RenPar'19 / SympA'13 / CFSE'7
Toulouse, France, du 9 au 11 septembre 2009

Observation de systèmes embarqués : une approche à base de composants

Carlos Prada-Rojas* [1], Vania Marangozova-Martin[1], Kiril Georgiev*[1],
Jean-François Méhaut[1], Miguel Santana[2]
[1]Prenom.Nom@imag.fr, [2]Miguel.Santana@st.com

Résumé

Dans ce papier, nous présentons EMBera, un modèle à composants pour l'observation de systèmes embarqués multiprocesseurs (MPSoC). Le modèle EMBera est indépendant de l'architecture matérielle et permet d'obtenir des informations d'observation multi-niveaux. Nous démontrons la validité du modèle par une implémentation sur une architecture embarquée à 5 processeurs produite par la société STMicroelectronics. Nous discutons les performances de notre implémentation et menons une réflexion sur les fonctionnalités d'observation de base à intégrer.

Mots-clés : systèmes embarqués, MPSoC, observation, composants

1. Introduction

La conception d'architectures matérielles et logicielles pour les systèmes embarqués devient aujourd'hui un vrai défi [18]. En effet, pour répondre aux besoins d'innovation et de puissance de ces systèmes, les fabricants suivent actuellement la mouvance des plates-formes multiprocesseurs/multicœurs (*Multiprocessor System on Chip - MPSoC*). Ils doivent faire face à des changements radicaux dans leurs supports logiciels d'exécution, dans leurs outils de développement et dans les modèles de programmation utilisés. D'autant plus que, pour répondre aux besoins concurrentiels du marché, le développement de nouvelles plates-formes doit se faire dans des délais de plus en plus courts.

Historiquement, la mise au point des MPSoC a toujours impliqué un développement du matériel et du logiciel très intégré. Les supports d'exécution et les outils de développement qui en ont résulté sont dédiés aux architectures matérielles sous-jacentes. Porter de tels logiciels sur des architectures nouvelles est une tâche longue et complexe qui devient un facteur limitant pour les fabricants de MPSoC. La question qui se pose actuellement est quels sont les modèles, les outils et les techniques de développement qui permettraient d'assurer une chaîne de production qui réutiliserait au maximum le logiciel déjà existant tout en l'adaptant de manière efficace aux nouvelles générations d'architectures.

Dans ce papier nous nous intéressons aux outils d'observation qui font partie intégrante du processus de développement pour MPSoC. En effet, le débogage et l'optimisation des logiciels embarqués est basée sur l'observation et l'interprétation d'événements qui se produisent à l'exécution. Le débogage a pour but d'éliminer le fonctionnement erroné des logiciels, alors que l'optimisation permet d'améliorer (minimiser) leur consommation de ressources. Pour un système embarqué qui a des ressources limitées et des contraintes temps réel, il est très important de minimiser le temps d'exécution (par exemple pour le décodage vidéo), de minimiser la consommation énergétique (pour la longévité de la batterie) et de minimiser la taille de la mémoire occupée.

* Bourse CIFRE STMicroelectronics

Pour proposer des outils d'observation qui puissent être utilisés sur des architectures MPSoC différentes, nous investiguons une approche basée sur les composants logiciels. Largement acceptés dans le domaine des systèmes distribués [17] [12], nous soutenons qu'ils offrent également une solution adaptée aux problèmes de développement et de déploiement des systèmes embarqués. En effet, nous pensons que leur capacité d'encapsulation permettrait à la fois de nous abstraire de l'hétérogénéité matérielle des architectures MPSoC et de traiter les différents modèles de programmation parallèle qui seraient utilisés. Leur adhérence naturelle aux principes de séparation des aspects nous permettrait de séparer le fonctionnement applicatif des logiciels embarqués des traitements d'observation.

Dans ce travail, nous proposons un modèle à composants simple que nous utilisons pour l'observation des couches logicielles d'un système embarqué. Nous discutons les fonctionnalités d'observation de base que nous considérons nécessaires et détaillons l'implémentation de notre modèle sur une plate-forme embarquée fournie par la société STMicroelectronics. Nous montrons que notre modèle permet d'obtenir des informations d'observation à différents niveaux logiciels (du système à l'application) et discutons les aspects de performances et d'intrusivité.

Le papier est organisé comme suit. Dans la section 2 nous présentons un bref état de l'art des travaux liés à l'observation des systèmes. Notre modèle à composants, EMBera, est présenté dans la section 3. Dans la section 4 nous décrivons l'implémentation de ce modèle sur la plate-forme *STi7200*. Nous analysons les différentes mesures d'observation que nous avons pu obtenir et les performances de cette implémentation dans la section 5. Nous présentons nos conclusions et perspectives de travail dans la section 6.

2. État de l'art

L'observation des systèmes est un problème classique traité dans de multiples domaines. Dans cette section nous nous concentrons principalement sur les travaux dans trois domaines : les systèmes embarqués, les systèmes parallèles et les systèmes à composants.

Le développement intégré du logiciel et du matériel dans les systèmes embarqués a produit des logiciels propriétaires de bas niveau (pilotes, systèmes d'exploitation). Les outils d'observation ne font pas exception. Dans leur majorité, ils sont dédiés à des plates-formes spécifiques et fournissent de l'information de très bas niveau comme l'état de la mémoire (valeurs des registres, contenu de la mémoire) ou les événements noyau (interruptions, appels système). Ils ne fournissent pas d'informations sur les couches logicielles supérieures et même s'ils le font, ils ne font pas le lien entre les événements des différentes couches. Des exemples de ce type d'outils sont KPTrace [15] ou SpyKer [10]. Développés respectivement par STMicroelectronics et LynuxWorks, ils permettent de tracer des applications développées pour un système d'exploitation Linux.

