COMPA backend : Runtime dynamique pour l'exécution de programmes flot de données sur plates-formes multiprocesseurs

Yaset Oliva, Emmanuel Casseau, Kevin Martin, Jean-Philippe Diguet, Thanh Dinh Ngo, Yvan Eustache

Compas'2015 : Parallélisme/ Architecture / Système
Lille, France, du 30 juin au 3 juillet 2015

COMPA backend : Runtime dynamique pour l'exécution de programmes flot de données sur plates-formes multiprocesseurs

Yaset Oliva[1], Emmanuel Casseau[1], Kevin J. M. Martin[2], Jean-Philippe Diguet[2], Thanh Dinh Ngo[2], Yvan Eustache[2]

[1]Univ. Rennes 1, IRISA, Équipe CAIRN, Lannion - Rennes, France
[2]Univ. Bretagne-Sud, UMR CNRS 6285, Lab-STICC, F-56100 Lorient, France

---

Résumé
Ce document présente nos travaux sur le développement d'un flot de conception pour les systèmes embarqués. Plus précisément, nous mettons en place un noyau capable de modifier le système lors de l'exécution pour augmenter le débit de données sans l'aide du concepteur. Le flot de conception prend en entrée une description de l'application en langage RVC-CAL et va jusqu'au déploiement du système sur l'architecture matérielle. En guise de cas d'étude, nous mettons en œuvre un décodeur MPEG-4 sur une architecture matérielle hétérogène et multiprocesseurs.

Mots-clés : Modèles de haut niveau, systèmes embarqués, algorithmes flot de données, codage vidéo reconfigurable, plate-forme multiprocesseurs

---

1. Introduction

L'évolution des standards de communication et de compression vidéo ainsi que les niveaux d'intégration des architectures matérielles, ont fait de la conception des systèmes électroniques modernes un processus ardu. Afin d'aider les concepteurs à accomplir leur tâche, l'utilisation de modèles d'abstraction à haut niveau devient quasiment incontournable. C'est dans ce contexte que des environnements de travail comme Open RVC-CAL Compiler (ORCC) [7] sont apparus.
Le projet COMPA [1] vise à réduire la complexité du processus de conception. COMPA est un projet ANR (Octobre 2011/Juin 2015) soutenu par le pôle « Images et Réseaux ». L'un des objectifs du projet est de proposer des mécanismes « intelligents » capables de rendre un système adaptable durant son fonctionnement sans intervention humaine. Par exemple, un système qui est capable de continuer à fonctionner lors d'une défaillance électronique ou de modifier sa structure pour fournir de meilleures performances.
Nous présentons ici nos travaux autour d'un noyau pour la gestion du système à l'exécution. De nombreux travaux théoriques gravitent autour de cette idée [5]. Plus qu'une tendance, la gestion dynamique, et notamment le mapping, doit faire face à de nombreux défis [5]. L'objectif principal de nos travaux est de créer un banc d'essai pour l'exploration d'algorithmes de mapping dynamiques pour les systèmes multiprocesseurs.

2. Le système mis en œuvre

Une première mise en œuvre du système a été déployée sur une plate-forme Zynq [9]. Un décodeur MPEG-4 Simple Profile est utilisée en tant que modèle d'application. Le but est de décoder un fichier vidéo stocké en mémoire et envoyer les images par le port HDMI de la plate-forme tout en montrant la capacité d'adaptation du système à l'exécution. Le système peut être divisé en trois parties principales, le code applicatif, l'architecture matérielle et les noyaux (Runtimes).

