Génération automatique de code distribué à l'aide de RTOS : application au codage d'images LAR

Ghislain ROQUIER Mickaël RAULET Jean-François NEZAN Olivier DÉFORGES

IETR Groupe Image et Télédétection UMR CNRS 6164

INSA de Rennes
20, avenue des buttes de Coësmes, 35043 Rennes, FRANCE
{groquier, mraulet, jnezan, odeforges}@insa-rennes.fr

Résumé

Les futures générations de téléphones mobiles, incluant toujours plus de services multimédia, representent un vrai défi en terme de systèmes temps-réels embarqués de part leurs besoins toujours croissants en fléxibilité et en puissance de calcul. Les architectures multi-composants programmables peuvent alors apporter une solution efficace et évolutive. Le but de nos travaux consiste à développer un processus de développement rapide et automatique spécialement adapté aux architectures multi-composants hétérogènes. Cet article présente un processus de développement basé sur la méthodologie Adéquation Algorithme Architecture (AAA), de la description conjointe d'une application et d'une architecture jusqu'aux exécutifs distribués temps-réel. Nous montrerons ensuite une génération automatique d'exécutifs issus de la méthodologie AAA basée sur l'utilisation de systèmes d'exploitations temps-réel résident (Real-Time Operating System - RTOS). Nous comparons cette approche avec celle sans RTOS en termes de complexité et de performance. Finalement, ce travail est illustré par l'exécution d'une application multimédia basée sur le codec LAR.

Mots clefs

Prototypage rapide, temps-réel, système embarqué, codec LAR

1 Introduction

Les systèmes multimédias modernes requièrent une puissance de calcul toujours plus importante et donc des contraintes d'embarquabilité plus difficiles à satisfaire. D'autre part, leurs temps de développement doivent sans cesse être réduits. Dans ces systèmes, la limitation de la puissance de calcul est souvent palliée par l'utilisation de circuits spécifiques dédiés. Cependant cette solution est difficilement compatible avec un temps de développement court et ne peut pas être mise à jour efficacement. Une alternative peut être apportée par l'utilisation de composants logiciels (DSP, ARM) ou matériels (FPGA) puisqu'ils ont l'avantages d'être programmables et réutilisables. Néanmoins, les aspects parallèles et hétérogènes d'architectures multi-composants laissent apparaître de nouveaux problèmes en termes de distribution et d'ordonnancement des applications sur les différents composants. Une solution de conception appropriée consiste à utiliser une méthodologie de prototypage rapide permettant, à partir d'une description de l'application temps-réel de haut niveau, l'implantation optimisée et automatique sur l'architecture cible. La vocation de la méthodologie Adéquation Algorithm Architecture (AAA) présentée ici est de répondre à ces besoins. Le but de la méthodologie AAA est de générer automatiquement des exécutifs distribués temps-réel à partir des descriptions respectives de l'application et de la cible matérielle. Les exécutifs sont ordonnancés hors-ligne et sont alors particulièrement bien adaptés aux systèmes déterministes, comme par exemple les algorithmes de traitement des images, ainsi qu'aux architectures multi-composants hétérogènes.

L'ordonnancement hors-ligne rend superflu l'utilisation d'un RTOS. En effet, un ordonnancement en-ligne implique plus de données sur le composant. Il est bien adapté lorsque le comportement de l'application ne peut être prédit, comme lors de traitements sur des événements apériodiques [1]. Lorsque le comportement de l'application est parfaitement déterministe, un ordonnancement hors-ligne est suffisant et peut être implémenté par un simple séquenceur de calcul. Un RTOS utilise des ressources trop souvent limitées dans un système embarqué. Néanmoins, une comparaison entre ces deux types d'implantations doit être effectuée pour déterminer l'impact sur la mémoire allouée, l'effet sur les communications entre composants ainsi que sur le comportement temps-réel de l'application. Cet article est organisé comme suit : le 2e chapitre introduit la méthodologie AAA. Les spécifications des exécutifs et l'utilisation de RTOS selon la méthodologie AAA sont décrites dans le 3e chapitre. L'implantation d'une application basée sur le codec vidéo LAR et une discussion sur les résultats sont donnés dans le 4e chapitre. Enfin les conclusions seront détaillées dans le 5e chapitre.