Étant donné que les systèmes sur puce deviennent *de facto* des systèmes parallèles, nous ne pouvons pas ignorer les solutions d'observation dans le domaine des architectures parallèles. En effet, dans les systèmes à mémoire partagée, nous trouvons des outils d'observation pour les programmes multithreadés [19, 13] ou écrits en OpenMP [11]. Dans les systèmes à mémoire distribuée, il existe des outils pour surveiller les programmes MPI [9, 14]. Toutefois, ces outils ne peuvent être utilisés tels quels puisque les applications embarquées n'appliquent pas ces modèles de programmation.

Les systèmes à base de composants logiciels proposent une approche d'observation différente puisqu'ils se focalisent sur les couches logicielles supérieures, notamment les applications utilisateur et les intergiciels. Les informations fournies concernent typiquement l'architecture à base de composants et leurs interactions. Le modèle à composants Fractal [3], par exemple, peut énumérer les composants en exécution et donner leurs interconnexions. OpenCCM [12], une implémentation *open source* du modèle à composants de CORBA, capture l'invocation de méthodes et ainsi surveille la communication entre composants. L'approche à base de composants est intéressante puisqu'elle permet de fournir des mécanismes d'observation qui sont indépendants du matériel. Cependant, l'observation des couches système, qui est indispensable dans le cadre des systèmes embarqués, n'est que rarement considérée.

Pour rapprocher les systèmes à composants et les systèmes embarqués, une solution possible est d'utili-

ser des systèmes d'exploitation à base de composants logiciels. Néanmoins, des systèmes comme Pebble [8] ou Flux OSKit [7] ne supportent pas les architectures embarquées. Le projet PURE [2] a pour cible les systèmes embarqués mais il s'intéresse plus à la conception qu'à l'observation de ces systèmes. Think [6] est une proposition de système d'exploitation à base de composants, ayant aussi comme cible les systèmes sur puce, dont l'observation n'est pas un de ces buts. Néanmoins, cette proposition a été la base de Nomadik Multiprocessing Framework [5] développé par STMicroelectronics, dans lequel notre travail doit être intégré.

3. Modèle d'observation EMBera

Les objectifs principaux du modèle d'observation EMBera sont les suivants :

- *Généricité* : Le modèle EMBera doit être indépendant de l'application et de ce fait pouvoir être utilisé pour observer différents types d'applications embarquées.
- *Observation multi-niveaux* : Le modèle doit permettre l'observation de différents niveaux logiciels liés à l'exécution d'une application embarquée. Typiquement, il doit pouvoir fournir de l'information sur les événement système, sur les opérations effectuées au niveau de l'intergiciel, ainsi que sur les interactions applicatives.
- *Support de l'hétérogénéité matérielle* : Le modèle doit pouvoir être utilisé pour l'observation d'applications s'exécutant sur différentes plates-formes matérielles.
- *Performances* : EMBera doit pouvoir être optimisé pour fournir une observation efficace du système embarqué. D'une part, l'implémentation d'EMBera devrait être faite de sorte à ce qu'elle utilise le moins de ressources possible. D'autre part, l'intégration d'EMBera dans un système embarqué devrait introduire une perturbation minimale.

En utilisant les composants logiciels pour le processus d'observation, nous ne voulons en aucun cas imposer aux applications embarquées un modèle de programmation orienté composants. En effet, étant donné l'utilisation récente des architectures multicœurs dans les systèmes embarqués, le domaine entier est encore à la recherche d'un modèle de programmation adapté. Il est actuellement difficile de dire vers quel modèle les applications vont évoluer : programmation à base de composants, programmation multithreadée, programmation parallèle à la OpenMP ou autre.

Dans notre travail actuel, nous avons choisi une approche simple qui consiste à réécrire les applications en termes de composants. Nous prévoyons d'investiguer comment notre modèle pourrait être réutilisé dans le cadre d'applications écrites autrement, surtout dans les cas d'applications OpenMP ou MPI.

Le modèle EMBera s'inspire du modèle à composants Fractal [3]. Fractal est un modèle général qui peut être mis en œuvre dans différents langages de programmation et peut être employé au niveau du système, de l'intergiciel ou de l'application. Fractal est un projet *ObjectWeb*[2] et de ce fait profite des interactions et de la visibilité au sein d'une large communauté. Un deuxième avantage majeur de Fractal est qu'il est déjà employé chez STMicroelectronics qui définit notre contexte de travail.

3.1. Le modèle à composants EMBera

Une application EMBera est un ensemble de composants connectés entre eux. Un composant est une entité logicielle avec une fonctionnalité bien définie. Une partie de cette fonctionnalité peut être visible par d'autres composants, dans ce cas le composant définit des interfaces *fournies*. Certains composants peuvent dépendre de cette fonctionnalité, dans ce cas, ils définissent des interfaces *requises*. Les *connexions* entre les composants sont établies en reliant les interfaces requises aux interfaces fournies.

Les composants d'une application communiquent à l'aide de messages. Les interfaces peuvent donc être vues comme des points de communication qui sont utilisés pour l'envoi et la réception de messages. La communication est basée sur deux primitives simples qui sont le `send` et le `receive`. La primitive `send` est utilisée pour envoyer, de manière non bloquante, un message depuis une interface requise vers une interface fournie. La primitive `receive`, bloquante, sert à recevoir des messages.

[2] http ://www.ow2.org/ : ObjectWeb est un consortium qui promue les projets d'intergiciel distribué open source

Les composants dans EMBera sont des entités actives et chaque composant a son propre flot d'exécution. Ce choix suit la pratique courante pour les applications MPSoC selon laquelle des traitements sont exécutés sur différentes unités de calcul.

