HAL
open science

Une approche à base de composants logiciels pour l'observation de systèmes embarqués

Carlos Hernan Prada Rojas

To cite this version:

Carlos Hernan Prada Rojas. Une approche à base de composants logiciels pour l'observation de systèmes embarqués. Autre [cs.OH]. Université de Grenoble, 2011. Français. NNT : 2011GRENM029 . tel-00621143

HAL Id: tel-00621143

https://theses.hal.science/tel-00621143v1

Submitted on 9 Sep 2011

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

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

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Carlos Hernán Prada Rojas

Thèse dirigée par **Jean-François Méhaut**
et codirigée par **Vania Marangozova-Martin et Miguel Santana**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
et de **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Une approche à base de composants logiciels pour l'observation de systèmes embarqués

Thèse soutenue publiquement le **XX juin 2011**,
devant le jury composé de :

Mr, XX
XX, Président
Mr, Pierre Boulet
Professeur à l'Université Lille 1, Rapporteur
Mr, Christian Perez
Chercheur à l'INRIA Rhône-Alpes, Rapporteur
Mr, XX
XX, Examinateur
Mr, XX
XX, Examinateur
Mr, Jean-François Méhaut
Professeur à l'Université Joseph Fourier, Directeur de thèse
Mme, Vania Marangozova-Martin
Maître de conférences à l'Université Joseph Fourier, Co-Directrice de thèse
Mr, Miguel Santana
Directeur du centre IDTEC à STMicroelectronics, Co-Directeur de thèse

Résumé

À l'heure actuelle, les dispositifs embarqués regroupent une grande variété d'applications, ayant des fonctionnalités complexes et demandant une puissance de calcul de plus en plus importante. Ils évoluent actuellement de systèmes multiprocesseur sur puce vers des architectures *many-core* et posent de nouveaux défis au développement de logiciel embarqué. En effet, Il a classiquement été guidé par les performances et donc par les besoins spécifiques des plates-formes. Or, cette approche s'avère trop couteuse avec les nouvelles architectures matérielles et leurs évolutions rapprochées. Actuellement, il n'y a pas un consensus sur les environnements à utiliser pour programmer les nouvelles architectures embarquées.

Afin de permettre une programmation plus rapide du logiciel embarqué, la chaîne de développement a besoin d'outils pour la mise au point des applications. Cette mise au point s'appuie sur des techniques d'observation, qui consistent à recueillir des informations sur le comportement du système embarqué pendant l'exécution. Les techniques d'observation actuelles ne supportent qu'un nombre limité de processeurs et sont fortement dépendantes des caractéristiques matérielles.

Dans cette thèse, nous proposons EMBera : une approche à base de composants pour l'observation de systèmes multiprocesseurs sur puce. EMBera vise la généricité, la portabilité, l'observation d'un grand nombre d'éléments, ainsi que le contrôle de l'intrusion. La généricité est obtenue par l'encapsulation de fonctionnalités spécifiques et l'exportation d'interfaces génériques d'observation. La portabilité est possible grâce à des composants qui, d'une part, ciblent des traitements communs aux MPSoCs, et d'autre part, permettent d'être adaptés aux spécificités des plates-formes. Le passage à l'échelle est réussi en permettant une observation partielle d'un système en se concentrant uniquement sur les éléments d'intérêt : les modules applicatifs, les composants matériels ou les différents niveaux de la pile logicielle. Le contrôle de l'intrusion est facilité par la possibilité de configurer le type et le niveau de détail des mécanismes de collecte de données. L'approche est validée par le biais de différentes études de cas qui utilisent plusieurs configurations matérielles et logicielles. Nous montrons que cette approche offre une vraie valeur ajoutée dans le support du développement de logiciels embarqués.

Mots-clés : Observation, composant logiciels, systèmes sur puce

Abstract

Embedded software development faces new challenges as embedded devices evolve from Multiprocessor Systems on Chip (*MPSoC*) with heterogeneous CPU towards *many-core* architectures. The classical approach of optimizing embedded software in a platform-specific way is no longer applicable as it is too costly. Moreover, there is no consensus on the programming environments to be used for the new and rapidly changing embedded architectures.

MPSoC software development needs debugging tools. These tools are based on observation techniques whose role is to gather information about the embedded system execution. Current techniques support only a limited number of processors and are highly dependent on hardware characteristics.

In this thesis, we propose EMBera, a component-based approach to MPSoC observation. EMBera aims at providing genericity, portability, scalability and intrusion control. Genericity is obtained by encapsulating specific embedded features and exporting generic observation interfaces. Portability is achieved through components targeting common treatments for MPSoCs but allowing specialization. Scalability is achieved by observing only the elements of interest from the system, namely application modules, hardware components or the different levels of the software stack. Intrusion control is facilitated by the possibility to configure the type and the level of detail of data collection mechanisms. The EMBera approach is validated by different case studies using different hardware and software configurations. We show that our approach provides a real added value in supporting the embedded software development.

Keywords : Observation, software components, embedded systems

Table des matières

1 Introduction **11**

I État de l'art **15**

2 L'observation des systèmes informatiques **17**

2.1 Pourquoi observer ? 17

2.2 Comment observer ? 17

2.2.1 La sélection d'entités à observer 18

2.2.2 La préparation de la cible à observer 18

2.2.3 La collecte de données brutes 19

2.2.4 Le pré-traitement des données générées 20

2.2.5 Le formattage 20

2.2.6 Le stockage 21

2.2.7 L'analyse post-mortem 21

2.2.8 La visualisation des données observées 22

2.3 Synthèse des étapes de l'observation 22

2.4 Critères d'étude des approches d'observation 23

2.5 Classification des approches pour l'observation 24

2.5.1 Systèmes embarqués 24

2.5.2 Systèmes parallèles 29

2.5.3 Systèmes distribués 35

2.6 Synthèse 38

3 Les composants logiciels **41**

3.1 Principes des composants logiciels 41
3.1.1 La définition 42
3.1.2 L'implémentation 43
3.1.3 Le déploiement 43
3.1.4 L'exécution 43
3.2 Critères d'évaluation des projets portant sur les composants 43
3.3 Classification des travaux autour des composants logiciels 44
3.3.1 Modèles génériques 45
3.3.2 Utilisation des composants dans les systèmes embarqués 48
3.3.3 Observation à base de composants 53
3.4 Synthèse 58

II Contribution 59

4 Proposition 61
4.1 Synthèse des travaux étudiés 61
4.2 Objectifs de l'approche 63
4.3 Proposition 64

5 EMBera : un modèle à base de composants logiciels pour l'observation de systèmes embarqués 67
5.1 Le modèle à composants EMBera 67
5.2 Le modèle d'observation EMBera 68
5.2.1 L'encapsulation d'entités 69
5.2.2 Les interfaces spécialisées 69
5.2.3 Les composants basiques 71
5.2.4 Le composants de traitement des données 72
5.2.5 Composant de stockage 74
5.3 Implémentation du modèle EMBera 75
5.3.1 Mise en œuvre du modèle à composants 75
5.3.2 Mise en œuvre du modèle d'observation 77

5.4 Comment instancier le modèle d'observation EMBera ? 79
5.5 Synthèse 82

III Expérimentation 85

6 Étude de cas : décodeur MJPEG 87
6.1 Architecture de la plate-forme embarquée 87
6.2 Mise en œuvre du modèle EMBera 90
6.2.1 Implémentation des composants EMBera 90
6.2.2 Implémentation de l'application MJPEG 91
6.2.3 Déploiement de l'application MJPEG 92
6.3 Application du modèle d'observation EMBera 93
6.3.1 Définition des objectifs d'observation et identification des entités . 93
6.3.2 Identification des composants EMBera 94
6.3.3 Préparation du système pour l'observation 94
6.3.4 Spécialisation du modèle pour la plate-forme 94
6.3.5 Déploiement des composants EMBera 96
6.4 Observations effectuées et problèmes détectés 96
6.4.1 Observations au niveau système 97
6.4.2 Observations au niveau intergiciel 99
6.4.3 Observations au niveau application 99
6.5 Synthèse 100

7 Étude de cas : application industrielle multimédia 103
7.1 Architecture de la plate-forme embarquée 103
7.2 Mise en œuvre du modèle EMBera 105
7.3 Application du modèle d'observation EMBera 106
7.3.1 Définition des objectifs d'observation et identification des entités . 107
7.3.2 Identification des composants EMBera 107
7.3.3 Préparation du système pour l'observation 110
7.3.4 Spécialisation du modèle d'observation pour la plate-forme 110
7.3.5 Instanciation et déploiement des composants EMBera 111

7.4 Observations effectuées et problèmes détectés 112
7.4.1 Observation au niveau de l'intergiciel 112
7.4.2 Observation au niveau du système 113
7.4.3 Corrélation entre les observations 114
7.5 Synthèse 116

8 Étude de cas : décodeur SVC 117
8.1 Architecture de la plate-forme multiprocesseurs 118
8.2 Mise en œuvre du modèle EMBera 123
8.3 Application du modèle d'observation EMBera 126
8.3.1 Définition des objectifs d'observation et identification des entités . 126
8.3.2 Identification des composants EMBera 127
8.3.3 Préparation du système pour l'observation 128
8.3.4 Spécialisation du modèle d'observation pour la plate-forme 130
8.3.5 Instanciation et déploiement des composants EMBera 131
8.4 Observations effectuées et problèmes détectés 132
8.4.1 Observations au niveau de l'application 133
8.4.2 Observations au niveau de l'intergiciel 134
8.4.3 Observations au niveau du système 137
8.4.4 Vue d'ensemble 138
8.5 Synthèse 138

9 Évaluation générale des résultats 141
9.1 Critères d'évaluation 141
9.1.1 Valeur ajoutée des observations 141
9.1.2 Simplicité d'intégration 141
9.1.3 Intrusivité 142
9.2 Évaluation des études de cas 142
9.2.1 Valeur ajoutée des observations 143
9.2.2 Simplicité d'intégration 147
9.2.3 Intrusivité 148

10 Conclusion et perspectives 155

Chapitre 1

Introduction

Dans notre vie quotidienne, nous utilisons de plus en plus de dispositifs électroniques pour le divertissement, la navigation GPS et la communication. Ces appareils, appelés des *dispositifs embarqués*, regroupent une grande variété d'applications, ayant des fonctionnalités complexes, qui demandent une puissance de calcul de plus en plus importante. Cette puissance de calcul est aujourd'hui disponible avec des systèmes multiprocesseur sur puce ou MPSoC [1]. Ces systèmes sont basés sur des processeurs hétérogènes et évoluent vers des architectures *many-core* (beaucoup de cœurs homogènes).