2.1. Le code applicatif

L'élément initial du flot de conception est le modèle de l'application. Ce modèle est défini dans l'environnement ORCC et il est composé d'un réseau d'acteurs, de canaux et du code fonctionnel en langage RVC-CAL. Les acteurs représentent des processus dans lesquels des données sont lues, des calculs sont effectués et des résultats sont écrits. Les canaux représentent des dépendances de données entre les acteurs ; voir [2] pour plus de détails sur les processus flot de données (DPN).
L'environnement ORCC est en charge de la conversion du code RVC-CAL des acteurs dans un langage de programmation plus classique. Divers langages sont ciblés, par exemple VHDL et C, pour lesquels des modules back-end sont définis. Dans notre cas, nous utilisons le back-end C.
En ce qui concerne les canaux, ils sont transformés en FIFO de longueur finie. De plus, pour être en accord avec le modèle flot de données (DPN), le nombre de jetons dans une FIFO doit pouvoir être vérifié. Les acteurs doivent pouvoir également accéder à la valeur des jetons dans la FIFO sans pour autant consommer ces jetons [8].
Pour notre implémentation, des modifications au back-end C de ORCC ont été nécessaires pour générer les éléments suivants :

— Pour chaque acteur, un ficher contenant sa fonction d'ordonnancement (son point d'entrée) et d'autres fonctions internes
— Pour chaque processeur, un fichier contenant les appels aux fonctions du noyau et au point d'entrée des acteurs
— Un fichier d'en-têtes contenant, pour chaque canal, la définition de la FIFO associée et son allocation à une adresse dans la Mémoire Données (voir la description de l'architecture)

Enfin, le nouveau back-end génère un mapping initial. Notons que nous appelons ici mapping l'association entre les acteurs et les processeurs. Plusieurs acteurs peuvent être associés à un processeur, mais un acteur ne peut pas être associé à plusieurs processeurs au même moment.
Le modèle du décodeur MPEG-4 Simple Profile est composé de 17 acteurs et 32 canaux. Au départ, tous les acteurs sont associés au processeur 0 en raison des exigences de mémoire (voir la description de l'architecture). En outre, les adresses pour les FIFOs sont dérivées de l'adresse de base de la Mémoire Données, la taille des FIFO et le nombre de consommateurs de chaque canal. Toutes ces informations doivent être fournies au back-end avant la génération du code.

2.2. L'architecture matérielle

L'architecture matérielle ciblée est composée de plusieurs éléments de calcul (EC). Ils suivent une organisation hiérarchique, un EC joue le rôle de maître et les autres d'esclaves. Le Maître assigne des tâches et donne des ordres aux esclaves qui à leur tour, répondent aux demandes du Maître et exécutent les tâches de l'application.

Le système a été implémenté sur une carte Zc706 de Xilinx. Elle contient une plate-forme Zynq qui intègre un système de calcul (PS) et une partie logique programmable (PL) dans la même puce. L'élément principal du système de calcul est un processeur ARM avec deux cœurs Cortex-A9, tandis que la partie programmable offre un FPGA Kintex-7.
La Fig. 1 représente l'architecture mise en œuvre. Pour notre système, un des deux cœurs Cortex-A9 de l'ARM est utilisé comme Maître et 16 Microblazes (divisés en 2 groupes de 8) sont synthétisés sur le FPGA pour jouer le rôle d'esclaves. La mémoire DDR placée dans la partie calcul est divisée en deux sections. Une première section de 512 Mo est réservée au Maître, alors que la deuxième section, de 512 Mo aussi, est partagée entre les esclaves. La section partagée est nommée Mémoire Code et contient le code applicatif des acteurs. Sur le FPGA, un Microblaze (MB0) est connecté à 512 Ko de mémoire privée via le bus LMB. Les autres esclaves sont connectés à 16 Ko de mémoire. Les mémoires privées sont utilisées pour stocker une copie du Runtime en plus des éventuels acteurs résidents (voir la description du partage du code des acteurs). Les mémoires de Contrôle (4 Ko de BRAM), sont de plus utilisés comme des boîtes à lettres pour les communications entre chaque esclave et le Maître. 128 Ko de BRAM composent la Mémoire de Données. Cette mémoire est accessible à tous les esclaves et contient les FIFO pour les échanges de données entre les acteurs. Les accès aux mémoires partagées sont effectués à travers des bus pré-élaborés (AXI Interconnect). D'autres composants tels que des compteurs et des moniteurs de performance (PMS) sont également utilisés. Les compteurs sont utilisés pour profiler l'exécution des acteurs. Les moniteurs sont utilisés pour observer les interconnexions et fournir des informations comme le nombre de transactions et la quantité de données transférées.