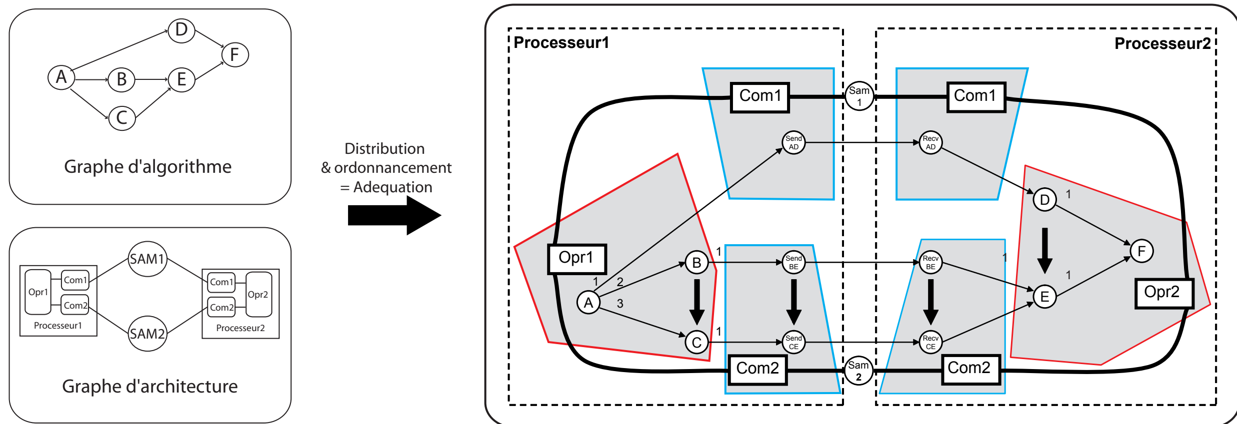

FIG. 1 – *Placement et ordonnancement d'un algorithme sur une architecture*

2 Méthodologie AAA

Le but de la méthodologie AAA consiste à trouver le meilleur placement et ordonnancement d'un algorithme sur une architecture multi-composant La méthodologie AAA trouve son fondement dans la théorie des graphes. L'algorithme et l'architecture sont décrits par deux graphes distincts qui révélent le parallélisme potentiel de l'algorithme et le parallélisme disponible de l'architecture. L'adéquation est une suite de transformations effectuées sur ces deux graphes qui aboutit à une implantation optimisée de l'algorithme au sens de la latence.

2.1 Modèles d'algorithme et d'architecture

L'algorithme de l'application est modélisé par un graphe flot de données (GFD) qui est un hyper-graphe orienté. Chaque sommet et chaque arête représentent respectivement une opération de l'algorithme et un transfert de données entre opérations. Un GFD révèle un ordre partiel pour l'exécution des opérations : deux opérations sans relation de dépendance de donnée peuvent être exécutées dans un ordre arbitraire et plus particulièrement, elles peuvent être exécutées simultanément par deux processeurs distincts. Le GFD permet donc d'exhiber le parallélisme potentiel d'un algorithme [2].

Dans la méthodologie AAA, afin d'être précis dans la description sans être trop complexe au niveau matériel, la machine à états finies est définie comme le composant atomique de l'architecture. Ainsi un processeur ou un circuit dédié peut être vu comme une composition de machines à états finies. Une architecture multi-composant est alors representée par un réseau d'automates finis interconnectés à l'aide de media de communication (bus, mémoires partagés ...). Une architecture peut être représentée par un graphe non-orienté où chaque sommet et chaque arête sont respectivement un processeur et un media de communication. Dans ce modèle, un processeur est composé d'un opérateur et autant de communicateurs que de media connectés. Un opérateur exécute une partie de l'algorithme et un communicateur execute un opération de communication lorsqu'un transfert de données est requis. L'opérateur et les différents communicateurs sont reliés entre eux à travers une mémoire partagée du processeur.

La figure 1 représente un graphe d'architecture composé de deux processeurs connectés via deux media. Chaque processeur est constitué d'un operateur et de deux communicateurs.