L'approche traditionnelle pour la construction de logiciel embarqué consiste à développer des solutions spécifiques pour chaque génération de plates-formes matérielles. Cette approche produit du logiciel performant, mais avec un coût en développement trop élevé dans un contexte où les architectures matérielles évoluent très rapidement. La question qui se pose actuellement est : « quels sont les modèles et les environnements à retenir comme standard pour le développement des logiciels sur MPSoC ? ». En réponse à cette question, des modèles de programmation à base de composants logiciels sont en train d'être explorés. Toutefois, aucun des modèles et environnements ne s'est encore imposé à ce jour. D'autre part, les schémas algorithmiques pour le développement du logiciel parallèle (*parallel patterns*), définis dans le contexte du calcul de haute performance, sont encore à valider dans le contexte de MPSoC et des architectures *many-core*.

Le logiciel pour une plate-forme MPSoC est développé sur une machine externe, dite de *développement*, où une chaîne de développement (compilateurs, éditeurs de liens) est installée. Sur cette machine, le code est produit puis compilé et téléchargé sur la plate-forme embarquée pour son exécution. Pour la phase de mise au point, des mécanismes d'observation sont déployés sur la plate-forme embarquée, et contrôlés par la machine de développement. L'observation est utilisée, à l'heure actuelle, pour supporter la mise au point des MPSoC ayant un nombre limité de processeurs. Le défi actuel est d'étendre l'observation afin de supporter la mise au point de plates-formes disposant d'un nombre beaucoup plus important de cœurs.

L'observation des MPSoC consiste à recueillir des informations sur le comportement

1. Acronyme anglais : *Multiprocessor System-on-Chip*

du matériel et du logiciel embarqués pendant leur exécution. Nous constatons que cette collecte d'information doit tenir compte des spécificités des plates-formes, ainsi que de l'intrusion ajoutée. Concernant les spécificités, la mise en œuvre de l'observation est fortement dépendante des caractéristiques de la plate-forme sous-jacente. En conséquence, aujourd'hui, les mécanismes d'observation sont spécifiques aux plates-formes et ont une extensibilité limitée en ce qui concerne les nouveaux systèmes sur puce, ce qui rend difficile la portabilité des mécanismes. Par rapport à l'intrusion, si nous souhaitons obtenir un comportement réaliste du système, il faut s'assurer que les mécanismes liés à l'observation ne perturbent pas son fonctionnement normal, et donc minimiser l'intrusion.

Dans cette thèse, nous proposons de nouveaux mécanismes d'observation qui essaient de répondre aux défis posés par la mise au point des logiciels sur les nouvelles architectures MPSoC. Les défis principaux pour l'observation sont les suivants :

- *Généricité* : il s'agit de pouvoir effectuer l'observation toujours dans les mêmes termes, par exemple, à l'aide d'un langage ou au travers d'une interface unique. Cela doit être possible malgré le fait que l'observation soit appliquée sur des éléments différents, qui fournissent des données hétérogènes.
- *Portabilité* : nous pensons qu'une solution pour l'observation doit permettre son adaptation sur différentes plates-formes disposant des configurations matérielles et logicielles spécifiques et différentes.
- *Passage à l'échelle* : nous sommes convaincus de l'importance de gérer un grand nombre d'éléments observables et d'établir des relations entre eux. Nous pensons que le passage à l'échelle peut s'aborder en faisant de l'observation partielle, ce qui veut dire, en tenant compte que d'une sous-partie des éléments du système observé.

- *Intrusion* : nous considérons qu'afin d'avoir une observation fidèle du logiciel sous étude, la solution d'observation doit permettre d'établir un compromis entre la précision des observations et la perturbation admissible du système. En effet, plus l'observation est précise (par exemple en augmentant la périodicité de collecte d'information), plus la perturbation est importante. L'intrusion doit être abordée aussi en réduisant au minimum et en optimisant l'instrumentation pour la production d'information sur le système sur puce.

Le travail de cette thèse a été développé au sein du centre d'expertise IDTEC de STMicroelectronics, qui s'intéresse à la conception et à la mise en place d'outils de débogage et d'observation pour la mise au point des applications embarquées. Les orientations actuelles sont de fournir des outils génériques, qui ciblent des plates-formes matérielles ayant un nombre important de cœurs et du logiciel effectuant des exécutions parallèles complexes.

Cette thèse a été encadrée scientifiquement par l'équipe LIG/INRIA-MESCAL dont le domaine de recherche est la conception de solutions logicielles pour l'exploitation des architectures parallèles et distribuées. L'un des ses domaines d'expertise est l'évaluation de la performance sur des systèmes massivement parallèles à travers des méthodes et des

outils d'observation et analyse, comme Pajé [dKdOS00]. À l'heure actuelle, les technologies du calcul de haute performance et de l'informatique embarquée convergent vers l'utilisation des architectures *many-core*. C'est pourquoi l'équipe LIG/INRIA-MESCAL s'intéresse à la conception, l'adaptation et l'application de ses méthodes de mise au point sur les nouveaux systèmes sur puce.

Organisation du manuscrit

Le document est organisé en trois parties :

1. ***État de l'art*** : Cette partie comporte deux chapitres, l'un sur l'étude des étapes et des approches pour l'observation des systèmes informatiques (chapitre 2), et l'autre sur les principes de base de l'approche à composants logiciels (chapitre 3).

2. ***Contribution*** : nous introduisons ici les conclusions de l'état de l'art et les objectifs de notre proposition (chapitre 4) ainsi que la présentation de notre modèle, appelé **EMBera**, proposé pour l'observation des systèmes sur puce (chapitre 5).

3. ***Validation expérimentale*** : Cette partie présente, sur trois études de cas, l'utilisation d'EMBera sur différentes configurations matérielles et logicielles. Ces études vont permettre d'observer une application de décodage MJPEG (chapitre 6), une application multimédia industrielle (chapitre 7) et un décodeur du format SVC (chapitre 8). Enfin, nous présentons une évaluation de l'utilisation d'EMBera sur ces études de cas (chapitre 9).

Première partie

État de l'art

Chapitre 2

L'observation des systèmes informatiques

Ce chapitre est consacré à l'état de l'art sur l'observation des systèmes informatiques. Tout d'abord nous définissons ce qu'est le processus d'observation et identifions ses différentes étapes. Nous étudions les travaux sur l'observation en nous concentrant sur trois domaines : les systèmes embarqués, les systèmes parallèles et les systèmes distribués.

2.1 Pourquoi observer ?

Pendant le cycle de développement et de mise au point des systèmes matériels et logiciels, l'observation joue un rôle déterminant. En effet, elle permet de comprendre et d'identifier les problèmes de comportement des systèmes. Les problèmes peuvent être liés au fonctionnement (erreur de calcul, erreur de synchronisation, etc.) ou à l'évaluation des performances du système (fuite de mémoire, processeurs sous-utilisés, etc.). L'identification de ces problèmes rend le processus de développement plus efficace et permet le dimensionnement des systèmes. Par exemple, l'utilisation augmentée de mémoire cache dans un système embarqué peut conduire à une consommation énergétique plus faible [TVK08].

Dans cette thèse, nous adhérons à la définition suivante :

Définition 2.1 *Observation : activité initiale dans la mise au point d'un système dont l'objectif est de comprendre le fonctionnement de ses éléments constitutifs, afin de déceler des problèmes potentiels et donner des informations nécessaires pour l'amélioration de la performance.*

2.2 Comment observer ?

L'observation est composée d'une suite d'étapes allant du choix des éléments à observer à l'analyse d'informations sur l'exécution (cf. figure 2.1). Dans la suite nous présentons

plus en détails ces différentes étapes.

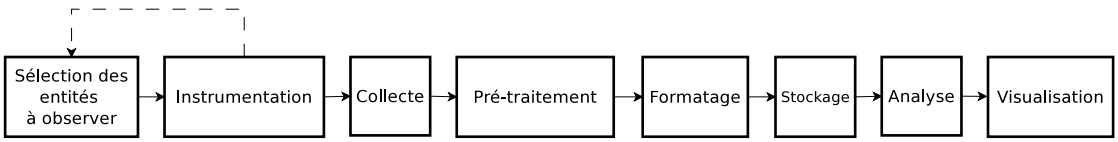

FIGURE 2.1 – Étapes suivies pour l'observation d'un système informatique

2.2.1 La sélection d'entités à observer

La complexité des systèmes informatiques fait que, de manière générale, il est impossible d'observer tout ce qui se produit pendant l'exécution. Pour cette raison, il faut décider et choisir sur quels éléments portera l'observation. Il peut s'agir d'éléments individuels ou bien d'ensembles d'éléments. Dans une application parallèle, par exemple, il est possible d'observer tous les processus individuellement ou alors s'intéresser à des tâches réalisées par plusieurs processus. Nous appellerons les éléments observés des *entités*.

Définition 2.2 *Entité observée : élément d'intérêt du système sous observation ayant des caractéristiques permettant de l'isoler, afin de l'étudier séparément.*

Pendant l'exécution du système sous observation, les entités observées peuvent participer à de nombreuses actions. Une autre question qui se pose donc est de décider si l'observation doit porter sur tous les *évènements* qui concernent ces entités ou sur une sélection. Par exemple, faut-il suivre l'évolution de tous les paramètres des processus ou alors se concentrer uniquement sur quelques caractéristiques (mémoire réservée, placement sur processeur, etc.) ?.