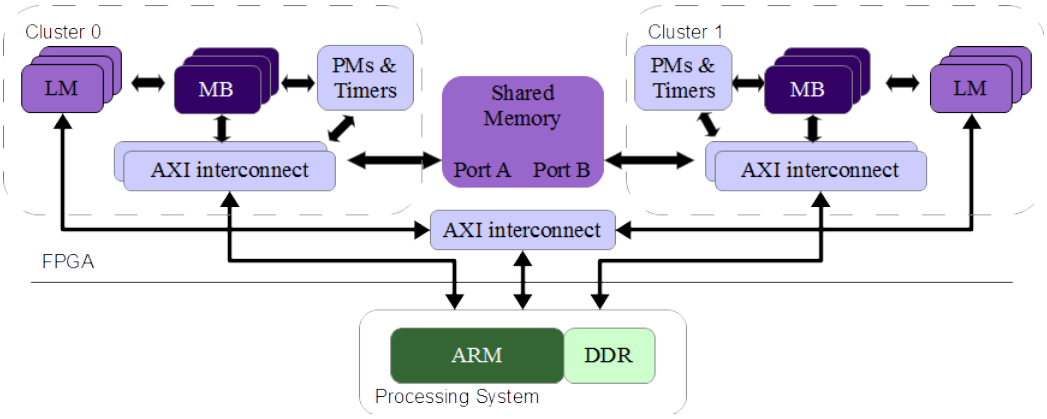


FIGURE 1 – Architecture matérielle

2.3. Noyau Runtime

Le Runtime est responsable du fonctionnement du système.

1. Il fournit une interface pour interagir avec le matériel, par exemple pour initialiser les composants du système (compteurs, moniteurs de performance)
2. Il garantit l'exécution des acteurs en fonction du mapping
3. Il fournit un moyen de communication entre le Maître et les esclaves

Notre Runtime est divisé en deux parties, le Runtime Global exécuté par le processeur Maître et le Runtime Local exécuté par les esclaves. Les communications entre le Maître et les esclaves sont effectuées en utilisant des messages. Ces messages circulent à travers des structures logicielles, nommées FIFOs de contrôle, qui sont implémentées dans les Mémoires de Contrôle. Les Runtimes définissent les messages qui peuvent être échangés ainsi que les routines pour faire l'interface avec les FIFOs de contrôle (c.-à-d. envoie et réception de message). Le tableau 1 résume les principaux messages et les actions effectuées par chaque esclave.

Message	Maître	Esclave
MSG_INIT_DONE	Initialise les compteurs et les moniteurs du système	Initialise l'indicateur InitDone
MSG_ACTORS_MAP	Envoie les indices des acteurs qui ont été associés à chaque esclave	Remplit le tableau d'acteurs Nomades
MSG_CHG_MAP	Indique qu'un nouveau mapping aura lieu	Actualise, en Mémoire Code, les modifications apportées au code des acteurs Nomades assignés préalablement
MSG_GET_METRICS	Demande des informations de profilage (en général) pour calculer un nouveau mapping	Recueille des données de performance et les envoie au Maître

