdc], edx  Stack: 0x1
0059D6A0  4 Source line : 0x5a1 Source Column: 0x12
Source file: d:\documents\maitrise\projetmiranda\projectsource\encoder\
core\src\svc_base_layer_md.cpp
```

Listing 5.7 Résultat de l'exécution de l'outil PIN/ProcedureMemTrace sur la fonction WelsMdInterMbRefinement

```
uint8_t* pRefCb = pMbCache->SPicData.pRefMb[1];
uint8_t* pRefCr = pMbCache->SPicData.pRefMb[2];
```

Listing 5.8 Code source des opérations mémoire effectué au listing 5.7

À l'aide de cet outil, nous pouvons obtenir l'information nécessaire pour lier différentes fonctions entre eux par leurs opérations mémoires. Si une instruction écrit en mémoire et une autre instruction plus loin dans une autre fonction/module lie cette zone, nous serons en mesure de rendre ce lien explicite et produire un graphe de flot mémoire comme illustré par la figure 5.1. Cette approche nous permet d'identifier les liens mémoires entre plusieurs fonctions et identifier leurs opérations mémoire dans leur forme absolue. Aucun besoin de les déterminer manuellement avec une abstraction comme celle offerte par la librairie SpaceLib. On peut par la suite modifier ces opérations par le modèle voulu (dma, fifo, etc.) lors de la transformation du code vers le matériel.

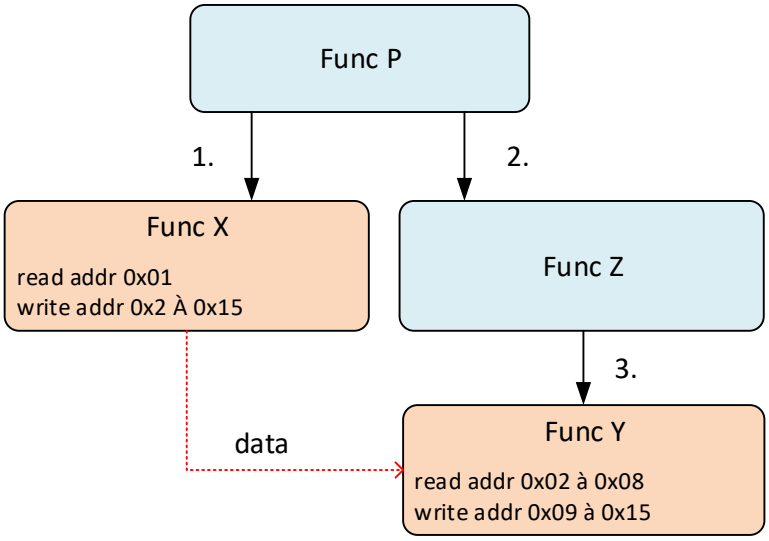



Figure 5.1 Liens mémoires rendus explicites par l'analyse dynamique

### 5.2.4 Transformation

Avec la combinaison des analyses dynamiques et statiques, nous obtenons de l'information ouvrant la porte à la création d'outils permettant d'effectuer automatiquement des modifications au flot de donnée. L'outil d'analyse dynamique serait utilisé pour identifier les opérations mémoire alors que Clang permettrait de changer les opérations en leur équivalence en fonction de l'outil de synthèse ciblé tel que VivadoHLS, CatapultC, Altera, etc. De cette façon, les modifications sont directement effectuées à la compilation du modèle et ne changeraient aucunement la forme du code initial. Aucune transformation du code source. Seule la représentation intermédiaire générée par SpaceStudio contiendrait les modifications.

Une approche à la modification automatique du code peut consister à rapporter toutes les entrées/sortie des fonctions enfant vers la signature de la fonction parent afin de les rendre explicites aux outils HLS. Il serait alors possible d'utiliser les compilateurs HLS, offrant les transformations au flot telles que protocole, segmentation mémoire et « stream », pour passer les donner vers le module matériel. Par contre, ces outils, bien que parfaits pour transformer une fonction en circuit numérique, ils ne peuvent prendre aucune initiative sur les communications à l'échelle du système. C'est-à-dire que la vue de ce type d'outil se limite à la fonction à être transformée. Les outils soutenant la méthodologie doivent prendre en compte la modularisation à plus grande échelle et le flot de données doit être analysé, tout comme le flot

de contrôle au niveau système.

Une autre approche serait d'en faire une analyse bout à bout et le choix du protocole imposé aux outils HLS ferait partie d'un choix de communication global. Il serait possible de permettre aux développeurs de modifier le chemin et le traitement du flot de données à haut niveau pour, par exemple, imposer un protocole particulier (piplinage, streaming, etc.). De cette façon, le flot serait cohérent à la grandeur tant au niveau logiciel que matérielle plutôt que de traiter uniquement aux frontières entre logiciel/matériel. Nous émettons l'hypothèse qu'une telle approche serait plus bénéfique pour les performances d'un système. Il serait aussi possible d'y inclure les communications optimisées en fonction de la plateforme ciblée (Zynq ACP, IBM Power8 , etc.).

Ceci complète l'atteinte du point 5) (Réalisation d'une preuve de concept visant à éliminer certains obstacles qui n'ont pu être traité dans le cadre de ce projet et qui serviront aux travaux futurs) et conclu la démonstration des contributions mentionnées à la section 1.3.

# CHAPITRE 6 CONCLUSION

Les FPGA sont de plus en plus présents dans le paysage informatique afin d'augmenter les performances et réduire la consommation des systèmes. La logique programmable est maintenant présente dans le nuage grâce aux géants Amazone et Google et l'achat d'Altera par Intel promet une prolifération de la logique programmable vers les plateformes grand consommateur. Cette démocratisation des FPGA met entre les mains des développeurs logiciels la possibilité de produire des systèmes hybrides logiciel/matériel. L'approche à l'aide du codesign est alors alléchante pour ceux-ci.

Afin de permettre au développeur logiciel ayant peu d'expérience dans le développement de logique numérique, il est primordial d'améliorer le flot de conception. Dans cette optique, travailler au niveau système à l'aide de spécification exécutable semble un choix naturel pour les développeurs logiciels. Il est donc nécessaire de guider le développeur à travers les embûches reliées aux divers idiomes de développement du domaine matériel. C'est dans cette optique que cette recherche a été effectuée. Elle se voulait une première ébauche.

Dans ce dernier chapitre, il sera question de faire la synthèse des travaux effectués et présenter les étapes restantes afin d'avoir une méthodologie de développement facile d'utilisation et un système Proxy fonctionnel sur FPGA pour la compagnie GrassValley.

## 6.1 Synthèse des travaux

Dans ce mémoire nous avons démontré la possibilité de développer un encodeur H.264 à partir d'une référence logicielle utilisée en entrée et avons proposé les étapes nécessaires pour effectuer les analyses clés d'une exploration architecturale afin de réduire l'aspect dynamique difficile à interpréter par les outils HLS et traduire le code applicatif et le flot de données du domaine logiciel vers le domaine matériel.

Pour expérimenter avec cette approche, nous avons développé en parallèle un encodeur H.264 basé sur une application utilisant la librairie OpenH264. Nous avons détaillé toutes les étapes suivies afin de rendre cette application exécutable sur la plateforme virtuelle du FPGA Zynq de Xilinx de SpaceStudio et produire deux coprocesseurs à l'aide des algorithmes d'estimation du mouvement de la librairie. Nous avons malheureusement produit uniquement une