Les composants EMBera fournissent une interface prédéfinie pour le contrôle du composant. Les opérations de contrôle incluent la création du composant, l'interconnexion des composants et la gestion du cycle de vie de composant (lancement et arrêt).

3.2. L'observation dans EMBera

Nous avons décidé de modéliser explicitement l'observation dans EMBera. À cette fin, nous avons défini une nouvelle interface de contrôle consacrée à l'observation. Les composants EMBera se retrouvent, par conséquent, avec deux interfaces de contrôle prédéfinies : une pour le contrôle général et une pour l'observation.

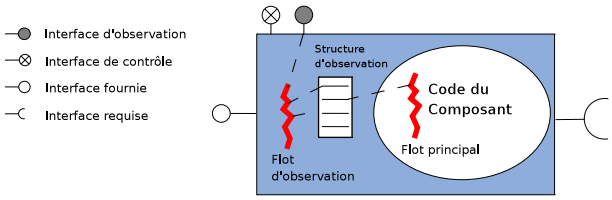

FIG. 1 – Vue schématique d'un composant

Pour minimiser l'intrusion due à l'observation, nous avons décidé d'introduire un deuxième flot d'exécution dans le composant. Ce flot d'exécution est chargé de récupérer les informations concernant l'exécution du composant et de les enregistrer dans une structure dédiée. Cette structure est en effet une ébauche de la structure nécessaire à la génération d'une trace de l'exécution du composant.

Nous avons défini une interface d'observation minimaliste qui contient des fonctions d'observation concernant le temps d'exécution et la mémoire utilisée. Nous considérons en effet, que ce type d'information est à la base de tout processus d'observation pour le débogage ou l'optimisation. Comme nous pensons que l'interface doit fournir des informations sur au moins trois niveaux, notamment le système d'exploitation, l'intergiciel et l'application, nous avons défini les fonctions de l'interface en fonction de ces trois niveaux.

Observation au niveau du système Les fonctions fournissant des informations au niveau du système d'exploitation sont les suivantes :

- `getComponentExecutionTime` : cette fonction sert à récupérer le temps d'exécution d'un composant qui est défini comme le temps d'exécution de son flot d'exécution principal.
- `getComponentMemoryUsed` : cette fonction permet de récupérer la mémoire utilisée par un composant à un moment de l'exécution de l'application.

Observation au niveau de l'intergiciel à composants Les fonctions liées au fonctionnement de la couche qui s'occupe de la gestion des composants sont les suivantes :

- `getComponentCreationTime` : cette fonction donne le temps pris par la fonction de contrôle qui s'occupe de la création d'un composant. La mesure prend en compte le temps de création de la structure du composant, ainsi que l'initialisation de ses flots d'exécution.
- `getSendAverageTime` et `getReceiveAverageTime` : ces fonctions mesurent respectivement le temps moyen d'un envoi et d'une réception de message. Étant donné que les primitives `send` et `receive` peuvent être implémentées de manières différentes et que les messages envoyés vont dépendre des interfaces des composants, les valeurs fournies par ces fonctions sont fortement liées à l'application et à l'implémentation d'EMBera. Elles vont dépendre de la taille des messages, du nombre de messages échangés, de la localisation des composants sur les différents processeurs, des mécanismes de synchronisation utilisés, etc.

Observation au niveau de l'application Au niveau de l'application nous fournissons des fonctions d'introspection qui renseignent sur son architecture. Actuellement nous avons la fonction `getComponentInterfaces` qui énumère les interfaces requises et fournies par un composant en spécifiant pour les interfaces requises leur état (connectée/non connectée).

3.3. Exemple d'application : le décodeur MJPEG

Illustrons EMBera en considérant une application de décodage d'un flux d'images JPEG [3]. Dans le processus de décodage, chaque image est divisée en blocs. Chaque bloc est décodé en appliquant un algorithme de Huffman, un ré-ordonnancement des pixels et une transformation discrète inverse de cosinus (IDCT). À la fin, tous les blocs sont réordonnés afin de reconstituer les images originales.

Notre implémentation de l'application MJPEG introduit un composant `Fetch`, un composant `Reorder` et plusieurs composants `IDCT`. `Fetch` s'occupe du découpage en blocs, les `IDCT` effectuent des calculs d'IDCT en parallèle et `Reorder` reconstitue les images. L'architecture résultante est donnée à la FIG. 2.

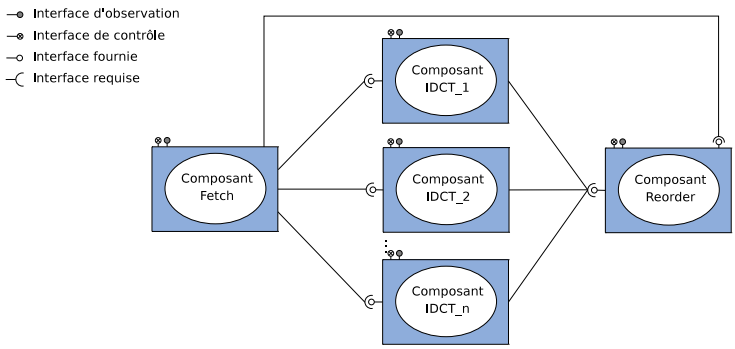

FIG. 2 – Application de décodage MJPEG avec des composants

4. Implémentation d'EMBera sur MPSoC

Nous avons implémenté le modèle à composants EMBera ainsi que le modèle d'observation sur deux plates-formes différentes : une plate-forme x86 SMP[4] à 16 cœurs et une plate-forme embarquée hétérogène. La plate-forme embarquée, *STi7200*, contient 5 cœurs et fait partie des produits actuels de STMicroelectronics. Dans la suite, nous décrivons l'implémentation d'EMBera sur cette dernière plate-forme.