TABLE 1 – Exemples de message Maître - Esclaves et les actions correspondantes

Partage du code des acteurs

Étant donné que notre but primaire est d'explorer les algorithmes de mapping dynamique, un mécanisme de migration d'acteurs est nécessaire. Un mécanisme de migration d'acteurs doit être capable de déplacer l'exécution d'un acteur d'un processeur à un autre sans arrêter le fonctionnement du système. Nous définissons deux types d'acteurs. Les Acteurs Résidents sont ceux qui ne peuvent pas être déplacé hors du processeur auquel ils ont été affectés en raison d'exigences matérielles. Par exemple dans notre système, les exigences de taille mémoire du code des acteurs `Source` et `Display` ne sont remplies que par le processeur MB0. Alternativement, les acteurs Nomades sont ceux qui peuvent être transférés sur n'importe quel processeur.
Un mécanisme de migration simple peut être obtenu en plaçant le code des acteurs dans une mémoire partagée. Notons que la seule exigence à cela est que les processeurs doivent avoir le même jeu d'instructions. Une bibliothèque partagée serait une solution formelle pour partager le code des acteurs parmi les esclaves. Toutefois, la création de bibliothèques partagées n'est pas prise en charge par le Kit de développement logiciel fournit avec notre cible matérielle (Xilinx SDK).
Au lieu de créer une bibliothèque partagée, nous avons créé une bibliothèque statique contenant le code des acteurs et une fonction de saut par acteur. Une fonction de saut est une technique utilisée par les programmes d'édition de liens lors de la création des bibliothèques partagées. La fonction de saut contient juste un appel à une autre fonction définie dans la bibliothèque et que l'on veut exporter. Dans notre cas, nous exportons les fonctions d'ordonnancement (c.-à.-d le point d'entrée) des acteurs. La seule différence entre les fonctions de saut est l'adresse de la fonction exportée, elles ont donc toutes la même longueur en nombre d'instructions après compilation. En plus, en utilisant le mot-clé `__attribut__`, il est indiqué à l'éditeur de liens que chaque fonction de saut doit être placée dans une section spécifique (la

section Saut) de l'exécutable final.
Un exécutable, nommé `ActorsBin`, en plus des exécutables correspondants aux processeurs esclaves est généré et lié à la librairie statique. Dans le script d'édition de liens de chaque esclave, la section Saut est définie à la même adresse. Cela oblige l'éditeur de liens à placer les fonctions de saut aux mêmes adresses dans chaque exécutable (toutes les fonctions de saut ont la même longueur). En outre, les scripts de liaison définissent une autre section (la section Code) pour tous les symboles inclus dans la bibliothèque statique (sauf les fonctions de saut). Au moment du chargement des programmes, seules les sections Saut et Code de l'exécutable `ActorsBin` sont chargées, celles des exécutables esclaves sont ignorées. Ainsi, le code des acteurs est chargé en mémoire une seule fois tout en garantissant que les processeurs esclaves y aient accès. Le chargement est fait par le Maître à l'initialisation du système.

3. Algorithme de mapping dynamique

Le processeur Maître contient un algorithme de mapping dynamique qui permet d'adapter le mapping à la volée en fonction des performances mesurées par le système. Selon les situations, il serait intéressant de pouvoir migrer un acteur d'un processeur à l'autre, ce qui est classiquement fait par un algorithme d'équilibrage de charge dynamique. Cependant, cette migration a un coût. Pour que la migration soit rentable, il faut que le surcoût de la migration ne dépasse pas le gain espéré. En effet, le code binaire des acteurs se trouve en mémoire partagée et chaque processeur possède un cache d'instructions. Le coût de la migration doit donc prendre en compte les défauts de cache lors de la première exécution du code. Concernant les données, celles-ci se trouvent également en mémoire partagée mais le processeur ne possède pas de cache de données. Le surcoût de la migration d'un acteur se restreint finalement au coût de la migration du code. Ainsi, le surcoût d'un nouveau mapping est la somme des surcoûts de chaque acteur migré. L'idée de notre algorithme de mapping dynamique est qu'il n'est pas réaliste de migrer un grand nombre d'acteurs d'un coup. À l'inverse, notre algorithme s'inspire des algorithmes de partitionnement largement utilisé dans le domaine de la conception VLSI. Nous proposons un algorithme de mouvement où un seul acteur peut migrer. Cet algorithme hérite directement de l'algorithme de Fiduccia-Mattheyses [3] (FM). L'algorithme est composé de deux phases. La première phase cherche les acteurs candidats à la migration. La deuxième phase calcule le compromis entre le coût de la migration et le gain en performance.