implémentation sur plateforme virtuelle de ce système, la migration vers le matériel ayant pris un temps considérable. Ce délai a été causé par les limitations de nos outils.

Il a été convenu que notre méthodologie avait certaines limitations et nous avons identifié les obstacles rendant complexe la migration d'algorithmes logiciels vers un modèle matériel durant l'exploration architecturale. Nous avons convenu qu'afin de rendre l'approche intéressante il était nécessaire de faciliter le remodelage du flot de données à l'aide d'outils qui seraient intégrés à SpaceStudio et la méthodologie de développement. Dans cette optique, nous avons proposé deux outils propices à faciliter la conception d'architecture ayant des mémoires unifiés et à identifier plus efficacement le flot de données de la spécification exécutable. Ces outils faisant preuve de concept n'ont pu être traités en profondeur dans le cadre de ce projet et serviront de travaux futurs.

## 6.2 Améliorations futures

Tout au long du développement de ce projet, nous avons identifié plusieurs améliorations qui seraient intéressantes pour le système H.264. Cette section détaillera les principales étapes restantes afin de produire une implémentation d'un système H.264 sur puce FPGA.

### 6.2.1 Système H.264

Lors du développement initial, nous nous sommes seulement attardés aux paramètres d'entrée et de sortie utilisés par les modules matériels. Or, il serait encore possible de réduire ceux-ci. Pour diminuer la quantité de données véhiculées entre les modules, il faut réduire la trace du contrôle en passant seulement les paramètres qui changent d'un macrobloc à un autre. Les modules matériels auraient alors une initialisation pour les paramètres standards reliés à une trame et une fois celle-ci faites, il suffit de passer les données liées à chaque macrobloc et réinitialiser les paramètres standard lors d'une nouvelle trame.

Lors du développement du projet h.264, nous avons ajouté plusieurs fonctionnalités tierces au module principal **MainEncoderTask**. C'est le cas des fonctions de validation (CRC), du calcul des performances et de la journalisation (logging). Toutes ces tâches externes au système et prises en charge par le module principal devront être déplacées dans des modules séparés. Nous voulons libérer la tâche d'encodage de ce fardeau, réduire l'impact sur les performances de l'encodeur et spécialiser les modules du système.

Suite au nettoyage des modules matériel et logiciel, il sera important de se concentrer sur la plus grande limitation de notre implémentation. Comme mentionné préalablement, le flot de données du système est sous optimal. Tout d'abord, nous l'avons conçu de manière à devoir transférer des images complètes vers la zone mémoire partagée à chaque nouvelle trame à encoder. Si nous conservons cette architecture mémoire, il serait nécessaire de déléguer ces transferts à un module spécialisé afin de différer les opérations mémoires pour limiter l'impact des communications sur l'encodage d'une trame. Comme nous avons fait pour les données en entrée avec le module **InputAdapter**. Il serait aussi possible d'utiliser un tampon double afin de charger plusieurs trames à la fois. Si l'on opte pour modifier le flot de données, il serait préférable d'utiliser un tampon de type « line buffer » partagé par les modules IME/FME et éliminer le dédoublement des trames entre la section logiciel et matériel. De cette façon, nous gagnons en vitesse en réduisant le délai de communication mémoire à l'aide du « buffering ». Nous réduisons aussi l'espace mémoire en conservent en mémoire partagé uniquement les pixels nécessaires plutôt que les trames complètes. Ceci permettrait de réduire à environ 960 Ko les deux tampons mémoires présentement utilisés. Avec le processeur ARM Cortex A9, il est possible d'atteindre une meilleure performance des communications mémoires en utilisant la DDR du Zynq et le bus ACP qui utilise la cohérence de cache et le partage de la mémoire centrale avec le processeur. Par contre, comme nous devons utiliser un processeur « softcore » ces accélérations ne sont pas disponibles.

La figure 6.1 illustre l'architecture finale du système incluant un flot mémoire optimisé et la spécialisation des modules logiciels du système.

Une fois le flot de données révisé, l'architecture de la plateforme virtuelle sera à changer. Le système est actuellement exécuté sur un processeur ARM sous Linux et ne représente pas la plateforme cible. Il sera nécessaire de traduire l'application vers le système d'exploitation microC (uC) afin d'être en mesure d'utiliser les processeurs « softcore » tels que uBlaze ou RISK-V comme convenu.

Une fois ces modifications effectuées, les modules matériels devront être passés dans un outil HLS afin de produire des fichiers HDL pour éventuellement synthétiser tout le système sur puce FPGA. Cette étape risque de prendre plusieurs itérations afin de produire un module fonctionnel, car les outils de synthèse peuvent avoir de la difficulté à interpréter correctement le code de ces modules. La capacité du synthétiseur à bien inférer l'architecture matérielle à

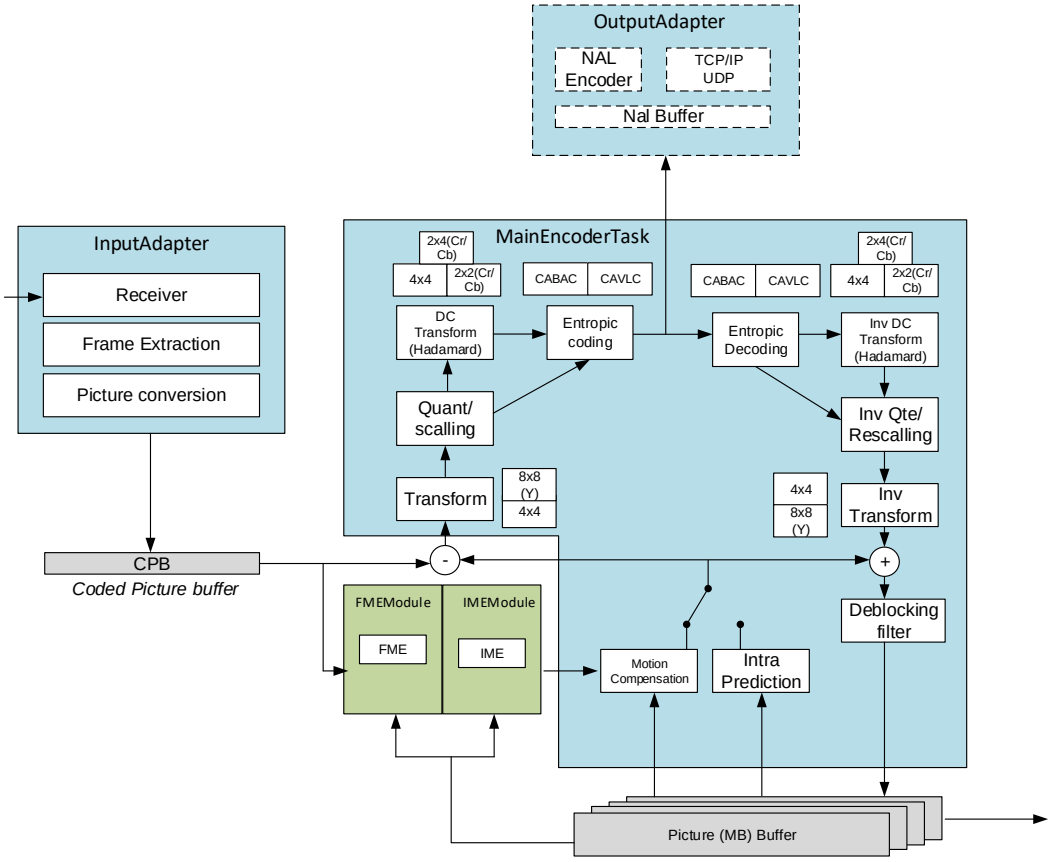



Figure 6.1 Architecture incorporant le nouveau flot de données

partir du logiciel indiquera la cadence possible du système. L'utilisation de bus haute vitesse HP0, les adaptateurs et les fifo de SpaceStudio, les tampons mémoires interne et les énoncées conditionnelles utilisées à profusion dans le code C peuvent tous causer problème aux outils HLS. Le code conditionnel produit des MUX à l'intérieur du circuit et augmente les délais des signaux. La mémoire peut elle même être multiplexée et cela produit des délais encore plus longs. Lorsque nous utilisons le bus HP0 sous SpaceCodesign, les adaptateurs instanciés produisent un lien entre le bus et le signal de réinitialisation trop long pour respecter le délai de 10ns. La méthode de développement SystemC afin de produire un code aisément synthétisable par un outil HLS devra être détaillée dans la méthodologie.