4.1. Architecture matérielle de STi7200

STi7200 (cf. FIG. 3) contient une unité centrale de calcul générique *RISC ST40* à 450Mhz et quatre processeurs *ST231* à 400Mhz. Le processeur central *ST40* a accès à toute la mémoire de la puce qui est une mémoire SDRAM de 2 Go. Chaque processeur *ST231* a accès à un bloc de cette mémoire qui représente sa mémoire locale. La communication entre processeurs se fait en utilisant un autre bloc de la mémoire qui est partagé et lié à un contrôleur d'interruptions. Au niveau le plus bas, la communication entre processeurs passe toujours (est acheminée et contrôlée) par le processeur principal *ST40*.

La plate-forme *STi7200* dispose d'un port JTAG au travers duquel il est possible de déployer directement les différents processeurs. Ce port s'utilise à l'aide d'un boîtier spécifique, appelé *STMC2*.

4.2. Environnement logiciel sur STi7200

La programmation sur *STi7200* se fait en utilisant une implémentation propriétaire du standard ANSI C. Cette implémentation fait partie d'un ensemble d'outils STMicroelectronics comprenant des compila-

[3] L'application a été mise en œuvre pour [1], dans le cadre du développement d'une plate-forme de simulation *cycle-accurate*

[4] Symmetric multiprocessing : machine ayant plusieurs unités de calcul (cœurs) qui accèdent à de la mémoire partagée

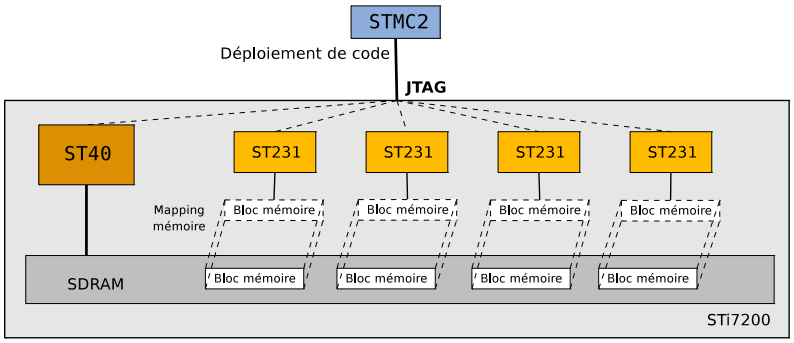

FIG. 3 – Plate-forme STi7200

teurs, des éditeurs de liens et des outils pour le débogage. Comme les processeurs *ST40* et *ST231* ont des jeux d'instructions différents, chacun dispose de son propre ensemble d'outils de développement.

Les processeurs de la plate-forme *STi7200* exécutent OS21, un système d'exploitation multitâches temps réel [16]. OS21 fournit des modules de gestion des flots d'exécution (tâches), de la mémoire et du temps. Le système fournit également tous les mécanismes nécessaires à la gestion des interruptions, des exceptions et de la synchronisation.

Les tâches OS21 sont identiques à des processus dans le sens où ce sont des structures qui gèrent un flot d'exécution avec des données propres et une pile. Néanmoins, contrairement aux processus, une tâche peut accéder à la mémoire d'une autre tâche s'exécutant sur le même processeur. Le système d'exploitation OS21 est lui-même géré comme une tâche. Pour créer une nouvelle tâche OS21, il est nécessaire de spécifier la région de la mémoire (appelée *partition*) qui va être utilisée par la tâche.

Les tâches OS21 s'exécutant sur un même processeur de la *STi7200* communiquent en accédant au bloc mémoire de ce processeur. Les tâches s'exécutant sur des processeurs différents communiquent à l'aide d'un intergiciel spécifique, appelé `EMBX`. Elles communiquent à l'aide de deux primitives : `EMBX_Send` (non bloquante) et `EMBX_Receive` (bloquante). Ces fonctions servent respectivement à écrire et à lire dans des régions de mémoire partagée, appelées *objets distribués*. Pour créer un objet distribué, une tâche doit utiliser deux primitives : `EMBX_Alloc` et `EMBX_RegisterObject`. La première fonction alloue la mémoire nécessaire pour l'objet dans le bloc mémoire du processeur correspondant. La deuxième primitive crée une référence qui peut être utilisée par les tâches s'exécutant sur d'autres processeurs. Cette référence leur est envoyée à l'aide de la fonction `EMBX_Send`.

Toute communication `EMBX` entre tâches s'exécutant sur des processeurs différents, nécessite l'établissement d'un canal de communication de bas niveau entre ces processeurs et la création de *ports* de communication. Ainsi, si une tâche *ST40* doit communiquer avec une tâche sur un *ST231*, il doit y avoir un couple de *ports* portant le même nom, un port par processeur, qui vont "acheminer" la communication entre les tâches.

4.3. Implémentation du modèle EMBera

Nous avons implémenté le modèle EMBera en langage C, puisqu'il est considéré comme le standard pour le développement de logiciel embarqué.

Dans la suite, nous allons détailler l'implémentation du modèle EMBera sur *STi7200*, la programmation des composants EMBera et le déploiement des applications sur la plate-forme.

4.3.1. Implémentation des composants EMBera

Un composant EMBera est composé d'une structure de données et de deux tâches OS21. La première tâche, que nous appelons *tâche principale*, est en charge du fonctionnement applicatif du composant. La deuxième tâche est dédiée à l'observation. La structure de données contient principalement les références aux tâches et aux fonctions que ces tâches vont exécuter.