2.2 Transformations de graphe

Le graphe d'implantation est obtenu par transformation du graphe d'algorithme et du graphe d'architecture. Cette transformation correspond à la distribution et à l'ordonnancement de l'algorithme. La distribution, également appelée partitionnement, alloue spatialement les différentes opérations du graphe d'algorithme sur les opérateurs du graphe d'architecture. Cela revient à diviser le graphe d'algorithme en plusieurs sous-graphes decrivant les opérations que chaque opérateur doit exécuter. L'ordonnancement utilise les dépendances de données du GFD pour allouer dans l'espace temporel les opérations sur les opérateurs. Cette transformation revient à définir la séquence d'exécution des opérations sur un opérateur comme le montre la figure 1. Les parties grisées représentent les différents opérateurs et communicateurs. Les dépendances de données définissent des précedences : ici, A est exécutée avant B. Lorsque les opérations n'ont pas de dépendances de données, des précédences sont insérées pour éviter les interblocages. Elles sont représentées par les flèches en gras. Cela implique dans notre exemple que B doit être exécutée avant C.

Le graphe d'implantation est obtenu par une optimisation simultanée de la distribution et de l'ordonnancement. Un grand nombre d'implantations est envisageable, le problème d'optimisation consiste à détermininer l'implantation la plus efficace en terme de respect des contraintes temps-réel. L'optimisation de la distribution et d'ordonnancement sur une architecture multi-composant est un problème NP-difficile, *i.e.* une recherche exhaustive de toutes les solutions possibles est inconcevable. Une heuris-

tique est donc utilisée pour trouver une approximation de la solution optimale dans un temps raisonnable. Cette heuristique [3] de type gloutonne vise à minimiser la latence de l'algorithme exécuté sur une architecture multi-composant.

2.3 Génération d'exécutif

Une fois le graphe d'implantation optimisée déterminé, un exécutif peu être automatiquement généré pour chaque opérateur. Avant cela, quelques modifications doivent être apportées à ce graphe. Tout d'abord, l'aspect répétitif de l'application ainsi que les synchronisations entre opérateurs doivent être ajoutés dans le graphe d'implantation optimisé. En effet, les applications réactives à implanter sont itératives par nature alors que le GFD ne permet pas de faire ressortir le caractère répétitif d'une application. Des boucles sont alors insérées dans chaque séquence d'opérations et de communications. L'ordonnancement ne fait pas non plus apparaître les synchronisations entre les différents séquenceurs d'un processeur. Des sémaphores sont donc insérés afin de garantir la précedence entre opérations de calcul et de communication sur un même processeur [4]. Une description plus détaillée est donnée dans [2]. Le réseau de Petri de la figure 2 représente les synchronisations entre les différents séquenceurs du premier processeur de notre précedent exemple où *P* et *V* [1] respectivement attend un sémaphore et envoie un sémaphore. Les paramètres d'un sémaphore définissent les opérations à synchroniser, le chemin vers le communicateur (A_1 par exemple), l'état du tampon (plein ou vide) et enfin le nom du media utilisé.

Dans cet exemple, l'opération *A* est exécutée et le résultat de ce calcul stocké dans le tampon mémoire *AD* pour être envoyé sur le communicateur *Com1*, comme défini par l'ordonnancement. Un sémaphore $\{A_1, 1, SAM1\}$ est alors envoyé au communicateur de telle sorte que l'opération de communication *send(AD)* de *AD* par *Com1* ne soit pas réalisé avant la fin de l'opération *A*. L'opération *send(AD)* envoie le contenu du tampon vers *processor2*. Un sémaphore $\{A_1, 0, SAM1\}$ est envoyé vers *OPR1* en fin de transfert pour que celui ci n'accède pas aux données avant qu'elles n'aient été envoyées. Si la séquence de communication est plus rapide, *Com1* devra alors attendre de nouveau que les données *AD* soient calculées par *A* sur *OPR1*. Notons que ces synchronisations sont générées automatiquement dans AAA.

Une fois le graphe d'exécution déterminé, il doit ensuite être transformé en autant de macro-exécutifs que de processeurs présents sur l'architecture. Un exécutif générique est composé d'une liste de macro-instructions qui permettent de spécifier les allocations mémoires, les synchronisations, les séquences de communications ainsi que les séquences de calculs. Ces macro-instructions sont dites génériques, c'est-à-dire qu'elles ne dépendent d'aucun langage spécifique de programmation. Cela permet de rester à un haut niveau d'abstraction sans se soucier de la cible matérielle envisagée. Une ultime transformation, détaillée section 3, permettra la création d'exécutifs dans le langage spécifique à la cible matérielle (C ou assembleur pour DSP et GPP, VHDL pour FPGA).