Finalement, afin d'aller chercher plus de performance il sera nécessaire d'augmenter le parallélisme du traitement des macroblocs. Chaque bloc matériel opère sur un macrobloc et n'a besoin que de l'information sur celui-ci. Ils ne partagent pas d'état ou de mémoire avec les tâches logicielles. Ceci rend l'opération des coprocesseurs atomique et le résultat de ceux-ci

est donné à la sortie de leur traitement. Les coprocesseurs ne produisent aucun effet de bord (side effect ou symptôme) sur le reste des données. La difficulté d'introduire le parallélisme au système «Proxy» provient de l'architecture logicielle de la librairie. Plus précisément de la fonction **WelsMdInterMbLoop()**. Après le traitement de chaque macrobloc, des statuts globaux, des pointeurs et d'autres structures sont modifiés. Il faut alors vérifier s'il est possible de rendre indépendante chaque itération de cette boucle de traitement. Si cela est possible, il serait intéressant de produire une grappe de coprocesseur IME/FME afin d'encoder plusieurs macroblocs en parallèle et produire une courbe de Pareto afin de trouver le meilleur compromis entre la performance de l'encodeur, la surface utilisée et la consommation du FPGA.

## 6.3 Méthodologie projeté

Pour terminer, nous élaborons les grandes lignes de la méthodologie telle qu'imaginée avec l'inclusion des outils d'analyse du flot de données et les interfaces augmentées pour SpaceStudio mentionné au chapitre 5.

1. Analyse de l'application au niveau système
   (a) Analyse statique de l'application à l'aide d'un profileur (Valgrind) intégré à SpaceStudio afin de produire un graphe d'appel et identifier les zones chaude.
   (b) Analyse dynamique à l'aide d'un profileur (PIN (Intel)) afin de permettre de lier les diverses fonctions du système en terme de producteur consommateur et avoir une image de la consommation mémoire des zones du code.
      i. Identifier les instructions effectuant les opérations mémoires
      ii. Lister toutes les adresses mémoires utilisées
      iii. Le type d'accès effectué (Lecture, Écriture, Lecture/Écriture)
      iv. Déterminer la quantité de données

      Une fonction/section de code peut alors être qualifiée par sa consommation de donnée et ses liens avec les autres
   (c) Représentation algorithmique de haut niveau du logiciel incluant les détails du flot de données sous forme d'un graphe (arbre de syntaxe abstraite, etc.)
      i. La manipulation des graphes modifie directement la ou les fonctions et leurs liens dans le code applicatif

   ii. Si on doit changer la représentation mémoire (pipeline, multithread, etc) on connaît les liens

2. Développement de la macroarchitecture
   (a) Définition et conception des principaux blocs fonctionnels
   (b) Ajout d'abstraction SpaceStudio automatique pour toutes les opérations mémoire à l'aide de l'outil d'analyse des communications et d'utilisation mémoire développé à l'aide de Clang. Permets une meilleure manipulation du « dataflow » à haut niveau et de facilement migrer une partie d'une application logicielle vers le matériel
      i. Regroupement de fonctions en module
      ii. Test des implémentations, Analyse dynamique, raffinement
      iii. Partionement HW/SW, Test des implémentations, Analyse dynamique, raffinement
      iv. Exploration architecturale. Inclure diverses architectures softcores telles que Microblaze, RISK-V, MIPS, Spark, etc.
3. Développement de la microarchitecture
   (a) Optimisation des blocs fonctionnels
   (b) Analyse des communications entre les modules et développement du « datapath » adapté aux modules
   (c) Réutilisation des modules de tests de la première phase pour valider les changements effectués sous cette phase
   (d) Simulation
   (e) HLS. Lors de l'itération HLS il sera nécessaire de modifier la méthodologie afin d'offrir des patrons de conception ainsi que des outils pour simplifier la conversion HLS.
   (f) Synthèse et implémentation
   (g) Raffinement de la microarchitecture afin de rencontrer les requis algorithmiques initiaux du client

# RÉFÉRENCES


[1] Barr group crc series, part 3 : Crc implementation code in c/c++. `https://barrgroup.com/Embedded-Systems/How-To/CRC-Calculation-C-Code`. Consulter le : 2016-06-01.

[2] Intel quartus prime design software. `https://www.altera.com/products/design-software/fpga-design/quartus-prime/overview.html`. Consulter le : 2016-11-22.

[3] Spectra-q engine. `https://www.altera.com/products/design-software/fpga-design/quartus-prime/features/spectra-q.html`. Consulter le : 2016-11-22.

[4] Open virtual platforms. `http://www.ovpworld.org/ovptechnology`, consulté le 15 novembre 2016.

[5] Aarno, Daniel, and Jakob Engblom. *Software and System Development using Virtual Platforms : Full-System Simulation with Wind River Simics*. Morgan Kaufmann, 2014.

[6] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal, and M. Luján. An empirical evaluation of high-level synthesis languages and tools for database acceleration. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2014.

[7] A. B. Atitallah, H. Loukil, and N. Masmoudi. Fpga design for h. 264/avc encoder. *International Journal of Computer Science, Engineering and Applications*, page 119, 2011.

[8] B. Bailey and G. Martin. Codesign experiences based on a virtual platform. In *ESL Models and their Application*, pages 273–308. Springer, 2010.

[9] Bellard, Fabrice. QEMU, a fast and portable dynamic translator. *USENIX Annual Technical Conference*, 2005.

[10] Binkert, Nathan, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 2011.

[11] Black, David C., et al. *SystemC : From the ground up*. Springer Science & Business Media, 2009.

[12] T. Bollaert. Catapult synthesis : a practical introduction to interactive c synthesis. In *High-Level Synthesis*, pages 29–52. Springer, 2008.

[13] Cai, Lukai, and Daniel Gajski. Transaction level modeling : an overview. *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2003.

[14] T.-C. Chen, C. Lian Jr, and L.-G. Chen. Hardware architecture design of an h. 264/avc video codec. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 750–757, 2006.

[15] J. Chevalier, M. de Nanclas, L. Filion, O. Benny, M. Rondonneau, G. Bois, and E. M. Aboulhamid. A systemc refinement methodology for embedded software. *IEEE Design & Test of Computers*, 23(2) :148–158, 2006.