3.1. Acteurs candidats à la migration

L'algorithme de mapping dynamique suppose un mapping existant. Dans notre cas, un premier mapping peut être obtenu en utilisant un algorithme simple, comme par exemple un algorithme de type « load balancing » (équilibrage de charge de calcul) [4]. L'algorithme que nous proposons reprend l'idée du « cell move » de l'algorithme de FM pour l'adapter en « actor move ». Une adaptation directe de l'algorithme conduirait à étudier tous les mouvements possibles de tous les acteurs sur tous les processeurs. Notre objectif étant d'améliorer le débit du système, il s'agit surtout de soulager le processeur le plus sollicité. Notre adaptation de l'algorithme propose de se concentrer sur les acteurs placés sur ce processeur. La figure 2 illustre comment est représentée la charge sur un processeur. Nous considérons les périodes d'exécution sur chaque processeur. Une période d'exécution est la somme des temps pour le calcul (comp. time) et des temps pour la communication des données à travers le bus (comm. time). Le processeur dont la période est maximum (Maximum period) est celui qui limite le débit du système.
L'algorithme 1 donne les différentes étapes permettant d'estimer le gain de la migration des

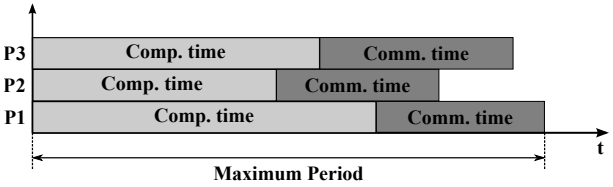


FIGURE 2 – Calcul de la période maximale d'un processeur

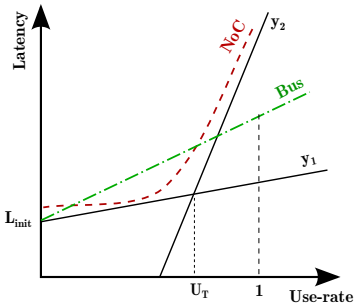


FIGURE 3 – Modèle analytique d'un media de communication (bus ou NoC)

différents acteurs. Le processeur qui possède la période maximale est identifié et une liste d'acteurs candidats à la migration est dressée (ligne 3). Ensuite, pour chacun des candidats, le gain brut (sans surcoût) est établi, et donné par l'équation 1, où le gain est la différence entre la période initiale ($maxPerI$) et la période nouvellement estimée ($maxPerN$). Puis, le coût de la migration est estimé. Enfin, le gain avec surcoût de la migration de l'acteur est calculé. Finalement, l'acteur dont le gain avec surcoût est maximal est retenu (ligne 9).

$$PerG(i) = maxPerI - maxPerN(i) \tag{1}$$

Algorithm 1 Runtime remapping - RR

Input : graph, arch, init_map, profile
Output : a better mapping

1: $get\_mapping\_info(graph, arch, init\_map, profile)$
2: $ActorMove\_estimation(graph, latency, move)$
3: $\mathbb{C} \leftarrow compute\_maxPerI()$
4: **for** $i \in \mathbb{C}$ **do**
5: $\quad Move\_gain(graph, latency, initL, arch, profile)$
6: $\quad PerG(i)$ {equation 1}
7: $\quad Cost_{mig}(i)$ {$lose\_cache\_miss$, equation 2}
8: **end for**
9: $select \max_{i \in \mathbb{C}}(gainT(i))$