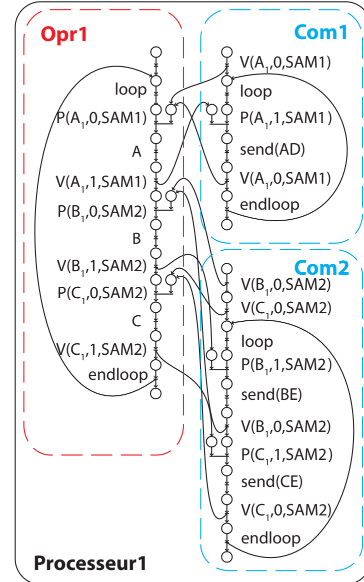

FIG. 2 – *Réseau de Pétri du graphe d'exécution*

2.4 SynDEx

Le logiciel SynDEx[2] est un outil de CAO niveau système librement téléchargeable. Il est principalement développé à l'INRIA Rocquencourt avec notre participation. Cet outil supporte la méthodologie AAA pour le prototypage rapide et l'implantation optimisée d'applications temps-réel distribuées sur architectures multi-composants. SynDEx permet de générer des exécutifs génériques qui respectent la méthodologie AAA.

3 Spécification des exécutifs

Les exécutifs génériques créés selon le méthodologie AAA doivent être transformés afin d'obtenir des exécutifs compilables qui pourront être ensuite chargés sur les éléments de la cible matérielle.

3.1 Traduction

Chaque exécutif générique est traduit en un exécutif compilable à l'aide d'un macro-processeur. Le macro-processeur transforme la liste de macro-instructions en un code source propre à la cible matérielle envisagée. Cela consiste à remplacer chaque macro-instruction par une définition. Chaque définition étant spécifiée à travers des bibliothèques (aussi appelée noyaux) dépendantes du processeur cible ou encore du media de communication reliant deux processeurs. Plusieurs types de bibliothèques

[1] ***P**robeer* et ***V**erhoog* signifient *décrémente* and *incrémente* en Néerlandais

[2] **Syn**chronized **D**istributed **Ex**ecutive

existent, elle permettent de donner des définitions propre à l'architecture comme les allocations mémoire, les synchronisations entre séquences ou encore les transferts de communications. D'autres bibliothèques sont plus proches de l'algorithme et permettent de traduire par exemple les prototypes ou les appels de fonctions. Puisque les macro-exécutifs sont génériques, il existe un grand nombre de traductions possibles amenant à un code source compilable. Cette phase est réservée à l'utilisateur qui doit choisir la traduction la plus appropriée à la cible envisagée. Le logiciel libre GNU-M4 est le macro-processeur que nous utilisons pour la phase de traduction. Il faut remarquer que dans tous les cas (avec ou sans RTOS), l'exécutif temps-réel distribué est statique et défini hors-ligne. Le comportement temps-réel et les synchronisations sans interblocages entre les différentes séquences sont garantis par construction.

3.2 Real-Time Operating Systems dans le contexte AAA

L'approche présentée dans cet article consiste à comparer deux types de traductions. Dans un premier temps, une traduction "classique" dans le sens de AAA est réalisée [5]. Cette traduction consiste à traduire sans l'aide d'un RTOS l'exécutif générique dans le langage de programmation approprié. Dans un deuxième temps, un autre traduction est réalisée mettant en jeux un RTOS. Cette approche consiste lors de la traduction à configurer un RTOS résident capable d'exécuter les différentes séquences et de gérer les synchronisations entre elles. A cette fin, de nouvelles bibliothèques de traduction ont été developpée. La figure 3 represente les deux approches proposées, et ce de la description haut niveau de notre application jusqu'aux basses couches matérielles.