[16] I. CoFluent. Intel cofluent studio. `http://www.intel.com/content/www/us/en/cofluent/cofluent-difference.html`, 2016.

[17] T. Damak, L. Ayadi, N. Masmoudi, and S. Bilavarn. Hls and manual design methodology for h. 264/avc deblocking filter. In *Information Technology and Computer Applications Congress (WCITCA), 2015 World Congress on*, pages 1–5. IEEE, 2015.

[18] Edgar E. Iglesias. Tlmu : Transaction level emulator. `http://edgarigl.github.io/tlmu/tlmu-doc.html`, consulté le 15 novembre 2016.

[19] Engblom, Jakob. Virtual to the (near) end-Using virtual platforms for continuous integration. *52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015.

[20] R. Ernst. Codesign of embedded systems : status and trends. In *IEEE Design and Test of Computers*, pages 45–54, Aug. 2002.

[21] T. Feist. Vivado design suite. *White Paper*, 5, 2012.

[22] Frédéric Fortier, Nicolas Gagnon, Vincent Longpré, Philippe Sasseville, Marvens Toussaint. Rapport final projet système embarqué encodeur/décodeur h.264.

[23] G. De Michell, R.K. Gupta. Hardware/software co-design. In *The Proceedings (IEEE)*, pages 349–365, 2002.

[24] J. Keinert, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, M. Meredith, et al. Systemcodesigner—an automatic esl synthesis approach by design space exploration and behavioral synthesis for streaming applications. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1) :1, 2009.

[25] X. W. KiemHung Nguyen, Peng Cao. Mapping H.264/AVC Fractional Motion Estimation Algorithm onto Coarse-Grained Reconfigurable Computing System. *Advances in Future Computer and Control Systems*, 2012.

[26] X. Liu, Y. Chen, T. Nguyen, S. Gurumani, K. Rupnow, and D. Chen. High level synthesis of complex applications : An h.264 video decoder. In *Proceedings of the 2016 ACM/-*

*SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 224–233, 2016.

[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin : building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[28] Magnusson, Peter S., et al. Simics : A full system simulation platform. *Computer*, 2002.

[29] Martin, Grant, Brian Bailey, Andrew Piziali. *ESL design and verification : a prescription for electronic system level methodology*. Morgan Kaufmann, 2010.

[30] M. Meredith. High-level systemc synthesis with forte's cynthesizer. In *High-Level Synthesis*, pages 75–97. Springer, 2008.

[31] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), Apr. 1965.

[32] L. Moss. *Profilage, caractérisation et partitionnement fonctionnel dans une plate-forme de conception de systèmes embarqués*. PhD thesis, École Polytechnique de Montréal, 2010.

[33] L. Moss, H. Guerard, G. Dare, and G. Bois. An esl methodology for rapid creation of embedded aerospace systems using hardware-software co-design on virtual platforms. Technical report, SAE Technical Paper, 2012.

[34] L. Moss, H. Guérard, G. Dare, and G. Bois. Recent experience on an esl framework for rapid design exploration using hardware-software codesign for arm based fpgas. In *SAME 2012 Conference, October*, volume 2, 2012.

[35] R. Nane, V.-M. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels. Dwarv 2.0 : A cosy-based c-to-vhdl hardware compiler. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 619–622. IEEE, 2012.

[36] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, et al. A survey and evaluation of fpga high-level synthesis tools. 2015.

[37] Open SystemC Initiative. Osci tlm 2.0 language reference manual. `http://accellera.org/images/downloads/standards/systemc/TLM_2_0_LRM.pdf`, 2009.

[38] Qualcomm. Snapdragon 820 processor product brief. `https://www.qualcomm.com/documents/snapdragon-820-processor-product-brief`, 2016.

[39] Richardson, Iain E. *The H. 264 advanced video compression standard*. John Wiley & Sons, 2011.

[40] J. Rowson. Hardware/Software Co-Simulation. *31st Conference on Design Automation*, 1994.

[41] Schirrmeister, Frank, and Filip Thoen. Hardware Virtualization for Pre-Silicon Software Development in Automotive Electronics. In *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, 2009.

[42] M. P. Singh and M. K. Jain. Evolution of processor architecture in mobile phones. *International Journal of Computer Applications*, 2014.

[43] J. Teich. Hardware/software codesign : The past, the present, and predicting the future. *Proceedings of the IEEE*, 100(Special Centennial Issue) :1411–1430, 2012.

[44] J. Teich. Hardware/Software Codesign : The Past, the Present, and Predicting the Future. In *The Proceedings (IEEE)*, pages 1411–1430, 2012.

[45] Tobias Blickle, Jürgen Teich, Lothar Thiele. System-Level Synthesis Using Evolutionary Algorithms. *Automation for Embedded Systems*, 1998.

[46] L.-G. C. Tung-Chien Chen, Chung-Jr Lian. Hardware architecture design of an H.264/AVC video codec. In *Conference on Design Automation, 2006. Asia and South Pacific*, 2006.

[47] V. Živojnovic, H. Meyr. Compiled HW/SW co-simulation. In *the 33rd annual Proceedings of Design Automation*, pages 690–695, 1996.

[48] R. Wang, M. Li, J. Li, and Y. Zhang. High throughput and low memory access sub-pixel interpolation architecture for h. 264/avc hdtv decoder. *IEEE Transactions on Consumer Electronics*, pages 1006–1013, 2005.

[49] G. M. Wayne Wolf, Ahmed Amine Jerraya. Multiprocessor System-on-Chip (MPSoC) Technology . In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1701–1713, 2008.

[50] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the h. 264/avc video coding standard. *IEEE Transactions on circuits and systems for video technology*, 2003.

[51] W. Wolf. A decade of hardware/software codesign. In *Computer*, pages 38–43, Apr. 2003.

[52] Yang Kun, Zhang Chun, Du Guoze, Xie Jiangxiang, Wang Zhihua. A Hardware-Software Co-design for H.264/AVG Decoder. *IEEE Asian Solid-State Circuits Conference*, 2006.

[53] S. Zhu and K.-K. Ma. A new diamond search algorithm for fast block matching motion estimation. In *Information, Communications and Signal Processing, 1997. ICICS., Proceedings of 1997 International Conference on*, volume 1, pages 292–296. IEEE, 1997.

# ANNEXE A PARAMÈTRES DE L'ENCODEUR