De manière plus formelle, pendant le processus d'observation nous nous intéresserons à des évènements, un évènement étant défini de la manière suivante :

Définition 2.3 *Évènement : changement quantitatif de l'état d'une entité, significatif soit du domaine d'application, soit de l'architecture du système. [LV95].*

2.2.2 La préparation de la cible à observer

Nous appelons *cible* le système qui doit être observé. Afin d'obtenir des informations sur les évènements qui nous intéressent, la cible doit être modifiée afin de pouvoir fournir des données sur ces évènements. Nous disons que nous *instrumentons* la cible afin de pouvoir *tracer* des évènements. Malheureusement, l'instrumentation modifie le comportement du système et nous fournit une vision qui peut être erronée. Nous parlons d'*effet de sonde* et *d'intrusivité* [Mai96].

L'instrumentation peut être faite au niveau matériel ou logiciel. L'instrumentation matérielle utilise des dispositifs matériels spécialisés, fournit des informations matérielles

très précises et n'introduit pas beaucoup d'intrusivité. Cependant, le coût de production et d'intégration de tels dispositifs est élevé et les données produites sont nombreuses et difficiles à interpréter.

Inversement, l'instrumentation logicielle, donne des résultats moins précis [CMdA+08] [SLJD08] et introduit un effet de sonde plus important. Cependant, elle fournit des observations plus accessibles puisque centrées sur les différents niveaux logiques et les entités du logiciel.

L'instrumentation logicielle consiste à rajouter des instructions de traçage dans le code source [Jai91a] ou le code binaire [LCM+05] [NS07] [Inc10]. La modification du code source est généralement plus facile à mettre en place et peut être faite à la main ou automatiquement. Toutefois, elle demande l'accès aux sources logicielles et leur recompilation ce qui peut affecter les transformations et les optimisations effectuées par le compilateur.

L'instrumentation du code binaire consiste principalement en l'interception des appels de fonctions entre le logiciel sous observation et des librairies externes. Son principal avantage est de fonctionner sans la recompilation du logiciel. L'inconvénient est la faible quantité d'évènements que cette technique permet d'observer, car limitée par les informations fournies par le compilateur.

2.2.3 La collecte de données brutes

Une fois instrumenté, le système est prêt à produire des informations sur les entités et les évènements choisis. La production d'informations est appelée *collecte des données brutes* et peut s'effectuer soit à chaque occurrence d'un évènement, soit de manière périodique. Dans le premier cas, nous utilisons *la collecte instantané d'évènements*. Dans le deuxième, nous choisissons *la collecte par échantillonnage*.

La technique utilisée, la périodicité et l'instant où la collecte est effectuée déterminent des stratégies d'observation. Pour l'observation des systèmes logiciels, nous pouvons en citer principalement trois, le *traçage*, le *profilage*[1] et la *surveillance*[2].

- *Le traçage* consiste à recueillir des informations sur des évènements au moment de leur occurrence. Le but est d'avoir un *historique* de l'exécution du logiciel i.e la liste des évènements qui se sont produits en vue d'une analyse postérieure [Lil00]. Les évènements tracés sont enregistrés dans une trace qui peut être organisée selon différents formats. Dans la plupart des cas, les entrées dans la trace comportent une estampille, un identifiant du type d'évènement et des valeurs. Des exemples de solutions génériques pour le traçage logiciel sont les outils `strace` [Kra08] et `TAU` [SM06]. `strace` intercepte et enregistre les appels système et les signaux concernant un programme s'exécutant sur un système Linux. `TAU` est ensemble d'outils pour le traçage d'applications parallèles.

1. en anglais *profiling*
2. en anglais *monitoring*

- *Le profilage* fournit des informations statistiques sur l'exécution d'un logiciel [Lil00]. Nous pouvons citer `gprof` [GKM04] et OProfile [Lev04] qui sont largement utilisés pour le profilage du système Linux.
- *La surveillance* consiste à échantillonner des évènements et à fournir des comptes-rendus de manière périodique. L'avantage principal est le faible coût, car la collecte est effectuée à des intervalles longues mais adaptées à un observateur humain (en d'autres termes, des échantillons de l'ordre de $10^{-1}s$). L'inconvénient le plus important est l'absence de précision. Un exemple typique est l'outil `top` qui surveille les processus du système.

2.2.4 Le pré-traitement des données générées

Selon le type et le nombre d'évènements à observer, la quantité de données brutes produites peut devenir difficile à gérer. Afin de réduire cette quantité et d'obtenir des informations plus condensées, nous pouvons appliquer des fonctions pour traiter les données avant les stocker.

En règle générale, nous pouvons filtrer ou agréger les données [Jai91b]. Le filtrage consiste à évaluer une condition sur les données brutes et à enregistrer seulement les données qui satisfont à la condition. L'agrégation est une fonction qui prend en entrée un ensemble de données et produit une seule donnée comme résultat.

Nous pouvons illustrer l'agrégation à l'aide de l'exemple suivant. Imaginons que nous ayons instrumenté une application multi-threadée et voulons connaître le temps que les threads passent à attendre à cause de la synchronisation. Afin de connaître ce temps, il faudra mesurer toutes les attentes de tous les threads, faire la somme et éventuellement comparer au temps total d'exécution.

2.2.5 Le formattage

Afin de pouvoir tracer différents types d'évènements, ainsi que de pouvoir utiliser ces traces avec des outils d'analyse et de visualisation, il est nécessaire de définir un *format* de trace. Un format est une description des types d'évènements spécifiant leur structure (nombre de champs et type de données). Un des objectifs est de séparer la forme du contenu des traces. Pour cette raison, les outils courants de génération d'historiques créent des fichiers séparés pour le format et pour les données.

Le but n'est pas d'imposer un format au système, mais de normaliser le format d'entrée des outils d'observation de manière à rendre ceux-ci indépendants du système et potentiellement réutilisables pour d'autres systèmes. Dans le traçage d'applications parallèles des formats de trace largement utilisés, comme Open Trace Format (OTF) [KBB+06], Pajé [dKdOS00] et Epilog [WM04], permettent aux données d'être portables entre différentes plates-formes et outils.

2.2.6 Le stockage

Les traces d'exécution sont stockées pour une utilisation future. Les mécanismes de stockage le plus souvent utilisés sont les fichiers de traces et les bases de données. Les premiers sont typiquement composés d'une définition du format, suivie par une suite d'enregistrements décrivant les évènements observés. La définition du format et les valeurs sont souvent organisées dans des fichiers indépendants.

Les supports de stockage basés sur des bases de données utilisent le modèle relationnel pour définir le format des traces. Les bases de données apportent des possibilités intéressantes d'organisation des données et de recherche d'information dans des traces. Néanmoins, elles nécessitent une infrastructure importante pour fonctionner, une structure qui n'est pas toujours accessible par les systèmes sous observation.

Parmi les différentes étapes de l'observation, le stockage de données est l'une des plus coûteuses en termes de temps d'exécution [WFM+06]. La cause principale est le temps d'accès aux supports persistants. Il est dû d'une part, au grand nombre d'interactions avec le système d'exploitation (opérations d'entrée et de sortie) et d'autre part, à la nature des dispositifs (p. ex. magnétiques, accessibles à travers un réseau).

Le coût de stockage des traces peut détériorer de manière considérable les performances des systèmes avec des ressources limitées, comme les plates-formes embarquées. De manière à réduire les temps de stockage dans ces dernières, une solution typique consiste à utiliser des *ports de trace matériels*. Les traces sont aussi stockées dans des portions de la mémoire embarquée et sont traitées par des FPGAs [3] ou par des processeurs spécialisés. L'alliance industrielle MIPI définit des standards pour les ports de traçage et débogage [VKR+08]. Des exemples de ports sont le System Trace Module [TG06] et le port JTAG [93801].

2.2.7 L'analyse post-mortem

Avant l'analyse effective, il peut être nécessaire de corriger et de préparer les traces. Par exemple, il peut être nécessaire de synchroniser des traces indépendantes afin d'obtenir une trace globale du système [GWWM09]. L'analyse post-mortem est l'étape qui utilise les traces d'exécution afin de comprendre le comportement du système. L'objectif est d'obtenir toute information qui permettrait de valider ou d'améliorer le système : vérifier la conformité de l'exécution par rapport aux spécifications, mesurer la performance, détecter des schémas d'exécution, etc.

Une tâche indépendante de l'analyse des données est l'évaluation de l'intrusivité. En effet, les activités d'observation peuvent induire une dégradation des performances ou un changement du comportement du système [Fag97]. L'évaluation de la dégradation consiste à obtenir les différences entre une exécution non observée et une observée. Les différences peuvent être constatées dans : 1) le temps total ou partiel d'exécution, 2) la modification de l'ordre des évènements logiciels et 3) l'utilisation de ressources supplémentaires pour l'observation (mémoire, unités de calcul, etc.).

3. Acronyme anglais : *Field-Programmable Gate Array*

2.2.8 La visualisation des données observées

La visualisation vise une représentation des données de trace qui puisse aider l'utilisateur à mieux comprendre les phénomènes d'exécution du système. Typiquement elle permettra de représenter les données de manière synthétique ou de mettre en évidence les liens entre différents évènements observés.

Une représentation typique est le diagramme temporel qui montre l'historique d'exécution d'un système ainsi que les liens causaux entre évènements (flèches). Un exemple peut être vu à la figure ci-dessous.