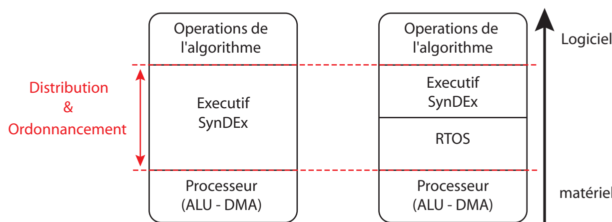

FIG. 3 – *Du logiciel jusqu'au matériel : deux approches*

Un RTOS est un système d'exploitation adapté aux applications temps-réel. Il permet de définir et de contrôler plusieurs tâches au sein d'un même processus. Des mécanismes de communication et de synchronisation entre tâches existent à cet effet. Dans cette approche, il faut considérer les différentes séquences de calcul et communications obtenues par la méthodologie AAA comme différentes tâches distincts du processus. Dans l'approche "classique", les sémaphores sont représentés par des booléens gérés "manuellement" par l'utilisateurs, alors que dans la deuxième approche, les sémaphores sont gérés par le RTOS pour synchroniser les séquenceurs. Il faut remarquer que le RTOS est uniquement utilisé pour la gestion des différentes tâches ainsi que pour le contrôle des synchronisations. Cependant, il faut aussi noter que l'intégration d'un RTOS dans un cible matérielle amène un surcoût. Surcoût en mémoire et en temps d'exécution lié à la gestion de contexte par le RTOS.

4 Travaux réalisés

4.1 Aperçu des RTOS

Un grand nombre de RTOS existent pour différents types de processeurs. Leurs primitives sont souvent spécifiques à une famille particulière de processeurs. *A contrario*, un RTOS plus générique et indépendant d'une famille de processeurs semble mieux approprié pour une implantation rapide sur un matériel varié. Le Linux embarqué semble avoir cet atout. En effet, ce RTOS a l'avantage d'être conforme à la norme POSIX, ce qui rend la programmation de tel RTOS plus aisée et independante de la cible. Pour les DSP de chez TI plusieurs RTOS Linux embarqué existent tels que MediaLinux OS [6], Lightweight OS [6]. Pour le moment, DSP-BIOS, le RTOS propriétaire de TI, est le RTOS que nous utilisons pour les applications que nous developpons. Il faut remarquer que DSP-BIOS n'est utilisable que sur les DSP de TI et que ses primitives sont spécifiques à ce RTOS.

4.2 Plateforme cibles

Plusieurs fournisseurs de matériel tels que Pentek, Sundance ou Vitec MM developpent des architectures multi-composant et quelques unes d'entre elles ont été validées pour supporter la méthodologie AAA. Pour le moment, une plateforme Sundance a été utilisée pour accueillir le RTOS résident. Ces plateformes sont composées d'un PC et d'une carte PCI. Cette carte peut être composée de différents modules interconnectés par différents media. Le module SMT361 est constitué de DSP C6416 bien adaptés pour le traitement des images et le module SMT319 est constitué d'un DSP C6414 connecter à des circuits intégrés permettant la conversion numérique-analogique pour un affichage ou une acquisition de l'image au format PAL. La plateforme utilisée est représentée sur la figure 6 composée de deux modules SMT361 et d'un module SMT319.

4.3 Résultats préliminaires

Afin de déterminer la différence entre les deux approches, une application de communication entre deux DSP a été testé. Les résultats sont donnés sur les figures 4 et 5. Le 1er graphe montre le temps nécessaire selon les deux approches pour exécuter l'application en fonction de la taille des données. Le 2e graphes nous donne le rapport entre les temps d'exécution des deux approches. Le processus utilisant le RTOS est toujours plus lent que celui ne l'utilisant pas. Cependant, plus la taille des données est grande, plus cet impact temporel est réduit. Ainsi, le temps d'exécution de l'application ne semble pas excessivement augmenté par l'utilisation d'un RTOS lorsque les données

sont grandes comme par exemple en traitement des images. En plus de cela, le code source généré est moins grand et plus compréhensible comparé à l'autre méthode. Le debogage durant la phase de vérification fonctionnelle de l'algorithme en est donc facilité.