La création de composants se fait avec la fonction `createComponent` qui initialise la structure de données et crée les tâches correspondantes. Les tâches d'un composant sont créées avec la fonction OS21 `task_create` et sont mises en attente à l'aide la fonction OS21 `task_suspend`. En effet, les composants ne doivent pas être lancés avant que leurs interfaces soient créées et connectées.

Les interfaces fournies et requises d'un composant sont créées avec la fonction `createInterface`. Une interface fournie est représentée par un objet distribué (voir section 4.2 ci-dessus), alors qu'une interface requise est représentée par une référence vers un objet distribué. La communication entre interfaces, faite à l'aide de `send` et de `receive` entre composants se traduit par des appels aux fonctions `EMBX_Send` et `EMBX_Receive`.

Nous établissons les liens entre les interfaces requises et fournies des composants avec la fonction `connect`. Cette fonction consiste à affecter à l'interface requise la référence de l'objet distribué représentant l'interface fournie. Dans le cas où les deux composants s'exécutent sur le même processeur, la récupération de la référence est triviale. Dans le cas où ils s'exécutent sur deux processeurs différents, la récupération de la référence est plus complexe. En effet, le composant qui détient l'interface fournie doit faire connaître la référence auprès du composant à interface requise. Pour ce faire, le premier composant envoie cette référence en utilisant le canal de communication et le port créés pour permettre la communication entre les processeurs (cf. section 4.2). Dans `connect` s'effectuent l'attente de cette référence, la récupération et l'affectation à l'interface requise.

Lorsque les connexions sont établies, la fonction `startComponent` démarre les composants. Cette fonction débloque les tâches du composant à l'aide de la fonction OS21 `task_resume`. Pour arrêter l'exécution des composants, la fonction `componentWait` est fournie. Basée sur la primitive OS21 `task_wait`, elle attend la fin de la tâche principale du composant.

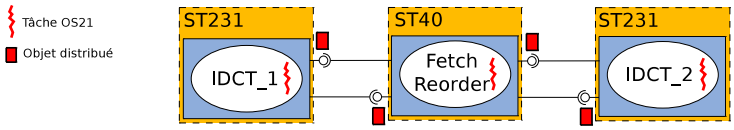

FIG. 4 – Déploiement de l'application de décodage MJPEG sur la plate-forme STi7200

Reprenons l'exemple de l'application MJPEG. Pour des raisons liées au matériel (expliquées dans la section 4.3.3), nous avons utilisé trois processeurs. Nous avons créé trois composants : un composant `Fetch-Reorder` et deux composants `IDCT` (cf. FIG.4).

Les composants `IDCT` sont identiques à ceux montrés dans l'architecture initiale de l'application (cf. FIG.2). Le composant `Fetch-Reorder` est une fusion des composants `Fetch` et `Reorder` montrés précédemment. Les `IDCT` sont déployés sur 2 *ST231* puisque ils exécutent des fonctions de calcul intensif, alors que `Fetch-Reorder` s'exécute sur le *ST40* car il effectue des operations d'entrée et sortie.

4.3.2. Programmation des composants EMBera

La programmation d'un composant consiste à fournir le code d'une fonction qui sera exécutée par la tâche principale de ce composant. Le code est un code C standard, à l'exception de l'utilisation des primitives `send` et `receive` pour l'envoi et la réception de messages. FIG. 5 présente un extrait simplifié de la fonction exécutée par un des composants `IDCT` de l'application MJPEG.

4.3.3. Déploiement des composants EMBera

Afin de déployer une application EMBera sur la plate-forme *STi7200*, nous générons un code binaire différent pour chacun des processeurs. Chaque code binaire est indépendamment chargé sur son processeur cible. Pour les expériences présentées dans la section 5, le chargement a été effectué manuellement. Chaque code binaire contient une fonction `main` permettant d'établir l'architecture en termes de composants déployée sur ce processeur.

```
#include "embera.h"
#include "os21support.h"
void *idct_function (void *args) {
        ...
        msgBuffer = receive(provIDCT0Fetch);         // IDCT0 receives input data
        ...
        while (strcmp(msgBuffer->msgType, "EOF") {   // End of messages verification
                ... computeIDCT(...); ...
        }
        send (msgResult, reqIDCT0Fetch);             // IDCT0 sends results to Fetch-Reorder
}
```

FIG. 5 – Extrait du code exécuté par le composant IDCT0

Regardons le programme de déploiement utilisé sur un des processeurs *ST231* dans l'exemple de l'application MJPEG.

```
#include "embera.h"
#include "os21support.h"
extern void *idct_function (void *args);
int main () {
        os21_start();
        os21PreDeployment("st231video0","st40");
        ...
        component *IDCT0 = createComponent (20, "IDCT0", idct_function);
        interface *provIDCT0Fetch = createInterface(IDCT0, "provIDCT0Fetch", "provided");
        interface *reqIDCT0Fetch = createInterface(IDCT0, "reqIDCT0Fetch", "required");
        ...
        connect(reqIDCT0Fetch, port);
        startComponent(IDCT0);
        componentWait(IDCT0);
        return(0);
}
```

FIG. 6 – Extrait du code `main` exécuté sur le processeur ST231

Dans ce programme (cf. FIG.6), nous initialisons tout d'abord le noyau OS21 (`os21_start`) et établissons les canaux de communication entre les processeurs (`os21PreDeployment`). La fonction `os21PreDeployment` crée deux ports portant le même nom, un sur le processeur *ST40* et un sur le processeur *ST231*. Ces ports vont être utilisés pour toutes les communications par `EMBX_Send` et `EMBX_Receive` entre des tâches sur ces deux processeurs. En effet, l'établissement des ports est fortement lié à la future architecture de l'application et peut être considéré comme un prédéploiement.