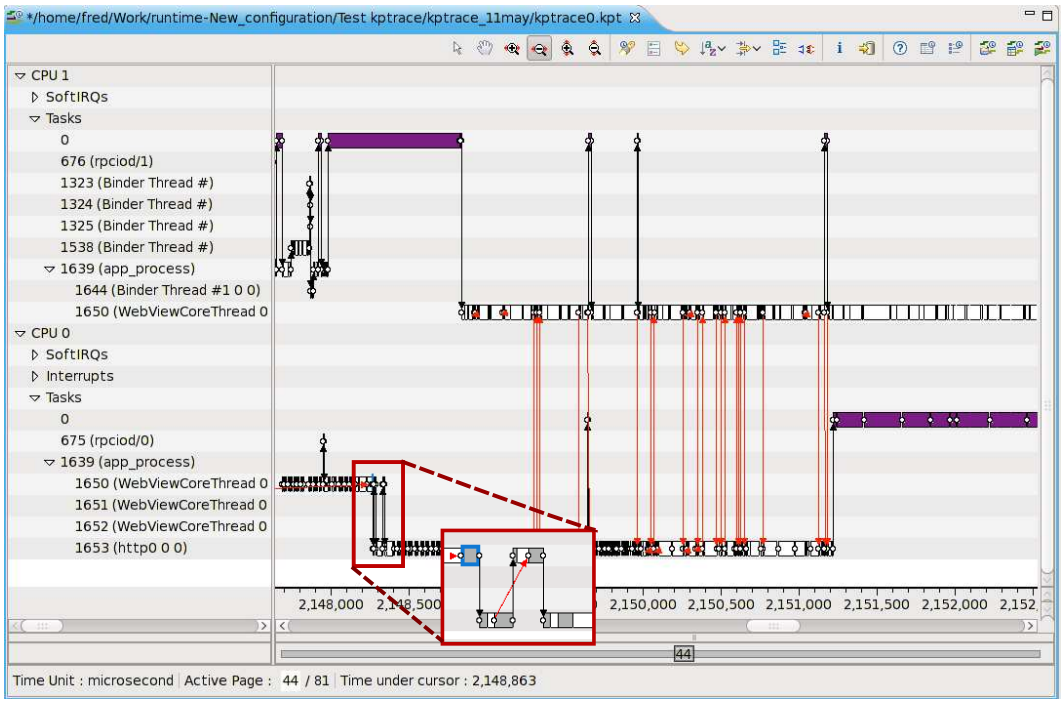

FIGURE 2.2 – Visualisation d'une trace d'exécution d'un système d'exploitation. Les lignes horizontales représentent les historiques d'exécution de différentes entités (cœurs, processus et threads)

2.3 Synthèse des étapes de l'observation

Nous avons présenté les étapes que nous considérons essentielles pour l'observation des systèmes informatiques. En effet, afin d'observer un système, il est nécessaire de : 1) choisir des éléments à observer, 2) instrumenter le système afin de collecter des informations concernant les éléments choisis, 3) collecter des traces pendant l'exécution du système, 4) produire des traces d'exécution formatées, 5) stocker les traces de manière persistante et 6) travailler (analyser, visualiser) avec les traces afin de comprendre le comportement du système. Parmi ces étapes, l'instrumentation et le stockage de données sont les plus intrusifs. Les méthodes utilisées pour ces étapes devront minimiser les modifications sur

la plate-forme et permettre de choisir le niveau de détail des observations, surtout dans le cadre de plates-formes embarquées. D'autre part, afin de rendre les traitements liés à l'observation plus efficaces, les volumes de données/traces doivent être réduites soit pendant les étapes de pré-traitement, soit dans une étape post-mortem.

Dans la suite, nous présentons les approches pour l'observation des systèmes informatiques. Nous présentons d'abord notre liste de critères de classification qui sont fortement liés à l'énumération des étapes d'observation que nous venons d'avoir.

2.4 Critères d'étude des approches d'observation

Pour classifier les travaux sur l'observation étudiés, nous avons choisi les critères suivants :

- *Choix des entités à observer :* Ce critère concerne la possibilité de définir les entités à observer. La question est de savoir si l'approche d'observation fournit un ensemble prédéfini d'éléments ou si la solution est plus ouverte avec possibilité de configuration de l'ensemble d'éléments à observer.

- *Simplicité d'intégration :* Ce critère est lié à l'instrumentation du système. Nous nous intéressons au type d'instrumentation et essayons d'évaluer la simplicité d'intégration. Nous considérons qu'une approche est simple à intégrer si elle introduit peu de modifications au système ou si elle offre une méthode automatisée pour l'instrumentation.

- *Configurabilité :* Est-il possible de régler des paramètres d'observation comme, par exemple, la précision ou la périodicité de collecte de données ?. Le paramétrage se fait-il statiquement, avant exécution, ou peut être fait aussi dynamiquement, pendant l'exécution ?

- *Mécanisme de collecte et de stockage :* Pour chaque travail considéré, nous nous intéressons aux techniques de collecte des données utilisées, ainsi qu'au supports persistants pris en compte. Les techniques identifiées sont traçage, profilage et surveillance. Le stockage peut être fait dans des fichiers, une base de données ou à travers un port de traces.

- *Fonctions de pré-traitement :* Nous analysons si les approches offrent des fonctionnalités pour le traitement des données avant leur enregistrement. L'existence de telles fonctions permet de réduire la quantité des données qui feront partie de l'historique.

- *Intrusivité :* L'intrusion des approches d'observation est le dernier critère considéré. L'intrusion des approches pourrait limiter leur utilisation et même produire des fausses observations. Dans notre classification nous considérons la mesure la plus communément utilisée pour l'intrusion : le temps d'exécution.

2.5 Classification des approches pour l'observation

L'observation des systèmes informatiques est un problème classique traité dans de multiples domaines. Dans cette section nous étudions des propositions dans trois domaines : les systèmes embarqués, les systèmes parallèles et les systèmes distribués.

Nous nous intéressons aux approches pour les systèmes embarqués qui sont le contexte de notre travail. Nous décrivons les travaux d'observation relatifs aux systèmes parallèles, car les systèmes embarqués deviennent *de facto* parallèles. Nous estimons que la prochaine génération de plates-formes embarquées comportera plusieurs grappes indépendantes de calcul parallèle, reliés par des réseaux de communication. À une échelle importante, ces plates-formes peuvent être traités comme des systèmes distribués, ce pourquoi nous étudions les propositions d'observation courantes du domaine.

2.5.1 Observation de systèmes embarqués

Dans cette section, nous présentons deux outils d'observation qui sont dédiés à des plates-formes matérielles spécifiques (Coresight [ARM10] et traçage de processeurs CELL [CRDI05]), ainsi que deux outils d'observation de systèmes d'exploitation pour l'embarqué (KPTrace [STM09a] et LTTng [Des09] [DD09]).

2.5.1.1 Coresight Trace : observation d'architectures ARM

La technologie Coresight, développée par ARM, comporte des fonctionnalités de traçage et de débogage des plates-formes embarquées ayant des processeurs ARM Cortex[4] [ARM10]. Pour le traçage, ces plates-formes sont munies des composants matériels *Embedded Trace Macrocell* (ETM) et *Program Trace Macrocell* (PTM) (cf. figure 2.3). Ces composants fournissent un support pour l'estampillage (même dans des architectures ARM multicœur) et pour le transport des évènements générés par les cœurs de la plate-forme vers un bus dédié au traçage.

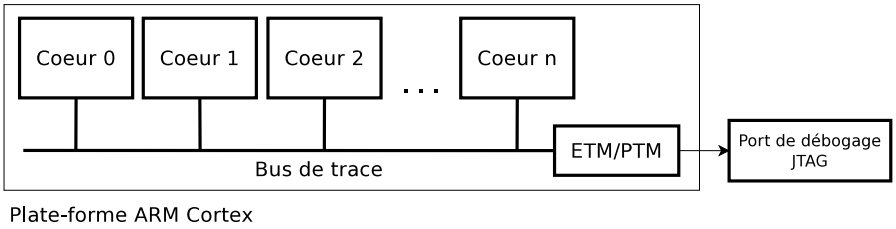

FIGURE 2.3 – Plate-forme multi-cœur ARM avec des composants de traçage ETM/PTM

– *Choix des entités à observer* : Les entités observables sont les cœurs et la mémoire de la plate-forme. Les cœurs sont observables à travers des registres du processeur qui fournissent des informations comme les compteurs d'instructions, la prédiction

4. Processeurs ARM à faible consommation électrique

de branchement, etc. Le choix des entités est effectué en activant l'échantillonnage des registres disponibles.

- *Simplicité d'intégration* : Au niveau matériel, la plate-forme est munie des composants ETM et PTM et n'a donc pas besoin d'instrumentation. Inversement, rien n'est prévu pour l'observation du logiciel et tout (collecte, formattage, traçage...) doit être implémenté à partir de zéro. L'interface de programmation offre seulement un ensemble des fonctions pour l'accès aux composants matériels de traçage.

- *Configurabilité* : L'approche permet de sélectionner les compteurs à observer et de régler la périodicité de l'échantillonnage et la taille du tampon pour l'enregistrement de la trace. Cette configuration peut être modifiée pendant l'observation.

- *Mécanisme de collecte et de stockage* : Pour le matériel, la collecte est effectuée en lisant des registres des processeurs ou en traçant des évènements spécifiques produits par la plate-forme (p. ex. dépassement d'un seuil de température d'un processeur). L'enregistrement de données est supporté par le port de trace fournit par le composant PTM.

- *Fonctions de pré-traitement* : L'approche ne propose pas de fonctionnalités spécifiques pour le filtrage ou l'agrégation des évènements, ni matériels, ni logiciels.

- *Intrusivité* : Les auteurs affirment que lors de la collecte des informations sorties du matériel, leur solution n'introduit aucune perturbation dans l'exécution du logiciel. En ce qui concerne l'observation du logiciel, il n'y a pas de documentation publique disponible là-dessus.

2.5.1.2 Analyse de la performance du processeur CELL basée sur des traces

Le CELL [CRDI05] est un processeur utilisé sur des plates-formes de calcul de haute performance, ainsi que sur des systèmes embarqués, comme les consoles de jeux. Il possède une architecture matérielle multiprocesseur hétérogène, composée d'un processeur maître (PPE) et huit processeurs accélérateurs (SPE).

L'approche d'observation est purement logicielle et consiste à tracer les évènements de chacun des processeurs et à les écrire sur une portion de la mémoire du PPE, accessible par tous les processeurs (cf. figure 2.4). Cette mémoire est lue en permanence par un démon [5] de gestion de traces qui stocke les données sur un support persistant [BST+08].

- *Choix des entités à observer* : L'outil définit trois modes de traçage. Le premier mode trace les débuts et fins d'exécution sur un processeur. Le deuxième mode concerne les accès mémoire. Le troisième mode est un mode complet qui trace les évènements des deux précédents, ainsi que les évènements de synchronisation, les signaux et les évènements définis par l'utilisateur.

5. Une partie d'un programme qui n'est pas appelé explicitement, mais qui se trouve en attente passive de l'occurrence d'une condition donnée [Ray96].