```
#define ENCODER_LOW_PROFILE

#ifdef ENCODER_LOW_PROFILE

#define SOC_ENCODER_INPUT_IMAGE_HEIGHT  180                                    /// Input height
#define SOC_ENCODER_INPUT_IMAGE_WIDTH  320                                     /// Input width
#define SOC_ENCODER_OUTPUT_IMAGE_WIDTH 320                                  /// Ouput Image Width
#define SOC_ENCODER_OUTPUT_IMAGE_HEIGHT 180                                 /// Ouput Image height

#endif

#ifdef ENCODER_HIGH_PROFILE

#define SOC_ENCODER_INPUT_IMAGE_HEIGHT  720                                    /// Input height
#define SOC_ENCODER_INPUT_IMAGE_WIDTH 1280                                     /// Input width
#define SOC_ENCODER_OUTPUT_IMAGE_WIDTH 1280                                 /// Ouput Image Width
#define SOC_ENCODER_OUTPUT_IMAGE_HEIGHT 720                                 /// Ouput Image height

#endif

#ifdef ENCODER_MIX_PROFILE

#define SOC_ENCODER_INPUT_IMAGE_HEIGHT  720                                    /// Input height
#define SOC_ENCODER_INPUT_IMAGE_WIDTH 1280                                     /// Input width
#define SOC_ENCODER_OUTPUT_IMAGE_WIDTH 640                                  /// Ouput Image Width
#define SOC_ENCODER_OUTPUT_IMAGE_HEIGHT 360                                 /// Ouput Image height

#endif
#define SOC_ENCODER_INPUT_FRAME_SIZE ((SOC_ENCODER_INPUT_IMAGE_HEIGHT * SOC_ENCODER_INPUT_IMAGE_WIDTH * 3)
      >> 1)

//#define SOC_ENCODER_USAGE_TYPE CAMERA_VIDEO_REAL_TIME          /// 0: camera video 1:screen content
#define SOC_ENCODER_USAGE_TYPE CAMERA_VIDEO_NON_REAL_TIME                /// 0: camera video 1:screen
     content
//#define SOC_ENCODER_INPUT_MAX_FRAMERATE 10                                   /// Input framerate
#define SOC_ENCODER_INPUT_MAX_FRAMERATE 25                                     /// Input framerate
#define SOC_ENCODER_TEMPORAL_LAYER 1                                        /// ---Temporal layer
#define SOC_ENCODER_INTRA_PERIOD 0                                             /// ---Intra Period
      ( multipler of GoP size or -1)
#define SOC_ENCODER_MAX_NAL_SIZE 0                                             /// Unit:Byte,
     Maximum Nal size, 0 = default value
#define SOC_ENCODER_SPS_PPS_ID_ADDITION_STRAT INCREASING_ID ///
#define SOC_ENCODER_ENABLE_SSEI false                                       ///
#define SOC_ENCODER_ENABLE_FRAME_CROPPING 1                                 ///
#define SOC_ENCODER_ENTROPY_CODING_TYPE  0                                  /// 0: CAVLC, 1: CABAC
#define SOC_ENCODER_LOOP_FILTER_DISABLE_IDC 0                            /// Loop filter idc (0: on, 1: off,
/// 2: on except for slice boundaries,
/// 3: two stage.slice boundries on in second stage
/// 4: Luma on but Chroma off(w.r.t.idc = 0)
/// 5: Luma on except on slice boundaries, but Chroma off in enh.layer(w.r.t.idc = 2)
/// 6: Luma on in two stage.slice boundries on in second stage, but Chroma off(w.r.t.idc = 3)

#define SOC_ENCODER_LOOP_FILTER_ALPHA_C0_OFFSET 0                        /// AlphaOffset(-6..+6): valid
     range
#define SOC_ENCODER_LOOP_FILTER_BETA_OFFSET 0                            /// BetaOffset (-6..+6): valid
     range
#define SOC_ENCODER_MULTIPLE_THREAD_IDC 1                                      /// 0: auto(dynamic imp.
     internal encoder); 1: multiple threads imp. disabled; > 1: count number of threads;
#define SOC_ENCODER_RC_MODE RC_QUALITY_MODE                                    /// 0: quality mode;  1:
     bitrate mode;  2: buffer based mode,can't control bitrate; -1: rc off mode;
#define SOC_ENCODER_TARGET_BITRATE_KPBS 300                              /// Unit Kbps
#define SOC_ENCODER_MAX_BITRATE_KBPS 1000                                      /// max bitrate overall,
     Kbps
```

Listing A.1 Fichier codec_app_def.h, partie 1

```
//#define SOC_ENCODER_TARGET_BITRATE_KPBS 8000                    /// Unit Kbps
//#define SOC_ENCODER_MAX_BITRATE_KBPS 15000                      /// max bitrate overall,
    Kbps
#define SOC_ENCODER_DENOISE_FILTER 0                              /// Enable Denoise (1:
    enable, 0: disable)
#define SOC_ENCODER_SCENE_CHANGE_DETECT 1                         /// Enable Scene Change
    Detection (1: enable, 0: disable)
#define SOC_ENCODER_BACKGROUND_DETECTION 1                        /// BGD control(1: enable,
    0: disable)
#define SOC_ENCODER_ADAPTIVE_QUANT 1                              /// Enable Adaptive
    Quantization (1: enable, 0: disable)
#define SOC_ENCODER_FRAME_SKIP 0                                  /// Enable frame
    skip (skipping input frames)
#define SOC_ENCODER_LONG_TERM_REF 1                               /// Enable Long
    Term Reference (1: enable, 0: disable)
#define SOC_ENCODER_LONG_TERM_REF_NB AUTO_REF_PIC_COUNT           /// number of reference frame used
#define SOC_ENCODER_LONG_TERM_MARK_PERIOD 30                      /// Long Term Reference Marking
    Period
#define SOC_ENCODER_PREFIX_NAL_ADDING_CTRL 0                      /// Control flag of adding prefix
    unit (0: off, 1: on)
/// It shall always be on in SVC contexts(i.e.when there are CGS / MGS / spatial enhancement layers)
/// Can be disabled when no inter spatial layer prediction in case of its value as 0