Une fois l'initialisation du système effectuée, nous créons les composants, leurs interfaces et les connectons. Ici nous créons un composant `IDCT0` qui a une interface fournie `provIDCT0Fetch` et une interface requise `reqIDCT0Fetch`, la création de `provIDCT0Fetch` se traduisant par la création d'un objet distribué sur le *ST231*. Pour connecter l'interface requise `reqIDCT0Fetch` à l'interface fournie du composant `Fetch-Reorder` s'exécutant sur *ST40*, nous avons besoin d'effectuer des communications entre les deux processeurs et pour cela utiliser le port défini précédemment. Le programme termine par le lancement du composant (`startComponent`) et la mise en attente de sa terminaison (`componentWait`).

Pour charger le code sur les processeurs, nous nous servons du boîtier *STMC2*, de l'interface JTAG (voir FIG.3) et d'un outil de débogage. Nous avons utilisé trois processeurs, le *ST40* et deux *ST231*, sur les cinq fournis par la plate-forme puisque la version du boîtier *STMC2* dont nous disposons ne peut être connectée qu'à trois processeurs en même temps.

4.4. Implémentation de l'interface d'observation d'EMBera

L'interface d'observation est implémentée comme un couple d'interfaces (l'une fournie et l'autre requise). L'interface d'observation fournie reçoit des messages demandant de l'information tandis que l'interface d'observation requise renvoie l'information demandée. Ce couple d'interfaces est géré par la tâche du composant dédiée à l'observation.

Pour implémenter les fonctions d'observation, nous nous sommes fortement basés sur des fonctions fournies par le système OS21. La plus importante de ces fonctions est `task_status` qui renseigne sur

la taille de la mémoire allouée et utilisée, sur le temps d'exécution d'une tâche depuis son lancement et sur l'état de la tâche (en attente ou en exécution). Étant donné que les tâches sur un même processeur ont accès à la mémoire des autres tâches, nous pouvons récupérer les informations sur la tâche principale d'un composant depuis la tâche d'observation.

Nous avons implémenté les fonctions d'observation comme suit :

- `getComponentExecutionTime` : Le temps d'exécution d'un composant est le temps d'exécution de la tâche principale de ce composant. Nous obtenons la mesure en utilisant la fonction `task_status`. Le résultat est rendu en *ticks* que nous convertissons en secondes à l'aide de la fonction OS21 `time_ticks_per_sec`.
- `getComponentMemoryUsed` : La mémoire utilisée par un composant est la somme de la mémoire utilisée par sa tâche principale et la taille de sa structure de données. Étant donné que la taille de la structure est négligeable dans notre implémentation, nous mesurons la taille mémoire de la tâche avec `task_status`.
- `getComponentCreationTime` : La création d'un composant consiste à créer sa tâche principale et allouer sa structure de données. Les mesures du temps sont faites avec la fonction OS21 `time_now`.
- `getSendAverageTime` et `getReceiveAverageTime` : Pour obtenir ces mesures, nous considérons tous les appels de `send` et de `receive` dans le code des composants. Cette fonctionnalité est activée/désactivée explicitement à la compilation d'une application EMBera.
- `getComponentInterfaces` : L'information sur les interfaces d'un composant est directement contenue dans sa structure.

5. Résultats expérimentaux

Nous avons observé l'exécution de l'application MJPEG sur un cas de décodage d'un fichier d'entrée de 578 images JPEG (taille du fichier 2.8 Mo). Les expérimentations effectuées portent sur une dizaine d'exécutions. Dans la suite, nous présentons les mesures obtenues aux trois niveaux d'observation considérés, c'est à dire le système, l'intergiciel et l'application (voir section 3.2).

5.1. Mesures au niveau système

Le tableau TAB.1 présente les mesures concernant les temps d'exécution des composants de l'application. Nous observons que les composants `IDCT` finissent leur exécution bien plus rapidement que le composant `Fetch-Reorder`. Ceci vient naturellement de la logique de l'application qui impose à `Fetch-Reorder` d'effectuer des traitements avant et après la terminaison des `IDCT`. D'autre part, même si le processeur *ST40* a une vitesse d'horloge supérieure à celle des *ST231*, ces derniers possèdent des caractéristiques qui les rendent plus performants en ce qui concerne les opérations multimédia.

Composant	Sans observation (s)	Avec observation (s)	Surcoût (%)
Fetch-Reorder	83.30	87.14	4.6
IDCT0	6.42	7.36	14.7
IDCT1	7.19	8.47	17.8

TAB. 1 – Temps d'exécution des composants du décodeur MJPEG

Nous constatons que l'intrusion due à l'observation en ce qui concerne le temps d'exécution est acceptable pour le composant `Fetch-Reorder` (4.6 %). Ce n'est pas le cas des composants `IDCT` pour qui le surcoût s'élève jusqu'à 17.8 %, même si les traitements d'observation sont identiques sur tous les processeurs. Nous pensons que ce coût est lié à la performance des *ST231* en ce qui concerne les changements de contexte impliqués dans l'ordonnancement des tâches des composants. Nous envisageons dans nos futurs travaux d'alléger les traitements d'observation sur les *ST231*, et d'en reporter le coût sur le *ST40*. En ce qui concerne l'écart entre les temps d'exécution des composants `IDCT`, nous pensons qu'il est dû au lancement manuel des exécutions.