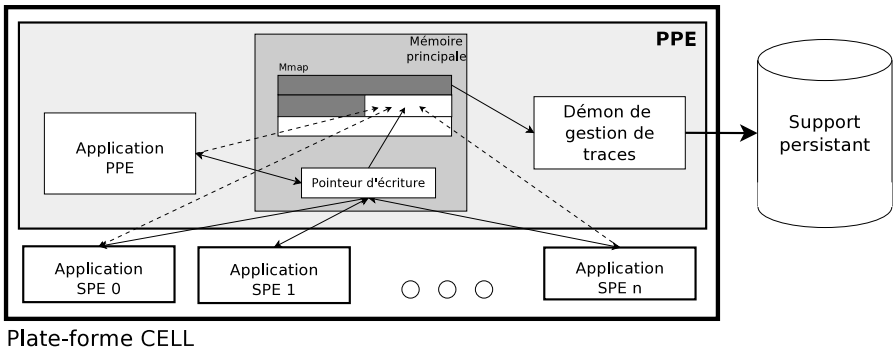

FIGURE 2.4 – Collecte de traces du processeur CELL

- *Simplicité d'intégration* : Pour le code exécuté sur le PPE, l'instrumentation est faite automatiquement lors de la compilation, grâce à des options spécifiques fournies par la chaîne de développement de la plate-forme. Le code des SPEs, cependant, doit être modifié manuellement en ajoutant des appels à des fonctions d'observation. Ces fonctions sont fournies par une boite à outils de développement d'applications pour le processeur CELL [Cor07b].

- *Configurabilité* : L'outil permet de régler le détail de l'information à collecter en sélectionnant un des modes de traçage mentionnés. La version courante de l'outil d'observation ne permet pas de modifier d'autres paramètres, ni avant, ni pendant l'exécution.

- *Mécanisme de collecte et de stockage* : La technique utilisée pour recueillir les données est le traçage. Chaque évènement est enregistré avec une estampille et l'identifiant de son cœur. Les traces sont stockées dans un fichier de trace géré par le module de collecte dans le PPE.

- *Fonctions de pré-traitement* : L'outil ne propose pas des fonctionnalités d'analyse initiale pour réduire le nombre des données à enregistrer. Le seul pré-traitement fourni est la correction des estampilles avant de l'enregistrement des données dans la trace.

- *Intrusivité* : Les auteurs affirment que l'intrusion ajoutée est proche de zéro si le système est instrumenté, mais la collecte de données est désactivée. Si elle est activée, l'intrusion dépend des choix des évènements et du type d'application observée. Par exemple, les auteurs évaluent l'intrusion sur deux versions d'une application de calcul de FFT [6], l'une avec une faible communication entre les cœurs et l'autre avec un taux de communication important. L'intrusivité mesurée a été respectivement de 6% et 25%.

6. A programming example : Large FFT on the CELL Broadband Engine [CFB05]

2.5.1.3 KPTrace : observation des systèmes d'exploitation multi-processus embarqués de STMicroelectronics

KPTrace [STM09a] est un outil d'observation du logiciel embarqué s'exécutant sur les systèmes d'exploitation Symbian [Com10] et les distributions Linux de STMicroelectronics [STM09b]. Cet outil fait partie d'une suite d'outils de débogage, traçage, analyse et visualisation de STMicroelectronics appelé STWorkbench [STM10].

- *Choix des entités à observer* : L'outil permet la sélection des évènements à observer, non des entités. L'outil active, de manière interne, le traçage sur les entités produisant les évènements choisis. Les entités gérées par l'outil sont principalement des unités de calcul, des flots d'exécution et des interruptions. En relation avec ces entités, l'outil peut observer les symboles du noyau (p. ex. le traitant d'interruptions, les appels du système, les changements de contexte, les fonctions et leurs arguments, etc.), les opérations sur la mémoire (allocation et libération), etc.

- *Simplicité d'intégration* : KPTrace nécessite que le système d'exploitation soit instrumenté par le développeur d'applications en ajoutant un module de KPTrace au noyau, puis en recompilant le système.

- *Configurabilité* : L'outil fournit un fichier de configuration où les évènements à tracer sont définis avant l'observation. Il fournit également un mécanisme pour activer/désactiver le traçage pendant l'exécution. Il permet aussi d'établir la taille et le mode de traitement du tampon pour le stockage temporaire.

- *Mécanisme de collecte et de stockage* : Les données sont collectées à l'aide du traçage. Si l'observation est activée pour un évènement, les données produites sont envoyées vers un démon de KPTrace, qui les enregistre dans un tampon dans la mémoire vive afin de réduire les accès au support persistant. Le stockage est effectué soit sur le système de fichiers de la plate-forme (typiquement sur une mémoire flash ou sur NFS [7]), soit sur un port de trace. Les données sont enregistrées dans un format fournit par KPTrace, qui contient le type d'évènement, l'estampille, l'identifiant du processus ou du thread système et une suite variable d'arguments.

- *Fonctions de pré-traitement* : KPTrace permet de filtrer les données observées pendant la collecte. Pour cela, il est possible de définir des seuils sur des valeurs des évènements observés.

- *Intrusivité* : Nous avons constaté, en expérimentant avec l'outil, que l'intrusion sur le système est proche de 0% si la production des données est désactivée. Si elle est activée, l'intrusivité dépend principalement du nombre d'évènements observés, de l'application en exécution et du mécanisme de stockage utilisé. Pour l'observation

7. Acronyme anglais : Network File System

d'un système STLinux qui exécute une application de décodage vidéo, l'intrusion est de l'ordre de 10%. La valeur est obtenue en observant tous les appels système et les interruptions disponibles, et en stockant la trace sur le système de fichiers.

2.5.1.4 LTTng : outil de traçage du système Linux

Il s'agit d'un outil pour le traçage des évènements du noyau Linux et des fonctions utilisateur [Des09] [DD09]. Il permet d'effectuer des observations multi-niveaux, depuis la plate-forme matérielle jusqu'à l'espace utilisateur. LTTng a été récemment porté sur des systèmes Linux embarqués. L'outil définit des fonctions de traçage génériques afin de fournir une interface commune d'observation et pour rester portable entre les différentes plates-formes. Cependant, les fonctions de traçage nécessitent des mécanismes d'estampillage spécifiques à chaque plate-forme pour être plus précises et moins intrusives.

- *Choix des entités à observer* : LTTng permet de tracer principalement des évènements du système d'exploitation (interruptions, changements de contexte, appels aux fonctions du système, etc.). Ils sont appelés *points d'instrumentation*. Les utilisateurs de LTTng peuvent activer les points d'instrumentation de leur choix.

- *Simplicité d'intégration* : LTTng utilise peut utiliser l'un des deux mécanismes d'instrumentation : LKM [Cor07a] et SystemTAP [EH06]. Ces mecanismes demandent la recompilation du noyau pour leur intégration. Pendant l'observation, SystemTAP permet d'activer/désactiver dynamiquement le traçage d'évènements, alors que, LKM demande une recompilation.

- *Mécanisme de collecte et de stockage* : La collecte des données utilise le traçage. LTTng ne propose aucun format pour l'enregistrement des données. Le stockage est fait dans un fichier de trace sur le système de fichiers de la plate-forme. En présence d'une exécution multiprocesseur, l'outil génère une trace par unité de calcul impliquée dans l'exécution.

- *Fonctions de pré-traitement* : LTTng ne propose pas de fonctions de pré-traitement avant l'enregistrement des données.

- *Intrusivité* : L'intrusion dépend du nombre d'évènements produits par l'application et le système, ainsi que de la stratégie d'instrumentation choisie. Si LTTng utilise LKM, le taux d'intrusion est d'environ 3% et celui de SystemTAP est d'environ 10% [DD06].

Synthèse des approches d'observation pour l'embarqué

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 en conséquence sont faciles à intégrer. Cependant, ils offrent

une portabilité très faible et fournissent de l'information de très bas niveau, comme l'état de la mémoire (valeurs des registres, contenu de la mémoire) ou les évènements du système d'exploitation (interruptions, appels système). Souvent, ces outils ne fournissent pas d'informations sur les couches logicielles applicatives ou, s'ils le font, ils n'établissent pas de corrélation entre les évènements des différentes couches. De plus, dans la plupart de cas les entités et les évènements à être observés sont prédéfinis.

En ce qui concerne le stockage, les approches tracent de gros volumes d'évènements. Pour cette raison, ces approches utilisent des mécanismes de gestion de traces, indépendants ou déportés du logiciel observé. Cette indépendance, fournie principalement par de ports de traçage, réduit l'intrusion de l'observation sur la plate-forme embarquée. Les outils qui n'utilisent pas un dispositif matériel implémentent des mécanismes logiciels, comme des démons dont l'objectif est également de gérer l'historique, mais ajoutant une intrusion plus élevée.

2.5.2 Observation de systèmes parallèles

Nous constatons aujourd'hui une convergence entre les systèmes parallèles et embarqués [Wol04]. Pour cette raison nous pensons que les approches d'observation pour les systèmes parallèles donnent des indications intéressantes pour la conception d'approches d'observation pour les systèmes embarqués multi-cœur.

Dans cette section, nous étudions quatre approches d'observation pour les systèmes parallèles. Ces approches sont fortement liées aux modèles de programmation utilisés dans le domaine parallèle, notamment la programmation à base de threads [POS93], OpenMP [DM98] et MPI [SOW+95]. Les travaux que nous considérons sont TAU [SM06], HPCToolkit [ABF+10], Scalasca [GWW+10], et Parallel Profile Snapshots [MSMS08].

2.5.2.1 TAU : ensemble d'outils d'observation et d'analyse

TAU est un ensemble d'outils pour l'observation des systèmes parallèles [SM06] qui fournit des fonctionnalités génériques d'instrumentation. TAU cible des programmes parallèles qui utilisent des environnements de programmation basés sur des processus légers (Threads), des processus distribués (MPI) et des bibliothèques pour l'utilisation de la mémoire partagée (OpenMP). En outre, TAU permet de supporter d'autres environnements de programmation grâce à des fonctionnalités pour les offertes par son interface aux développeurs.

- *Choix des entités à observer* : TAU permet d'observer les compteurs de performance et prédéfinit les éléments observables pour chaque environnement de programmation. Pour des applications MPI, par exemple, TAU rend observables les primitives de communication. L'approche offre également des fonctions génériques pour l'observation d'évènements définis par l'utilisateur.