#define SOC_ENCODER_IS_LOSSLESS_LINK 0
#define SOC_ENCODER_SPATIAL_LAYER_NB 1                            /// Number of layer

/// Layer 1 (Only one ouput for more output, we need to add layers)
#define SOC_ENCODER_OUTPUT_IMAGE_WIDTH_720P 1280                  /// Ouput Image
    Width
#define SOC_ENCODER_OUTPUT_IMAGE_HEIGHT_720P 720                  /// Ouput Image
    height
#define SOC_ENCODER_OUTPUT_IMAGE_WIDTH_320P 320                   /// Ouput Image Width
#define SOC_ENCODER_OUTPUT_IMAGE_HEIGHT_320P 180                  /// Ouput Image
    height

#define SOC_ENCODER_OUTPUT_FRAMERATE 30                           /// Ouput framerate
//#define SOC_ENCODER_OUTPUT_FRAMERATE 10                         /// Ouput framerate
#define SOC_ENCODER_ENCODING_PROFILE PRO_BASELINE                 /// Encoding Profile
#define SOC_ENCODER_ENCODING_LEVEL LEVEL_UNKNOWN                  /// value of profile IDC (0 or
    LEVEL_UNKNOWN for auto-detection)
#define SOC_ENCODER_OUTPUT_SPATIAL_BITRATE_KBPS 300               /// Output Bitrate
#define SOC_ENCODER_OUTPUT_MAX_SPATIAL_BITRATE_KBPS 1000          /// Output max bitrate
//#define SOC_ENCODER_OUTPUT_SPATIAL_BITRATE_KBPS 6000            /// Output Bitrate
//#define SOC_ENCODER_OUTPUT_MAX_SPATIAL_BITRATE_KBPS 8000        /// Output max bitrate
#define SOC_ENCODER_QUANTIZATION_QUALITY 24                       /// Quantization parameters
     for base quality layer
#define SOC_ENCODER_SLICE_MODE SM_SINGLE_SLICE                    /// 0: single slice mode; >0:
    multiple slices mode, see below;
#define SOC_ENCODER_SLICE_SIZE 1500                               /// only used when
    uiSliceMode=4
#define SOC_ENCODER_SLICE_NUM 1                                   /// multiple slices
     number specified (1 to 8)
#define SOC_ENCODER_SLICE0_NB_MB 960                              /// Slice 0 number of
    Macroblock
//#define SOC_ENCODER_SLICE1_NB_MB 0
//#define SOC_ENCODER_SLICE2_NB_MB 0
//#define SOC_ENCODER_SLICE3_NB_MB 0
//#define SOC_ENCODER_SLICE4_NB_MB 0
//#define SOC_ENCODER_SLICE5_NB_MB 0
//#define SOC_ENCODER_SLICE6_NB_MB 0
//#define SOC_ENCODER_SLICE7_NB_MB 0
```

Listing A.2 Fichier codec_app_def.h, partie 2

```
#if (SOC_ENCODER_OUTPUT_SPATIAL_BITRATE_KBPS > SOC_ENCODER_TARGET_BITRATE_KPBS)
#error "Invalid output spatial bitrate. The output can't be greater than the input spatial bitrate\n"
#endif


#if (SOC_ENCODER_OUTPUT_MAX_SPATIAL_BITRATE_KBPS > 0 && SOC_ENCODER_OUTPUT_MAX_SPATIAL_BITRATE_KBPS <
    SOC_ENCODER_OUTPUT_SPATIAL_BITRATE_KBPS)
#error "Invalid max spatial(#%d) bitrate(%d) setting::: < layerBitrate(%d)!\n"
#endif


// !!! Not to be redefined for different Hardware spec
#define SOC_ENCODER_TARGET_BITRATE (SOC_ENCODER_TARGET_BITRATE_KPBS * 1000)                    // bits per
    second
#define SOC_ENCODER_MAX_BITRATE (SOC_ENCODER_MAX_BITRATE_KBPS * 1000)                          // bits per
    second
#define SOC_ENCODER_OUTPUT_SPATIAL_BITRATE (SOC_ENCODER_OUTPUT_SPATIAL_BITRATE_KBPS * 1000)
#define SOC_ENCODER_OUTPUT_MAX_SPATIAL_BITRATE (SOC_ENCODER_OUTPUT_MAX_SPATIAL_BITRATE_KBPS * 1000)

/* Constants */
#define MAX_TEMPORAL_LAYER_NUM          4
#define MAX_SPATIAL_LAYER_NUM           4 //SOC_ENCODER_SPATIAL_LAYER_NB 4
#define MAX_QUALITY_LAYER_NUM           4

#define MAX_LAYER_NUM_OF_FRAME          128
#define MAX_NAL_UNITS_IN_LAYER          128     ///< predetermined here, adjust it later if need

#define MAX_RTP_PAYLOAD_LEN             1000
#define AVERAGE_RTP_PAYLOAD_LEN         800


#define SAVED_NALUNIT_NUM_TMP           ( (MAX_SPATIAL_LAYER_NUM*MAX_QUALITY_LAYER_NUM) + 1 +
    MAX_SPATIAL_LAYER_NUM )  ///< SPS/PPS + SEI/SSEI + PADDING_NAL
#define MAX_SLICES_NUM_TMP                      SOC_ENCODER_SLICE_NUM /*( ( MAX_NAL_UNITS_IN_LAYER -
    SAVED_NALUNIT_NUM_TMP ) / 3 )*/


#define AUTO_REF_PIC_COUNT  -1          ///< encoder selects the number of reference frame automatically
#define UNSPECIFIED_BIT_RATE 0          ///< to do: add detail comment*/
```

Listing A.3 Fichier codec_app_def.h, partie 3

```
#ifdef ELIX
#define VIDEO_FILE_PATH "../../../../../../../../test_videos/"
#else
#define VIDEO_FILE_PATH "/home/root/tb/"
#endif

#ifdef DYNAMIC_COLORFULL_TEST
//InputFile
#define INPUT_FILE_NAME  VIDEO_FILE_PATH "inception_test_320p.yuv"
//OutputFile
#define OUTPUT_FILE VIDEO_FILE_PATH "yuv_video_test_dynamic_colorfull_result.264"

#define OUTPUT_STATS_FILE_OFFSET 0

#define GOLDEN_MODEL_DATA_MEMORY_OFFSET 0x146ED7

#endif

#ifdef STATIC_UNIFORM_TEST
//InputFile
#define INPUT_FILE_NAME  VIDEO_FILE_PATH "heat_test_320p.yuv"
//OutputFile
#define OUTPUT_FILE VIDEO_FILE_PATH "yuv_video_test_static_uniform_result.264"

#define OUTPUT_STATS_FILE_OFFSET 0

#define GOLDEN_MODEL_DATA_MEMORY_OFFSET 0

#endif

#ifdef DYNAMIC_UNIFORM_TEST
//InputFile
#define INPUT_FILE_NAME  VIDEO_FILE_PATH "video_test_320p.yuv"
//OutputFile
#define OUTPUT_FILE VIDEO_FILE_PATH "yuv_video_test_dynamic_uniform_result.264"