Afin de connaître l'évolution de la consommation mémoire, nous prenons périodiquement des mesures

de la mémoire utilisée par les composants `Fetch-Reoder` et `IDCT0`. Nous observons que la mémoire utilisée par `Fetch-Reorder` (FIG. 7 (a)) augmente rapidement dans un premier temps, puis ralentit. Nous pouvons faire correspondre cette empreinte mémoire au comportement du composant, lequel initialise des données au début de son exécution et effectue ensuite un calcul itératif dans lequel il ne libère pas de mémoire.

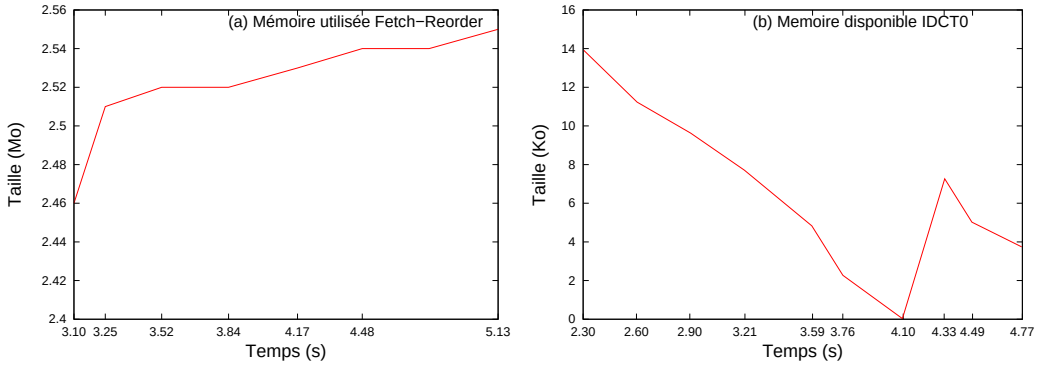

FIG. 7 – Évolution de la mémoire des composants

L'utilisation de la mémoire par le composant `IDCT0` est présentée différemment. FIG. 7 (b) montre la place disponible dans la partition mémoire de la tâche principale du composant. La tâche occupe de plus en plus de mémoire, jusqu'à occuper la totalité de la partition. Dans cette situation, OS21 élargit dynamiquement la partition mémoire.

Le surcoût lié à la mémoire (voir TAB.2) est négligeable pour le composant s'exécutant sur le *ST40* (au maximum il est de 1 %) et important sur les accélérateurs *ST231* (au moins 7%). Ceci nous mène aux mêmes conclusions que celles que nous avons eues pour le temps d'exécution, notamment qu'il est important de repenser l'architecture d'observation et d'alléger les traitements sur les *ST231*.

Composant	Taille au lancement (Ko)	Taille en fin d'exécution (Ko)	Taille structures d'observation (Ko)	Surcoût (%)
Fetch-Reorder	2 460	14 870	24	$1 \rightarrow 0$
IDCT0 ou IDCT1	132	332	24	$18 \rightarrow 7$

TAB. 2 – Mémoire utilisée par les composants et les structures d'observation dans l'application MJPEG

5.2. Mesures au niveau intergiciel

Nous avons mesuré le temps de création des composants. Nous avons pu constater qu'il est consacré à la création des tâches. Pour le composant `Fetch-Reorder`, ce temps est de $57ms$, séparé en $33ms$ pour la création de la tâche principale (58%) et $23ms$ pour la création de la tâche d'observation (42%). Pour les composants `IDCT`, le temps de création est plus long ($81ms$) : $51ms$ pour la tâche principale (63%) et $30ms$ pour la tâche d'observation (27%). Ces mesures reflètent l'architecture de la plate-forme *STi7200* : étant donné que le processeur *ST40* gère la plupart des opérations liées au contrôle de périphériques, ce processeur doit être plus performant dans les opérations de gestion des tâches.

En ce qui concerne les mesures des temps de communication, nous observons par exemple que les temps d'envoi des messages augmentent exponentiellement avec la taille des messages. Il est à noter que ces temps évoluent différemment sur les deux types de processeurs *ST40* et *ST231*.

5.3. Observation au niveau application

Nous avons comptabilisé le nombre de `send` et de `receive` afin de quantifier la communication entre composants. Nos avons obtenu 10386 appels (autant de `send` que de `receive`) pour le composant `Fetch-Reorder` et 5193 appels pour les composants `IDCT`. Le composant `Fetch-Receive` exécute en effet le double d'opérations comparé aux `IDCT` ce qui nous donne des indications sur les schémas algorithmiques utilisés dans l'application MJPEG.

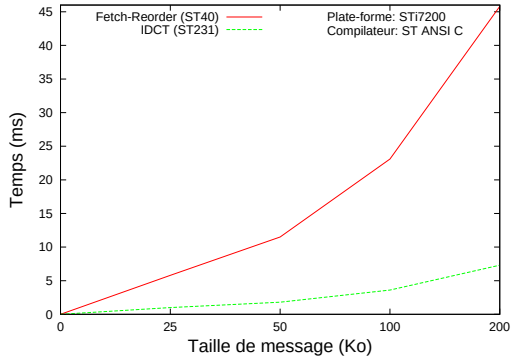

FIG. 8 – Temps de communication en utilisant la primitive `send`

Nous avons également utilisé le mécanisme d'observation fourni par EMBera pour obtenir la liste des interfaces du composant. FIG. 9 montre que le composant `Fetch-Reorder` possède 6 interfaces : un couple d'interfaces consacrées à l'observation (une fournie et une requise), ainsi que deux couples pour la communication avec `IDCT0` et `IDCT1`.

```
Interfaces du composant [Fetch]
---------------------------
[Interface]  [Type]
observation  provided
_fetchIdct0  provided
_fetchIdct1  provided
observation  required
_fetchIdct0  required
_fetchIdct1  required
```