3.2. Compromis coût de la migration et gain en performance

Dans notre approche, nous considérons le coût de la migration comme le coût du défaut de cache pour les instructions. Le coût de la migration est donc directement proportionnel à la taille du binaire de l'acteur. Le code de l'acteur se trouvant en mémoire partagée, le rapatriement du code vers le cache du processeur se fait par le bus. Ce bus étant utilisé à la fois pour les données et pour les instructions, plus il est sollicité plus la latence augmente, ce qui impacte le coût de la migration. Nous proposons d'utiliser un modèle analytique du bus pour estimer le temps nécessaire à la migration. La figure 3 illustre un modèle analytique permettant de modéliser à la fois un bus et un NoC. Le temps de communication peut être modélisé par des

fonctions affines (y_1,y_2). Dans le cas d'un bus, seul la fonction y_1 est utilisée. Dans ces travaux, nous nous appuyons sur ce modèle pour considérer un bus. L'aspect NoC n'est pas du tout abordé.
Ainsi, le coût de la migration d'un acteur i est donné par l'équation 2, où Cs^i désigne la taille du code de l'acteur i et f_{cl} est le temps de communication donné par l'équation 3.

$$Cost_{mig}(i) = f_{cl}(x) * Cs^i \tag{2}$$

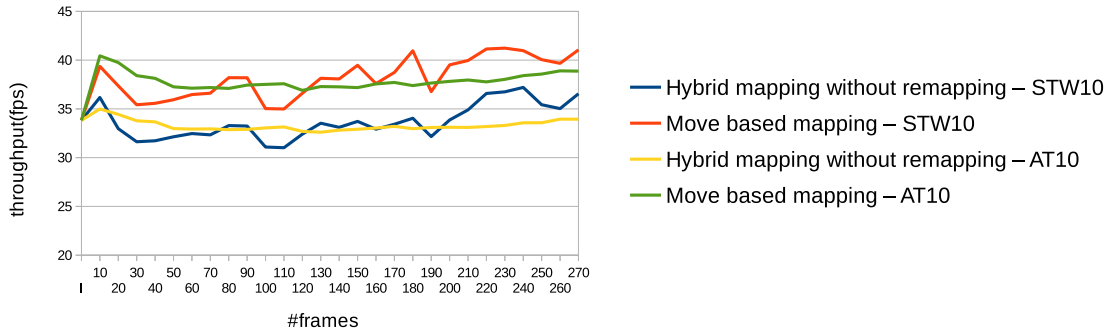
$$f_{cl}(x) = \begin{cases} y_1 & \text{if } x \leq threshold(U_T) \\ y_2 & \text{otherwise} \end{cases} \tag{3}$$

Finalement, le gain apporté par la migration d'un acteur i, en considérant le surcoût de la migration du code étant donné l'activité estimée sur le bus, est donné par l'équation 4.

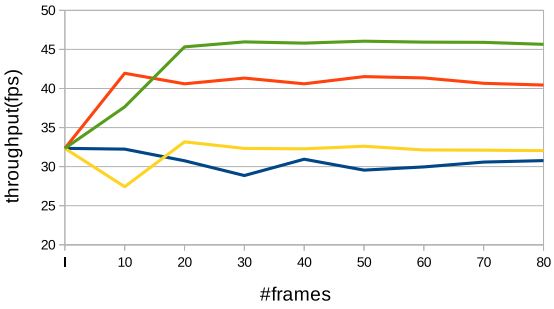
$$gainT(i) = PerG(i) - Cost_{mig}(i) \tag{4}$$