#define OUTPUT_STATS_FILE_OFFSET 0

#define GOLDEN_MODEL_DATA_MEMORY_OFFSET 0xE4A21

#endif
```

Listing A.4 Fichier codec_test_file_def.h

# ANNEXE B PROTOTYPE DÉVELOPPÉ DE L'OUTIL PIN

```
/* ================================================================== */
/* Global Variables */
/* ================================================================== */

std::ofstream TraceFile;

enum memory_operation {NIL, READ, WRITE, RW };
string memory_operation_str[] = { "NIL", "READ", "WRITE", "RW" };

typedef struct InstructionStruct
{
        string _symbole;
        string _mnemonic;
        ADDRINT _address;
        memory_operation _mem_op;
        BOOL _is_stack_operation;
        VOID * _writeAddr;
        INT32 _writeSize;
        UINT64 _value;
        INT32 _source_column;
        INT32 _source_line;
        string _source_file;
} Inst_Stats;

// procedure
typedef struct FunctionStruct
{
        string _name;
        string _image;
        ADDRINT _address;
        ADDRINT _stack_ptr;
        RTN _rtn;
        std::map<ADDRINT, Inst_Stats*> inst_list;
        struct FunctionStruct* _child_func;
        struct FunctionStruct* _parent_func;
} Func_Stats;


std::map<ADDRINT, Func_Stats*> func_list;
static Func_Stats *_calling_func = NULL;

/* ================================================================== */
/* Commandline Switches */
/* ================================================================== */

KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
"o", "pinatrace.out", "specify trace file name");
KNOB<BOOL> KnobValues(KNOB_MODE_WRITEONCE, "pintool",
"values", "1", "Output memory values reads and written");
```

Listing B.1 Déclarations globales, partie 1

```
static VOID EnterFunction(Func_Stats *func_stat, ADDRINT sp)
{
        if (_calling_func != NULL)
        {
                _calling_func->_child_func = func_stat;
        }

        func_stat->_parent_func = _calling_func;
        _calling_func = func_stat;

        func_stat->_stack_ptr = sp;
}

static VOID ExitFunction(Func_Stats *func_stat)
{
        _calling_func = func_stat->_parent_func;
}
```

Listing B.2 Fonctions de rappel appelées à l'entrée et la sortie d'une fonction. Utilisé pour conserver l'ordre de préséance

```
static VOID EmitMem(VOID * ea, INT32 size, Inst_Stats *ins_stat)
{
        if (!KnobValues)
                return;

        switch(size)
        {

                case 1:
                        ins_stat->_value = static_cast<UINT64>(*static_cast<UINT8*>(ea));
                        break;

                case 2:
                        ins_stat->_value = static_cast<UINT64>(*static_cast<UINT16*>(ea));
                        break;

                case 4:
                        ins_stat->_value = static_cast<UINT64>(*static_cast<UINT32*>(ea));
                        break;

                case 8:
                        ins_stat->_value = *static_cast<UINT64*>(ea);
                        break;

                default:
                        break;
        }
}

static VOID RecordMem(VOID * ip, memory_operation mem_op, VOID * addr, INT32 size, BOOL isPrefetch,
    Func_Stats *func_stat, Inst_Stats *ins_stat, BOOL is_stack_operation)
{

        ins_stat->_writeAddr = addr;
        ins_stat->_writeSize = size;
        ins_stat->_mem_op = mem_op;
        ins_stat->_is_stack_operation = is_stack_operation;

        TraceFile << ip << ": " << memory_operation_str[ins_stat->_mem_op] << " " << ins_stat->_symbole <<
            " " << " Stack: " << is_stack_operation
        << setw(2 + 2 * sizeof(ADDRINT)) << ins_stat->_writeAddr << " "
        << dec << setw(2) << ins_stat->_writeSize << " "
        << hex << setw(2 + 2 * sizeof(ADDRINT))
        << " Source line : " << ins_stat->_source_line << " Source Column: " << ins_stat->_source_column <<
             " Source file: " << ins_stat->_source_file;

        // retreive value at mem addr
        if (!isPrefetch)
        EmitMem(addr, size, ins_stat);

        TraceFile << endl;

}

static VOID * WriteAddr;
static INT32 WriteSize;

static VOID RecordWriteAddrSize(VOID * addr, INT32 size)
{
        WriteAddr = addr;
        WriteSize = size;
}


static VOID RecordMemWrite(VOID * ip, Func_Stats *func_stat, Inst_Stats *ins_stat, BOOL is_stack_operation)
{
        RecordMem(ip, memory_operation::WRITE, WriteAddr, WriteSize, false, func_stat, ins_stat,
            is_stack_operation);
}
```

Listing B.3 Fonctions d'identification dynamique des opérations mémoires effectuées

```
static VOID ImageLoad(IMG img, VOID *v)
{
        // For simplicity, instrument only the main image. This can be extended to any other image of
            course.
        if (IMG_IsMainExecutable(img))
        {

                // To find all the instructions in the image, we traverse the sections of the image.
                for (SEC sec = IMG_SecHead(img); SEC_Valid(sec); sec = SEC_Next(sec))
                {

                        // For each section, process all RTNs.
                        for (RTN rtn = SEC_RtnHead(sec); RTN_Valid(rtn); rtn = RTN_Next(rtn))
                        {

                                Func_Stats* _func_stat = new Func_Stats();
                                _func_stat->_rtn = rtn;
                                _func_stat->_name = RTN_Name(rtn);
                                _func_stat->_child_func = NULL;
                                _func_stat->_parent_func = NULL;
                                _func_stat->_address = RTN_Address(rtn);

                                func_list[_func_stat->_address] = _func_stat;

                                _func_stat->_name = PIN_UndecorateSymbolName(_func_stat->_name,
                                    UNDECORATION_NAME_ONLY);

                                RTN_Open(rtn);

                                unsigned int inst_index = 0;

                                //Instrument enter and exit function to map child and parent function (link
                                    /call graph)
                                RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)EnterFunction,
                                IARG_PTR, _func_stat,
                                IARG_REG_VALUE, REG_STACK_PTR,
                                IARG_END);

                                RTN_InsertCall(rtn, IPOINT_AFTER, (AFUNPTR)ExitFunction,
                                IARG_PTR, _func_stat,
                                IARG_END);

                                if (_func_stat->_name == "WelsEnc::WelsMdInterMbRefinement")
                                {

                                        for (INS ins = RTN_InsHead(rtn); INS_Valid(ins); ins = INS_Next(ins
                                            ))
                                        {

                                                Inst_Stats* _inst = new Inst_Stats();
                                                _inst->_address = INS_Address(ins);
                                                _inst->_symbole = INS_Disassemble(ins);
                                                _inst->_mnemonic = INS_Mnemonic(ins);
                                                _inst->_mem_op = memory_operation::NIL;
                                                func_list[_func_stat->_address]->inst_list[inst_index] =
                                                    _inst;