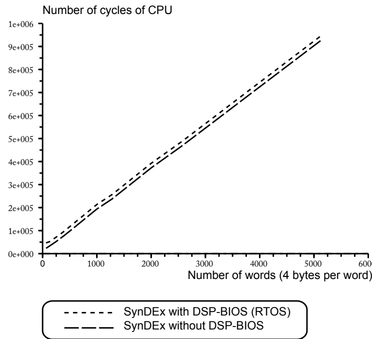

FIG. 4 – *Comparaison des temps d'exécution*

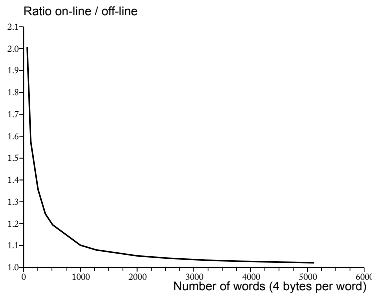

FIG. 5 – *Rapport entre les temps d'exécution des deux approches*

4.4 Implantation du codec LAR

Le LAR[3] est un algorithme de compression et de décompression vidéo developpé dans notre laboratoire [7], bien adapté pour la transmission d'image. Le principe de cette technique est d'adapter la résolution locale (tailles des pixels) selon l'uniformité de la luminance, typiquement une basse résolution (bloc de 16×16 pixels) pour une zone où la luminance est uniforme et au contraire une forte résolution (bloc de 2×2 pixels) pour une zone présentant de forte singularité. C'est un codec vidéo scalable permettant aussi bien un codage en niveaux de gris très bas débit qu'une compression de vidéo couleur sans perte.

L'objectif est de réaliser une implantation de ce codec sur notre architecture multi-composant selon la méthodologie AAA. Le codeur et le décodeur sont respectivement implantés sur le 1er et le 2e DSP de la plateforme. Le GFD du codec LAR est représenté sur la figure 7. Trois différents algorithmes scalables du LAR sont implantés afin de comparer les résultats lorsque la complexité est croissante [8].

Algorithme 1 : codec vidéo spatial pour la luminance (caractérisé par des blocs 2×2, 4×4 et 8×8).

[3] **L**ocally **A**daptive **R**esolution

algorithme	sans RTOS	avec RTOS
1	18.03 ms	18.05 ms
2	25.35 ms	25.45 ms
3	31.84 ms	32.07 ms

TAB. 1 – *Temps d'exécution des algorithmes LAR*

algorithme	Codeur		Decodeur	
	sans RTOS	RTOS	sans RTOS	RTOS
1	874 kB	928 kB	899 kB	953 kB
2	747 kB	802 kB	658 kB	713 kB
3	642 kB	697 kB	528 kB	583 kB

TAB. 2 – *Mémoire utilisée par le codec*

Algorithme 2 : codec vidéo spatial pour la chrominance ajouté à l'algorithme 1.

Algorithme 3 : codec vidéo spectral pour les blocs 2×2 (caractérisé par l'ajout de l'erreur résiduelle) ajouté à l'algorithme 2.

Les résultats obtenus de l'implantation des différents algorithmes est conforme aux résultats obtenus précédemment. Le temps d'exécution du codec est légèrement plus lent avec l'utilisation d'un RTOS (cf TAB. 1). Le codeur est la partie de l'algorithme la plus lente (typiquement 4 fois plus lent que le décodeur), c'est pourquoi le comportement temps-réel du codec LAR est donné par le temps d'exécution du codeur puisque l'exécution totale est "pipeliné" par les processeurs. L'utilisation du RTOS a aussi un impact sur la mémoire des processeurs (55 kO). Cependant, plus la mémoire est utilisée, plus l'impact devient proportionnellement petit . Pour l'algorithme 3, DSP-BIOS augmente la mémoire utilisée par le codeur et par le décodeur de respectivement 7% et 5% (cf TAB. 2).

5 Conclusion et perspective

Cet article a permis de présenter l'intégration d'un système d'exploitation temps-réel lors de la génération automatique d'exécutifs de la méthodologie AAA. La méthodologie a été introduite afin de présenter la génération automatique d'exécutifs distribués temps-réel. Le developpement de bibliothèques lors de la spécification des exécutifs nous a permis d'utiliser un RTOS et de l'implanter dans l'architecture cible PC-multi-DSP.

Bien que le RTOS ait un impact autant sur la mémoire allouée que sur le temps d'exécution d'un processus, nous avons constaté que ce surcoût devenait faible lorsque la taille des données traitée augmentait. Ainsi, l'utilisation d'un RTOS semble presque aussi efficace que l'approche utilisée auparavant pour le traitement des images où les données sont souvent grandes. D'ailleurs, les exécutifs générés avec des primitives du RTOS est plus simple et permet donc une meilleure compréhension pour l'utilisateur. Ce point est très important pour que notre génération automatique soit toujours utilisable dans le futur. L'implantation du codec vidéo LAR avec l'utilisation d'un