- *Simplicité d'intégration* : TAU offre plusieurs mécanismes d'instrumentation. Il permet l'instrumentation manuelle du code source en fournissant des fonctions de

trace. Il fournit un pré-processeur qui instrumente le code source automatiquement. Dans un code binaire, il permet d'intercepter les appels entre l'application et une librairie . Enfin, TAU permet de rajouter des instructions, voire des modules au noyau (p. ex. PAPI [BDG+00]) afin d'avoir des observations d'un niveau plus bas (p. ex. les compteurs du processeur).

– *Configurabilité* : Pour les évènements concernés par l'instrumentation du code source, TAU offre une interface de contrôle et de configuration. Typiquement, il permet d'activer/désactiver l'observation des évènements et de configurer la périodicité d'échantillonnage.

– *Mécanisme de collecte et de stockage* : La collecte des données est effectuée en utilisant des techniques de traçage, de profilage et d'échantillonnage. Les données, dans la plupart de cas, sont stockées dans des fichiers. La configuration par défaut de TAU crée un fichier par flot d'exécution parallèle. Quand les fichiers sont crées, ils sont traités en comparant et en corrigeant la base de temps des évènements observés. La figure 2.5 montre la chaîne des traitements des traces : le logiciel est instrumenté afin de produire des évènements ; après, une trace est générée par chaque thread de l'application ; enfin, une seule trace est produite.

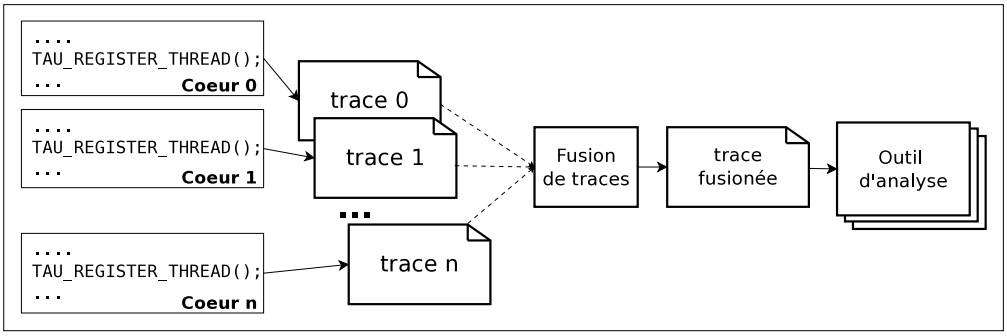

FIGURE 2.5 – Gestion des traces dans TAU

– *Fonctions de pré-traitement* : TAU propose uniquement des fonctions d'analyse post-mortem.

– *Intrusivité* : L'intrusion de TAU dépend, bien évidemment, de l'application observée, de la configuration de TAU et de l'ensemble des évènements à observer. Pour l'exécution d'une application du calcul de la décomposition de LU [8], sur une grappe de calcul Linux à 8 nœuds, l'observation effectuée avec TAU ajoute les intrusions suivantes : profilage, environ 1% ; traçage, proche de 5% [Eva05].

8. Application appartenant au benchmark NAS [VdWW02].

2.5.2.2 HPCToolkit : outils de mesure de programmes parallèles à l'aide d'échantillons

HPCToolkit est un ensemble d'outils pour la mesure, l'analyse et la visualisation d'applications MPI ou OpenMP [ABF+10]. L'observation est basée sur l'échantillonnage d'informations de profilage. Autrement dit, l'application produit des données de profilage qui sont récupérées par HPCToolkit à un intervalle de temps donné. Les analyses proposées permettent de quantifier l'utilisation des processeurs et d'identifier des problèmes, comme des goulots d'étranglement dans le code optimisé. Trois outils principaux sont fournis (cf. figure 2.6) :

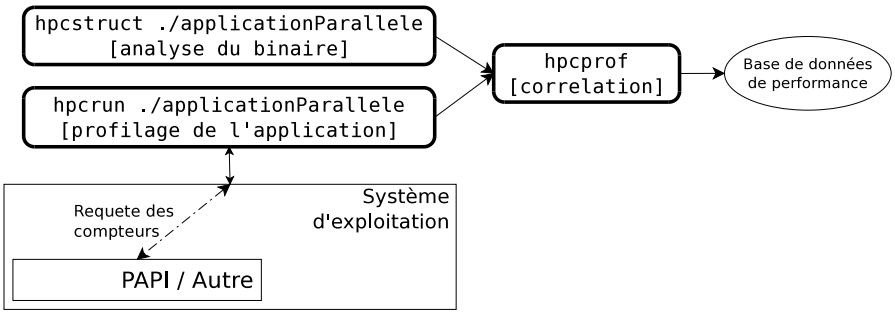

FIGURE 2.6 – Collecte et traitement de données dans HPCToolkit

- `hpcrun` : lance le code binaire instrumenté des applications et récupère les données observées (p. ex. des compteurs de performance),
- `hpcstruct` : analyse le code source et binaire de l'application afin de reconstruire la structure du programme. La structure est reconstruite par l'identification des structures internes comme des procédures et des boucles imbriquées.
- `hpcprofile` : effectue la corrélation entre les deux outils précédents.

- *Choix des entités à observer* : Les éléments observables sont prédéfinis et sont centrés sur les flots d'exécution de l'application. Sont observés, par exemple, les appels de fonctions, l'évolution des valeurs des compteurs de performance de chaque processeur, la migration des flots entre processeurs, etc.

- *Simplicité d'intégration* : Les applications doivent être recompilées avec des options spécifiques. Pour l'accès aux compteurs de performance, nous devons ajouter des bibliothèques externes lors de l'édition de liens ou par recompilation de l'application.

- *Mécanisme de collecte et de stockage* : L'outil `hpcrun` lance l'application et collecte les données de profilage et des compteurs de performance lors de l'exécution de l'application (cf. figure 2.6). Le résultat est stocké dans des fichiers temporaires, un par processeur impliqué dans le calcul. Enfin, le résultat est corrélé avec la structure obtenue par `hpcstruct`, puis la corrélation est enregistrée dans une base de données.

- *Configurabilité* : Il est possible de configurer uniquement la période de prise des échantillons et le seuil que doivent atteindre les valeurs d'un évènement donné pour les observer. Cette configuration est faite à l'aide de l'outil `hpcrun`.

- *Fonctions de pré-traitement* : Les données sont traitées par l'outil `hpcprof` avant le stockage dans la base de données. Le traitement consiste à corréler les valeurs des compteurs de performance, mesurées par `hpcrun`, avec les structures de l'application, identifiées par `hpcstruct`.

- *Intrusivité* : Les auteurs ont utilisé HPCToolkit pour observer le déroulement de l'application PFLOTRAN [9] déployée sur 8 184 cœurs cadencés à 2.6 Mhz chacun et confinés dans une machine Cray XT5. Les observations ont été effectuées en observent des compteurs de performance. La fréquence de collecte de données a été de 230 $échantillons/s$. Le surcoût calculé sur le temps d'exécution total a été de 3% [TAMC10].

2.5.2.3 Scalasca : observation de programmes parallèles à grande échelle

Scalasca est une boîte à outils pour l'observation d'applications MPI et OpenMP déployées sur plusieurs milliers de processeurs [GWW+10]. L'outil collecte des données sur l'exécution de tous les processus ou threads.

- *Choix des entités à observer* : Les entités et les évènements sont établis par l'environnement de programmation. Ils sont principalement des flots d'exécution et des appels aux fonctions parallèles et de synchronisation entre les flots.

- *Simplicité d'intégration* : Scalasca utilise l'instrumentation soit manuelle, soit automatique, des sources de l'application. L'instrumentation automatique est faite avec des options de compilation et l'ajout d'une bibliothèque d'instrumentation lors de l'édition de liens. Elle utilise, par exemple, la bibliothèque POMP [MMSW02] pour l'observation des applications OpenMP.

- *Configurabilité* : Scalasca peut être paramétrée avec des fichiers de configuration. qui spécifient les évènements à collecter ou les analyses à effectuer. Les auteurs ne spécifient pas de possibilités de contrôle pendant la collecte de données. Cependant, l'outil peut changer en cours d'exécution les évènements observés, si la régularité ou le volume de données collectées dépasse un seuil établi.

- *Mécanisme de collecte et de stockage* : Les données sont produites principalement à l'aide du traçage. Un fichier de trace est généré par flot d'exécution parallèle. Le stockage et l'analyse sont effectués de manière itérative. En d'autres termes, quand une analyse est effectuée, les résultats sont enregistrés pour l'étape d'analyse suivante.

9. Application MPI pour la modélisation multi-phase et multi-composant des flux du sous-sol [ea09]

– *Fonctions de pré-traitement* : Scalasca offre des fonctions de correction d'estampilles, d'analyse et de filtrage de données observées, avant leur enregistrement. L'analyse permet d'identifier les éléments le plus souvent trouvés dans les observations, qui sont considérées comme des informations triviales. Puis le filtre exclut les éléments identifiés afin d'alléger l'historique pour l'analyse post-mortem.

– *Intrusivité* : Scalasca fournit un outil pour mesurer le temps passé à tracer les évènements. Les auteurs de Scalasca affirment que le traçage des fonctions le plus souvent observées est la cause principale de l'intrusivité. Pour cela ils recommandent de désactiver leur observation. Néanmoins, dans les références étudiées, les auteurs ne donnent jamais une évaluation quantitative de l'intrusivité.

2.5.2.4 Parallel Profile Snapshots : vues de performance pour des programmes concurrents

Parallel Profile Snapshots fournit des bilans d'exécution pour des applications MPI et OpenMP. Ces bilans, appelés *vues de performance*, sont basés sur des données de profilage [MSMS08]. La particularité est que les données de profilage sont estampillés dans chacune des vues, puis enregistrées. La figure 2.7 montre une vue générée à un instant t_2, qui contient le profil des fonctions d'une application MPI.