4. Résultats

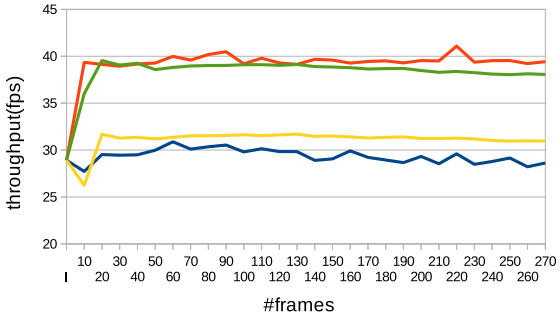
Le tableau 2 montre les ressources matérielles utilisées lors de l'implémentation sur la plateforme Zynq Zc706. La carte d'évaluation Zc706 embarque un Zynq XC7Z045, offrant 350 000 Logic Cells, 218 600 LUTs, 2 180 Ko de BRAM, et 900 Blocks DSP. Cette cible peut accueillir les 16 processeurs microblaze et laisse encore des possibilités d'intégrer des accélérateurs matériels.
Nous avons testé l'algorithme de mapping dynamique sur le décodeur MPEG4-SP. Trois séquences video, *foreman*, *Stefan*, et *Coastguard* issus de [6] ont servi pour les expérimentations. Le format choisi est CIF, une résolution de 352x288 pixels. Un premier profiling est obtenu sur un ordinateur de bureau. Deux collectes ont été effectuées toutes les 10 frames : une basée sur le temps moyen (AT10) et une deuxième basée sur une fenêtre glissante de 10 frames (STW10). Différentes politiques de mapping ont ensuite été appliquées pour observer l'apport de notre approche. Notre algorithme de mapping est comparé pour les deux types de profiling. Les résultats sont donnés par la figure 4. Ainsi, « Move based mapping » désigne notre mapping basé sur la migration d'un acteur, basé sur le temps moyen (AT10) ou sur la fenêtre glissante de 10 frames (SWT10). Ce mapping dynamique est comparé à un mapping dit « Hybride », c'est-à-dire que le choix du mapping est effectué au lancement de la séquence vidéo basé sur le profiling, mais ce mapping reste identique tout le long du décodage. Les figures 4a, 4b et 4c montrent le gain en termes de débit pour les séquences vidéo.
La première observation est une amélioration systématique du débit grâce au mapping dynamique. Cependant, la fenêtre d'observation des temps d'exécution a un impact sur ce gain. En effet, pour la séquence *Stefan*, il apparaît que le meilleur débit est obtenu en se basant sur le temps moyen sur toute la séquence, alors que pour les autres séquences, le débit est légèrement meilleur en considérant la fenêtre glissante. Ceci peut s'expliquer par la « dynamicité » des séquences vidéo. La figure 5 montrent la variation des temps de calcul et de communication pour les différentes séquences vidéo. Le contenu de la séquence impacte directement le débit selon le mapping donné. Il paraît donc évident que pour une même application, ici un décodeur vidéo, le mapping doit être adapté au contenu, ce qui peut être fait à la volée uniquement.
Pour connaître le temps d'exécution de l'algorithme de mapping, nous avons porté l'algorithme sur le processeur ARM de la carte Zynq. Le tableau 3 donne le temps d'exécution moyen de l'algorithme de migration. L'algorithme est exécuté toutes les 10 frames. Étant donné que la complexité de l'algorithme dépend directement du nombre d'acteurs sur le processeur à