FIG. 9 – Interfaces du composant Fetch-Reorder

6. Conclusion

Dans ce document, nous avons présenté EMBera, un modèle à base de composants pour l'observation de systèmes embarqués. Nous avons défini le concept d'interface d'observation et les fonctions de base pour une observation multi-niveaux. Nous avons décrit l'implémentation d'EMBera sur une plate-forme à 5 cœurs de STMicroelectronics et avons montré, à l'aide d'une application de décodage vidéo, les différentes informations qui peuvent être obtenues. Nous avons recueilli des informations relatives au système d'exploitation, à l'intergiciel à composants et à l'application. Nous avons montré qu'il est possible de calculer des informations ponctuelles comme le temps d'exécution d'un composant, mais aussi d'avoir un suivi comme pour l'évolution de l'utilisation de la mémoire.

Nos mesures de performances et surtout d'intrusivité ont montré que le mécanisme d'observation est très sensible à l'architecture matérielle de la plate-forme. Ceci nous amène à la conclusion que le mécanisme d'observation ne doit pas être identique sur tous les cœurs d'un MPSoC mais doit prendre en compte leurs disparités.

La suite de notre travail porte sur la définition plus précise des fonctions d'observation. Typiquement, nous voudrions avoir plus d'informations concernant les ressources matérielles (compteurs, énergie), ainsi que plus d'informations sur l'interaction et la logique applicative. Nous nous intéresserons également aux liens qu'il faut établir entre les informations provenant de différents niveaux. Un point obligatoire à développer est la production de traces d'exécution où se poseront les questions de taille de la trace, d'événements à sélectionner pour la trace, du support d'enregistrement, etc.

En ce qui concerne l'implémentation d'EMBera, nous devons trouver des moyens afin d'optimiser notre prototype. Nous avons développé EMBera à partir de rien mais avons la possibilité d'utiliser Cecilia [4], qui est l'implémentation C de référence de Fractal. Adopter Cecilia nous apporterait plus de visibilité et d'applications pour notre modèle mais supposerait de modifier Cecilia pour y intégrer un modèle d'exécution (flots d'exécution) qui correspond aux architectures embarquées.

A plus long terme il serait intéressant de voir comment des applications parallèles écrites en OpenMP ou en MPI pourraient être observées avec EMBera. Il faudrait arriver à proposer une encapsulation de ces applications dans des composants et fournir, en utilisant les principes génériques d'observation du modèle, les mêmes informations que les outils spécifiques existants.

Bibliographie

1. Ivan Augé, Frédéric Pétrot, François Donnet, and Pascal Gomez. Platform-Based Design From Parallel C Specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(12) :1811–1826, December 2005.
2. Danilo Beuche, Abdelaziz Guerrouat, Holger Papajewski, Wolfgang Schroder-preikschat, Olaf Spinczyk, and Ute Spinczyk. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In *In 2nd IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '99*, pages 45–53, 1999.
3. Éric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The Fractal Component Model and its Support in Java. *Software – Practice and Experience (SP&E)*, 36(11-12) :1257–1284, September 2006. Special issue on "Experiences with Auto-adaptive and Reconfigurable Systems".
4. Cecilia Framework. http ://fractal.ow2.org/cecilia-site/current/index.html.
5. Jean-Philippe Fassino. Nomadik Multiprocessing Framework, a Component-based Programming Model for MP-SoC. 7th International Forum on Application-Specific Multi-Processor SoC, June 2007. http ://www.mpsoc-forum.org/2007/slides/Fassino.pdf.
6. Jean-Philippe Fassino, Jean-Bernard Stefani, Julia L. Lawall, and Gilles Muller. Think : A Software Framework for Component-based Operating System Kernels. In *ATEC '02 : Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 73–86, Berkeley, CA, USA, 2002. USENIX Association.
7. Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit : A Substrate for Kernel and Language Research. *Proceedings of 16th ACM Symposium on Operating Systems Principles (SOSP)*, 1997.
8. Eran Gabber, Christopher Small, John Bruno, José Brustoloni, and Avi Silberschatz. The Pebble Component-based Operating System. In *ATEC '99 : Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 20–20, Berkeley, CA, USA, 1999. USENIX Association.
9. William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6) :789–828, 1996.
10. LynuxWorks. SpyKer. http ://www.lynuxworks.com/products/spyker/spyker.php3.
11. Bernd Mohr, Allen D. Malony, Sameer Shende, and Felix Wolf. Design and Prototype of a Performance Tool Interface for OpenMP. *Journal of Supercomputing*, 23(1) :105–128, 2002.
12. ObjectWeb Project. CORBA Component Model, v4.0. http ://openccm.objectweb.org/doc/index.html.
13. POSIX Thread Trace Toolkit (PTT). http ://nptltracetool.sourceforge.net/.
14. Sameer Shende and Allen Malony. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2) :287–311, 2006.
15. STMicroelectronics. Dynamic Kernel Tracing with KPTrace. http ://www.stlinux.com/docs/manual/development/advanced_development30.php.
16. STMicroelectronics. *OS21 User Manual*, 2007.
17. SUN Microsystems. Enterprise JavaBeans Technology. http ://java.sun.com/pro- ducts/ejb/.
18. Wayne Wolf. The Future of Multiprocessor Systems-on-Chips. In *DAC '04 : Proceedings of the 41st annual conference on Design automation*, pages 681–685, New York, NY, USA, 2004. ACM.
19. Qiang A. Zhao and John T. Stasko. Visualizing the Execution of Threads-based Parallel Programs. Technical Report GIT-GVU-95-01, Georgia Institute of Technology, 1995.