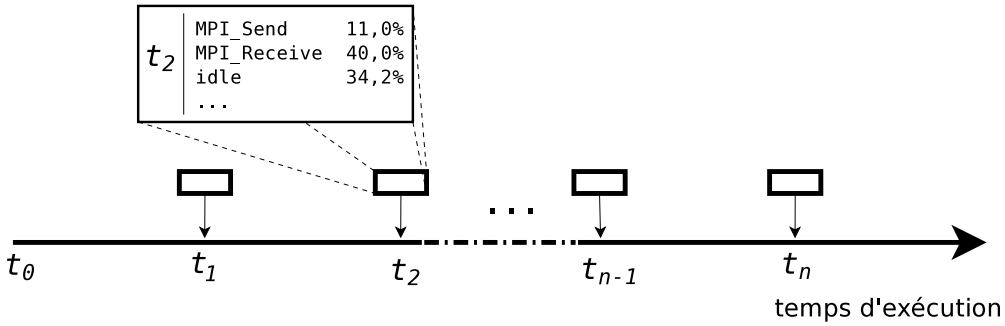

FIGURE 2.7 – Vues de performance basées sur de profilage

– *Choix des entités à observer* : L'outil prédéfinit un ensemble d'entités et d'évènements observables, qui peuvent être activés ou désactivés. Il s'agit, typiquement, des flots d'exécution, des opérations de communication, de l'utilisation de la mémoire ou des processeurs.

– *Simplicité d'intégration* : L'instrumentation utilise des techniques proposées par TAU : la modification manuelle ou automatique du code source avec des fonctions de traçage et de profilage. Pour l'observation de l'utilisation de ressources, l'outil se sert des interfaces fournies par le même système.

– *Configurabilité* : L'outil permet d'être configuré avant l'observation en choisissant les évènements à observer, ainsi que, la régularité ou la condition nécessaire

pour déclencher la génération d'une vue de performance. Le contrôle pendant l'observation n'est pas considéré par la proposition.

- *Mécanisme de collecte et de stockage* : L'application génère des évènements de trace pendant son exécution (un ensemble d'évènements par flot d'exécution), mais à la différence du traçage classique, les données ne sont pas toutes enregistrées. À chaque fois qu'une condition est atteinte, l'outil déclenche la génération d'une vue de performance avec les évènements observés depuis la dernière vue. La génération d'une vue est faite selon une périodicité établie ou déclenchée par un évènement. L'outil synchronise la génération des données de profilage de manière à produire toutes les vues au même temps pour tous les flots d'exécution. Les vues sont enregistrées dans un fichier XML, utilisant un format défini par l'outil. Les fichiers sont stockés localement et, dans une étape postérieure, sont combinés afin de produire un seul fichier de profilage de toute l'exécution.

- *Fonctions de pré-traitement* : Les vues sont générées, tout au long de l'exécution, grâce à l'application des fonctions d'agrégation qui permettent d'obtenir le profil de l'application dans le créneau établi. Par exemple, si nous observons une application MPI, le profil doit montrer le temps passé dans la vue par l'exécution de la fonction `MPI_Send`.

- *Intrusivité* : Les auteurs ont utilisé l'application de simulation FLASH [RCD+00] sur 128 nœuds pour évaluer leur outil. Le taux d'intrusion ajouté est d'environ 8%, comparé avec un taux de 6% pour l'observation de l'application utilisant du profilage traditionnel.

Synthèse des approches d'observation pour les systèmes parallèles

Contrairement aux systèmes embarqués, les outils d'observation pour les systèmes parallèles sont dirigés principalement par les modèles et les environnements de programmation. La montée dans le niveau d'abstraction permet la corrélation des informations de l'environnement parallèle, du système d'exploitation et de l'application.

Les approches d'observation qui ciblent un même environnement de programmation souvent prédéfinissent les mêmes entités observables. Néanmoins, les outils offrent de la souplesse pour la sélection de ces entités. Ils utilisent des mécanismes pour configurer, statiquement ou dynamiquement, la périodicité, la taille des tampons, etc.

D'autre part, les approches d'observation des systèmes parallèles ont mis au point des mécanismes pour la gestion de grands volumes de données, ainsi que pour la synchronisation des observations. Dans certains cas, les approches proposent aussi des fonctionnalités de traitement des traces. Ces traitements cherchent à réduire le volume des traces ou à identifier des séquences intéressantes dans la trace.

Pour la plupart des approches, la minimisation de l'intrusivité n'est pas un objectif. En effet, les approches étudiées donnent des pourcentages d'intrusion de l'observation,

mais seulement à titre indicatif. Cependant, les exécutions parallèles peuvent produire des traces assez volumineuses dont la gestion peut impacter fortement le système.

Nous pensons que les évolutions dans l'embarqué nous amèneront vers des plates-formes sur puce massivement parallèles, ce qui veut dire que nous devons tenir compte des propositions d'observation des systèmes parallèles. Néanmoins, pour les rendre utilisables sur les MPSoC, les approches doivent être capables de supporter les environnements de programmation utilisés dans les systèmes embarqués. Elles doivent également être optimisées, en termes d'empreinte mémoire et d'intrusion ajoutée.

2.5.3 Observation de systèmes distribués

Les systèmes distribués posent des défis pour leur observation, dûs à la gestion d'un grand nombre d'éléments, à la difficulté de calcul d'un état global, à l'absence d'une horloge globale et au coût de la communication entre les éléments du système [Gar97].

Dans les systèmes embarqués de la prochaine génération, nous trouverons des architectures fortement parallèles composées de plusieurs grappes de calcul distribuées [BLM+07]. Les objectifs de ces approches d'observation seront très probablement pris en compte par les nouvelles plates-formes.

Nous présentons deux propositions pour l'observation des systèmes distribués avec des objectifs différents. La première est Ganglia [MCC04], pour la surveillance et le contrôle des grands systèmes de calcul. La deuxième est Black Boxes [AMW+03], pour le débogage d'un système à partir des traces des opérations de communication.

2.5.3.1 Ganglia : surveillance des systèmes de calcul de haute performance

Ganglia est système de surveillance distribuée pour des grappes ou des grilles de calcul [MCC04]. Les éléments de la grappe sont organisés dans des hiérarchies. Une hiérarchie définit des groupes de grappes (fédérations), des grappes (ensembles de nœuds) et des nœuds. Dans chaque grappe, un nœud est désigné pour collecter les données. Par exemple, dans la figure 2.8, nous voyons que le nœud à gauche de chaque grappe est en charge de la collecte des données de la grappe.

- *Choix des entités à observer* : Les entités observées sont celles que nous trouvons dans les hiérarchies Ganglia, notamment les nœuds, les grappes et les fédérations. Les informations collectées concernent les ressources matérielles, typiquement le pourcentage d'utilisation du CPU, l'utilisation de la mémoire, etc.

- *Simplicité d'intégration* : Afin d'observer une machine, il est nécessaire d'installer Ganglia. Les métriques sont obtenues en appelant des fonctions du système d'exploitation et sont rendues disponibles à l'aide d'une hiérarchie de démons. Le démon `gmond` s'exécute au niveau d'un nœud et met les résultats à disposition du niveau suivant dans la hiérarchie. Pour l'observation des métriques des grappes, le démon `gmetad` doit être ajouté au nœud désigné du grappe. La figure 2.8 montre

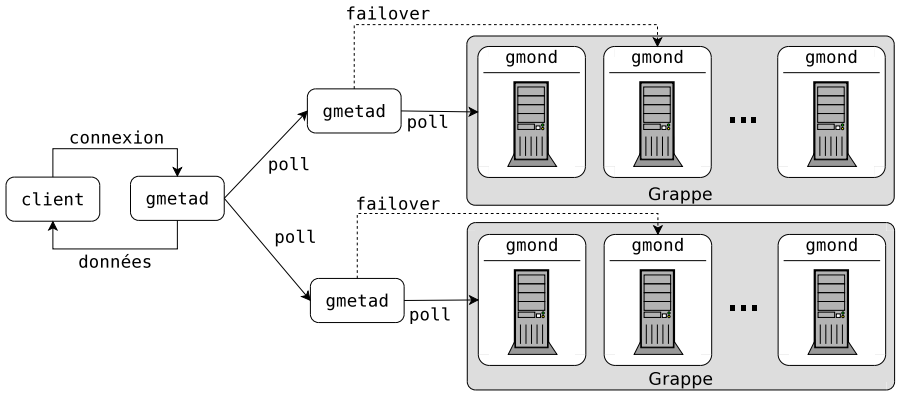

FIGURE 2.8 – Hiérarchie de surveillance dans Ganglia [Kra09]

les hiérarchies entre les démons.

– *Configurabilité* : Ganglia permet d'ajouter des nouvelles grappes ou des nœuds pendant l'observation. Au niveau d'un nœud, il permet soit de choisir sur une liste, soit de définir des nouveaux évènements à observer.

– *Mécanisme de collecte et de stockage* : La collecte de données est faite par échantillonnage. Les données sont produites lors d'une requête reçue depuis un niveau supérieur dans la hiérarchie. Les données sont stockées localement (par le même nœud) et par le nœud qui en a fait la demandé. Les données sont stockées dans des fichiers XML suivant un format défini par Ganglia.

– *Fonctions de pré-traitement* : Ganglia fournit des fonctions de filtrage et d'agrégation afin de faire remonter moins de données dans la hiérarchie. De plus, l'outil permet de générer des bilans d'exécution des nœuds ou des grappes.

– *Intrusivité* : Les développeurs de l'outil ont effectué des expérimentations sur trois plates-formes distribuées : Millenium, SUNNY et PlanetLab, ayant respectivement 94, 2 000 et 84 nœuds (représentant 42 sites, 2 nœuds par site). L'intrusion locale mesurée (exécution de `gmond`) a été de moins de 1% dans le cas général, et d'environ 2% pour les nœuds désignés (exécution supplémentaire de `gmetad`). Le taux total d'intrusion n'a pas été calculé, car ce dernier est négligeable par rapport à la latence des communications dans Internet.

2.5.3.2 Black Boxes : débogage de performance des systèmes distribués

Il s'agit d'un outil qui trace les opérations de communication entre les nœuds du système [AMW+03]. La figure 2.9 montre une suite d'interactions de communication entre quatre composants A, B, C et D. Pour l'enregistrement, les interactions sont organisées selon leur ordre causal (partie inférieure de l'image).