(a) Foreman

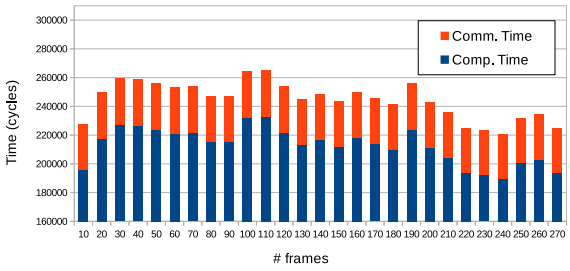


(b) Stefan

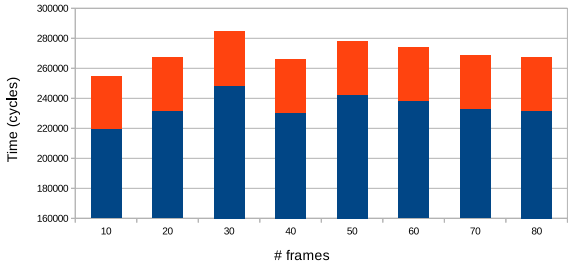


(c) Coastguard

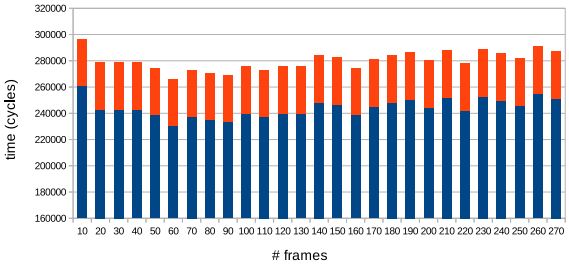
FIGURE 4 – Comparaison du débit de l'application selon les approches de mapping pour les différentes séquences vidéo



(a) Foreman



(b) Stefan



(c) Coastguard

FIGURE 5 – Variation en temps de calcul et de communication pour les différentes séquences vidéo

décharger et du nombre de processeurs cibles, nous avons comparé les différents temps d'exécution. La colonne variance du tableau 3 montre clairement que le temps d'exécution varie très peu. Ce type d'algorithme serait donc aisé à intégrer dans un système à temps contraints où la garantie d'un temps d'exécution est important.

Ressource	Utilisation
Slices LUTs	168 116
Slices Registers	148 502
BRAM36	384
BRAM18	7
DSP	105
Cortex A9	1
DDR 1Go	1

TABLE 2 – Ressources utilisées

Séquence	Temps (ms)	Variance
Foreman	43,54	0,001
Stefan	54,06	0,004
Coastguard	54,01	0,004

TABLE 3 – Temps d'exécution et variance pour différentes séquences d'entrée

5. Conclusion

Ce papier présente une implémentation matérielle de type multi-cœurs allant jusqu'à 16 microblaze sur la carte Zynq Zc706 d'un décodeur vidéo MPEG4-SP et un algorithme de mapping dynamique sur cette architecture. Cet algorithme permet d'améliorer le débit de décodage par rapport à un algorithme statique, où le mapping n'évolue pas au fil du temps. Les résultats montrent clairement que pour une même application, le placement des acteurs doit être adapté à la séquence vidéo pour atteindre des débits élevés.

Bibliographie

1. COMPA, conception orientée modèle de calcul pour multi-processeurs adaptables, http ://www.compa-project.org.
2. Lee (E.) et Parks (T.). – Dataflow process networks. *Proceedings of the IEEE*, vol. 83, n5, May 1995, pp. 773–801.
3. Lim (S. K.). – *Practical Problems in VLSI Physical Design Automation*. – Springer Publishing Company, Incorporated, 2008, 1 édition.
4. Ngo (D.-T.), Diguet (J.-P.), Martin (K.) et Sepulveda (D.). – Communication-model based embedded mapping of dataflow actors on heterogeneous mpsoc. – In *Proceedings of the Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2014.
5. Singh (A.), Shafique (M.), Kumar (A.) et Henkel (J.). – Mapping on multi/many-core systems : Survey of current and emerging trends. – In *2013 50th ACM / EDAC / IEEE Design Automation Conference (DAC)*, pp. 1–10, mai 2013.
6. Xiph.org video test media, http ://media.xiph.org/video/derf/.
7. Yviquel (H.), Lorence (A.), Jerbi (K.), Cocherel (G.), Sanchez (A.) et Raulet (M.). – Orcc : Multimedia development made easy. – In *Proceedings of the 21st ACM International Conference on Multimedia*, *MM '13*, MM '13, pp. 863–866, New York, NY, USA, 2013. ACM.
8. Yviquel (H.), Sanchez (A.), Jaaskelainen (P.), Takala (J.), Raulet (M.) et Casseau (E.). – Efficient software synthesis of dynamic dataflow programs. – In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pp. 4988–4992, May 2014.
9. Zynq, http ://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.