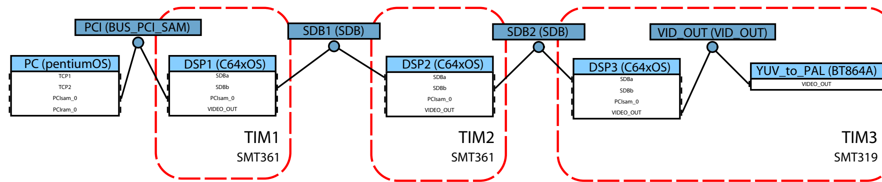

FIG. 6 – *Platforme cible*

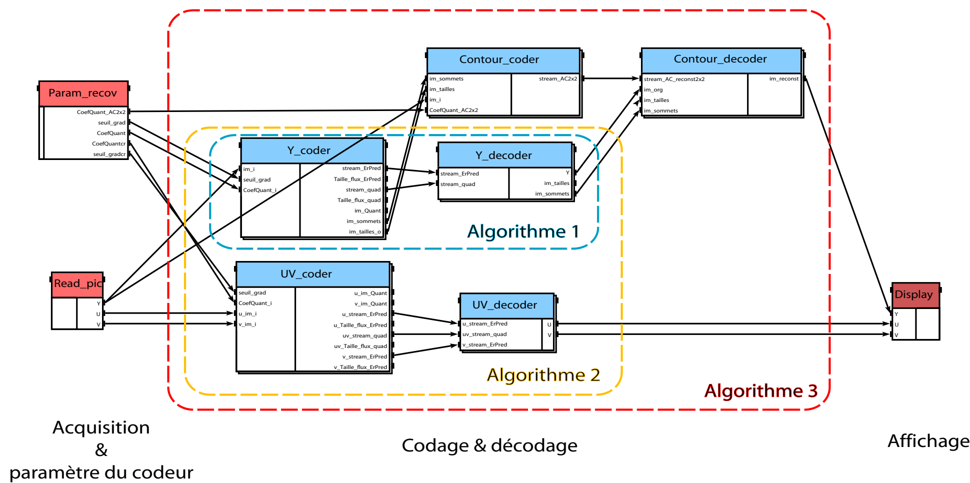

FIG. 7 – *description GFD du codec LAR*

RTOS résident nous a permis de vérifier le comportement temps-réel de l'algorithme nécessaire dans le contexte de systèmes embarqués.

Nous travaillons quant à l'intégration d'un RTOS plus générique, tels que ceux utilisant la norme POSIX, et indépendant du processeur cible. Cela permettra un développement plus rapide de processus multitâche sur des architectures variées à l'aide de la même spécification du RTOS. Ce travail permettra l'implantation de nouveaux algorithmes de traitement d'image en développement dans notre laboratoire tels que le standard MPEG-4 AVC ainsi que le standard MPEG-21 SVC.

Références

[1] F. Balarin and *al.* Scheduling for embedded real-time systems. *IEEE Design and Test of Computers*, 15(1) :71–82, Janvier-Mars 1998.

[2] T. Grandpierre and Y. Sorel. From algorithm and architecture specifications to automatic generation of distributed real-time executives : a seamless flow of graphs transformations. In *First ACM and IEEE International Conference on Formal Methods and Models for Co-Design, Mont Saint-Michel, France*, Juin 2003.

[3] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real time embedded heterogeneous multiprocessors. In *proc. of IEEE CODES'99 7th Int. Workshop on Hardware/Software Co-Design, Rome, Italie*, Mai 1999.

[4] Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages : NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[5] M. Raulet, F. Urban, J-F. Nezan, O. Déforges, and C. Moy. Syndex executive kernels for fast developments of applications over heterogeneous architectures. In *EUSIPCO'05, Antalaya, Turquie*, Septembre 2005.

[6] J. Kretzschmar and R.Baumgartl. Lightweight rtai for dsps. In *1st Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), Palma de Mallorca, Espagne*, Juin 2005.

[7] O. Déforges and J. Ronsin. Region of interest coding for low bit-rate image transmission. *IEEE International Conference on Multimedia and Expos (ICME), New-York, USA*, Août 2000.

[8] M. Raulet, F. Urban, M. Babel, O. Déforges, J-F. Nezan, and Y. Sorel. Automatic coarse-grain partitioning and automatic code generation for heterogeneous architectures. In *IEEE Workshop on Signal Processing Systems (SIPS'03), Seoul, Corée*, Septembre 2003.