                                                PIN_GetSourceLocation(_inst->_address, &(_inst->
                                                    _source_column), &(_inst->_source_line), &(_inst->
                                                    _source_file));
```

Listing B.4 Fonction ImageLoad de notre outil d'analyse dynamique, partie 1

```
                                        if (INS_IsMemoryRead(ins) && INS_IsStandardMemop(ins))
                                        {
                                                INS_InsertPredicatedCall(
                                                ins, IPOINT_BEFORE, (AFUNPTR)RecordMem,
                                                IARG_INST_PTR,
                                                IARG_UINT32, memory_operation::READ,
                                                IARG_MEMORYREAD_EA,
                                                IARG_MEMORYREAD_SIZE,
                                                IARG_BOOL, INS_IsPrefetch(ins),
                                                IARG_PTR, _func_stat,
                                                IARG_PTR, _inst,
                                                IARG_BOOL, INS_IsStackRead(ins),
                                                IARG_END);
                                        }

                                        if (INS_HasMemoryRead2(ins) && INS_IsStandardMemop(ins))
                                        {
                                                INS_InsertPredicatedCall(
                                                ins, IPOINT_BEFORE, (AFUNPTR)RecordMem,
                                                IARG_INST_PTR,
                                                IARG_UINT32, memory_operation::READ,
                                                IARG_MEMORYREAD2_EA,
                                                IARG_MEMORYREAD_SIZE,
                                                IARG_BOOL, INS_IsPrefetch(ins),
                                                IARG_PTR, _func_stat,
                                                IARG_PTR, _inst,
                                                IARG_BOOL, INS_IsStackRead(ins),
                                                IARG_END);
                                        }

                                        if (INS_IsMemoryWrite(ins) && INS_IsStandardMemop(ins))
                                        {
                                                INS_InsertPredicatedCall(
                                                ins, IPOINT_BEFORE, (AFUNPTR)RecordWriteAddrSize,
                                                IARG_MEMORYWRITE_EA,
                                                IARG_MEMORYWRITE_SIZE,
                                                IARG_END);

                                                if (INS_HasFallThrough(ins))
                                                {
                                                        INS_InsertCall(
                                                        ins, IPOINT_AFTER, (AFUNPTR)RecordMemWrite,
                                                        IARG_INST_PTR,
                                                        IARG_PTR, _func_stat,
                                                        IARG_PTR, _inst,
                                                        IARG_BOOL, INS_IsStackWrite(ins),
                                                        IARG_END);
                                                }
                                                if (INS_IsBranchOrCall(ins))
                                                {
                                                        INS_InsertCall(
                                                        ins, IPOINT_TAKEN_BRANCH, (AFUNPTR)
                                                             RecordMemWrite,
                                                        IARG_INST_PTR,
                                                        IARG_PTR, _func_stat,
                                                        IARG_PTR, _inst,
                                                        IARG_BOOL, INS_IsStackWrite(ins),
                                                        IARG_END);
                                                }

                                        }
                                }
                        }

                        inst_index++;
                        RTN_Close(rtn);
                }
        }
    }
}
```

Listing B.5 Fonction ImageLoad de notre outil d'analyse dynamique, partie 2

```
/* ===================================================================== */
/* Main                                                                  */
/* ===================================================================== */

int main(int argc, char *argv[])
{
        string trace_header = string("#\n"
        "# Memory Access Trace Generated By Pin\n"
        "#\n");


        PIN_InitSymbols();

        if( PIN_Init(argc,argv) )
        {
                return Usage();
        }

        TraceFile.open(KnobOutputFile.Value().c_str());
        TraceFile.write(trace_header.c_str(),trace_header.size());
        TraceFile.setf(ios::showbase);

        IMG_AddInstrumentFunction(ImageLoad, 0);
        PIN_AddFiniFunction(Fini, 0);

        // Never returns
        PIN_StartProgram();

        RecordMemWrite(0, 0, 0, false);
        RecordWriteAddrSize(0, 0);


        return 0;
}
```

Listing B.6 Fonction main

# ANNEXE C PROBLÈMES RENCONTRÉS LORS DE LA MIGRATION INITIALE VERS SPACESTUDIO

Lors de la migration de la spécification exécutable nous avons fait face à quelques problèmes reliés aux outils de compilation et d'analyse de SpaceStudio.

Un premier problème est lié au modèle ELIX qui utilise le compilateur MinGW. Ce compilateur permet la compilation sous la norme POSIX sous Windows. Or, les compatibilités entre ces deux plateformes sont problématiques sous SpaceStudio. Les types 64 bits définis sous la norme C99 sont un de ces problèmes. Si l'on veut afficher un type 64 bits sous POSIX, le format d'affichage est %lld. Par contre, ce format n'existe pas sous les librairies WIN32, car les libraires C de Windows ne supportent pas C99. Afin de limiter les problèmes l'application à migrer sous SpaceStudsio doit être développé avec le standard C89 ou C++ 2003. De plus, certains déréférencements de pointeur causaient des avertissements et erreurs lors de la compilation. Ceci était lié au raccourci utilisé par les développeurs d'OpenH264 afin de copier 4 octets à la fois dans un tableau d'octet à trois dimensions. Avec l'option [-Wstrict-aliasing], le compilateur gcc ne permet pas d'accéder à une zone mémoire à partir de deux types de pointeur différents. Dans ce cas, le tabelau est un pointeur de type uint8_t alors que l'opération déréférence le tableau en pointeur de type uint32_t. Bien qu'il s'agit seulement d'avertissement, ces déréférencements ont été corrigé par l'utilisation de pointeur du même type.

Un second problème est lié aux outils de communication. Lors du développement d'une architecture, SpaceStudio exécute deux passes de compilation. La première nommée « Generate » vérifie les communications entre les divers modules afin d'assurer l'intégrité. C'est-à-dire qu'une écriture vers un module est accompagnée d'une lecture de ce module. Lors de cette passe, le compilateur de SpaceStudio utilise des fichiers « stub » des librairies Windows et Linux afin de limiter le temps d'analyse étant donnée que seulement le code associé à la SpaceLib est pertinent. Ceci a engendré une quantité de problèmes lors de la migration, car plusieurs des fichiers d'en-tête référencé par l'encodeur n'étaient pas définis dans ces « stub ». À l'aide des conseilles des ingénieurs de SpaceStudio, nous avons placé les fichiers d'en-tête problématique dans les fichiers .cpp plutôt que dans les .h. Ceci contourne le problème, car l'analyseur des communications ne lit que les fichiers d'en-tête.

Après la correction de ces erreurs nous avons obtenu une version ELIX fonctionnel. La prochaine étape est de développer le modèle Simtek de l'encodeur. C'est-à-dire, compiler et exécuter le module principal sous la plateforme virtuelle de SpaceStudio.