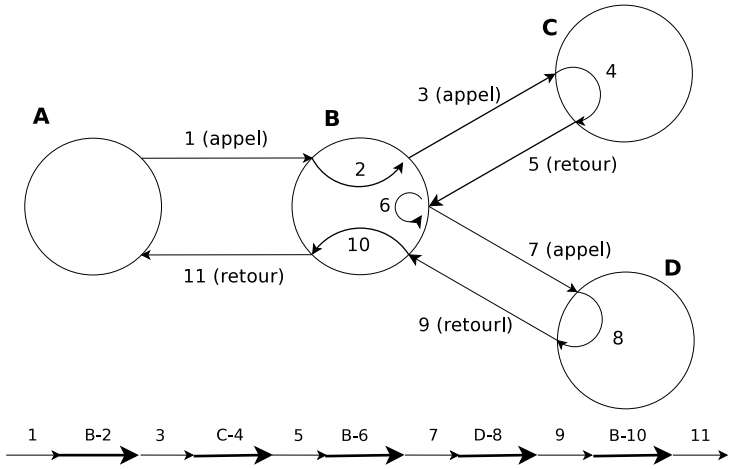

FIGURE 2.9 – Génération de l'historique des interactions dans BlackBoxes [AMW+03]

- *Choix des entités à observer* : Les entités dans cet outil sont les nœuds d'un système distribué. Les évènements sont liés principalement aux messages échangés entre les nœuds.

- *Simplicité d'intégration* : L'intégration est faite en ajoutant au système d'exploitation un programme qui collecte des données, par des mécanismes passifs de traçage du réseau. Les mécanismes sont soit la *réplication de ports* au niveau des commutateurs réseau, soit le *reniflage de paquets*[10] sur chaque nœud.

- *Configurabilité* : L'outil permet de choisir les nœuds à observer et le type de communication utilisé par l'application. De même, il offre une interface pour choisir le mécanisme passif du traçage du réseau à utiliser.

- *Mécanisme de collecte et de stockage* : La collecte de données est faite par traçage. Les données sont collectées, puis enregistrées dans un fichier de trace et stockées temporairement sur chacun des nœuds. En fonction des caractéristiques des liens de communication entre les nœuds (notamment leur vitesse), les données observées sont envoyées vers l'un des nœuds, choisi pour générer une trace unique. Si les données ne sont pas envoyées, les données temporaires dans ces nœuds sont localement stockées dans des fichiers.

- *Fonctions de pré-traitement* : L'outil propose deux algorithmes pour l'analyse des données, visant principalement la reconstruction de l'ordre causal des évènements observés. Le premier analyse les communications de style RPC[11] en agissant sur l'ordre relatif de chacun des messages de la trace. Le deuxième analyse des messages, en utilisant des techniques de traitement de signal (algorithme de convolution, présenté dans la section 5.2 de [AMW+03]).

10. Termes anglais, respectivement *port mirroring* et *packet sniffing*.
11. Acronyme anglais : Remote Procedure Call

– *Intrusivité* : Les auteurs ont observé un système distribué J2EE, exécutant l'application PetStore [Sun10]. Les données sont récupérées avec un renifleur de paquets et analysées en avec un algorithme de convolution. PetStore est exécutée sur trois machines, simulant chacune plusieurs nœuds. Pour une exécution d'environ 35 *mn*, l'outil génère une trace d'environ 230.000 messages. L'intrusivité mesurée est d'environ 219%. Les auteurs soutiennent que cette intrusion est acceptable, car l'observation est activée uniquement lors de la phase de débogage.

Synthèse des approches d'observation distribuées

Les approches d'observation des systèmes distribués présentent des informations à un niveau d'abstraction plus élevé. En effet, nous voyons que ces approches produisent dans la plupart de cas, des bilans, de tout le système ou d'une sous-partie, plutôt que des informations élément par élément.

Cette vue de haut niveau sur un grand nombre d'éléments est possible à l'aide de l'établissement des hiérarchies. Ces dernières sont définies en suivant une organisation physique existante (grappe ← nœud) ou une organisation logique. Les hiérarchies permettent l'acheminement de l'information principalement pour qu'elle soit traitée à différents niveaux dans la hiérarchie.

Nous trouvons très intéressante dans ces approches la gestion des éléments des systèmes observés grâce à l'organisation hiérarchique suivie. Nous pensons qu'une organisation hiérarchique, à part de permettre l'application de traitements, peut aider à l'observation partielle des éléments d'un système.

2.6 Synthèse

Ce chapitre nous a permis, dans sa première partie, de donner une définition à l'observation des systèmes informatiques. Nous avons énuméré et décrit les étapes liées à l'observation de ces systèmes, de la définition des entités à observer jusqu'à leur visualisation et analyse. Néanmoins, dans cette thèse, nous n'étudions pas les aspects liés à l'analyse post-mortem, ni à la représentation des données. Nous visons à fournir une approche pour l'observation des prochaines plates-formes embarquées, qui adresse de manière efficace les étapes de définition des données à observer, leur collecte, leur traitement initial ainsi que leur stockage.

Dans le contexte de l'embarqué, nous pouvons signaler deux aspects principaux. Tout d'abord, les outils d'observation sont fortement dépendants des architectures matérielles spécifiques. Or avec l'évolution des MPSoC, les approches existantes devront s'adapter et passer à l'échelle. Ils devront également pouvoir tenir compte d'un grand nombre d'unités matérielles hétérogènes.

D'autre part, les futures techniques devront absolument considérer l'impact de l'observation sur le système cible. En effet, les outils logiciels ajoutent des surcoûts plus

importants que ceux basés sur des composants matériels. Les approches d'observation à venir doivent utiliser des techniques logicielles pour observer au mieux des applications complexes, mais en utilisant des composants matériels des plates-formes afin d'introduire un impact plus faible lors des étapes comme le stockage des données.

Lors de l'étude des approches d'observation pour les systèmes parallèles, nous avons constaté qu'elles sont dirigées principalement par les environnements de programmation. En effet, les approches définissent les éléments et les caractéristiques à observer suivant les flots d'exécution parallèles ou les appels aux fonctions de synchronisation et de communication. Nous avons également constaté que le surcoût introduit par les outils d'observation est mesuré à titre informatif, mais pas en visant la réduction de l'intrusion. L'étroite relation entre les outils et les environnements et la faible considération de l'intrusion dans la conception des outils, font que les approches d'observation des systèmes parallèles ne sont pas directement applicables dans un contexte embarqué.

Malgré les inconvénients énoncés, les propositions dans ce domaine apportent des indications pour faire face aux problèmes des systèmes parallèles. Nous devons nous inspirer des techniques pour l'observation d'un grand nombre d'opérations concurrentes, ainsi que du traitement et de l'analyse des volumes assez importants des données observées.

Enfin, l'observation des systèmes distribués nous apporte des pistes sur l'organisation d'un grand nombre d'éléments et sur l'évaluation d'un état global du système afin d'en simplifier la compréhension. Pour cela, les outils d'observation organisent le système distribué dans des hiérarchies, qui permettent l'acheminement et le traitement des observations. Les propositions étudiées présentent l'observation en tant qu'un ensemble de fonctionnalités bien définies, déployées dans la hiérarchie. Dans un système embarqué, le découplage des étapes liées à l'observation permet de contrôler de manière précise les différentes étapes. D'avantage, le découplage permet également de déployer les différentes étapes de l'observation entre la plate-forme cible et une machine de développement.

Par ailleurs, nous avons souligné que l'utilisation des hiérarchies peut bénéficier à l'observation d'un sous ensemble des éléments du système, permettant l'observation partielle. Nous pensons que cette observation partielle peut permettre d'aborder le défi du passage à l'échelle de l'observation, l'un des défis de l'observation des systèmes sur puce des prochaines années.

Chapitre 3

Les composants logiciels

Nous constatons que le problème d'observation des systèmes sur puce a besoin de solutions flexibles et performantes capables de suivre l'évolution des plates-formes. Nous pensons qu'il est nécessaire de s'abstraire du système cible à observer et donner une définition générique de l'observation qui soit réutilisable sur différentes familles de MPSoC. D'autre part, afin de fournir un mécanisme performant, une solution d'observation devrait être configurable et prendre en compte les spécificités de la plate-forme.

Nous pensons que les besoins pour les solutions d'observation pour les MPSoC peuvent être adressés par l'approche à composants logiciels. Ces derniers ont été proposés afin d'adresser la complexité dans le développement et la gestion d'applications. Ils proposent, en effet, un moyen pour la structuration modulaire d'applications, permettent la réutilisation et surtout, fournissent des techniques pour la gestion d'applications. En ce qui concerne la gestion, les composants essaient de séparer les aspects liés à la configuration du logiciel de la programmation du code métier. La séparation d'aspects dans les nouvelles propositions à composants cherchent à isoler la partie générique du logiciel (la partie réutilisable) de la partie spécifique (celle particulière a la technologie), plus particulièrement, l'exploitation efficace des ressources (hiérarchie de la mémoire, processeurs où les données sont traitées, etc.).

Dans ce chapitre, nous introduisons les concepts de base des composants, puis nous dressons un état de l'art du domaine. Nous considérons différents modèles à composants et regardons les travaux appliqués aux systèmes embarqués ou au problème d'observation. Nous concluons le chapitre par une synthèse sur l'utilité et l'applicabilité d'une approche à composant pour l'observation des systèmes sur puce.

3.1 Principes des composants logiciels

Dans notre travail, nous utilisons la définition de composant logiciel donnée dans [BDH+98] par Stal. Nous voyons les ***composants*** comme des entités autonomes d'encapsulation (boîtes noires) qui fournissent et requièrent des fonctionnalités à leur environnement en utilisant des ***interfaces*** ouvertes et bien définies (cf. figure 3.1(a)). Les

interfaces définissent la syntaxe et la sémantique des fonctions qu'elles représentent. En effet, elles déterminent le contrat entre un composant et son environnement. Étant donné que l'environnement d'un composant est structuré en termes de composants, les interfaces définissent les contrats entre composants.

Afin de construire une application à base de composants, les composants sont interconnectés à l'aide de leurs interfaces (cf. figure 3.1(b)). Certains composants, ainsi que l'application sont donc assemblés par composition de composants. (cf. figure 3.1(c)).

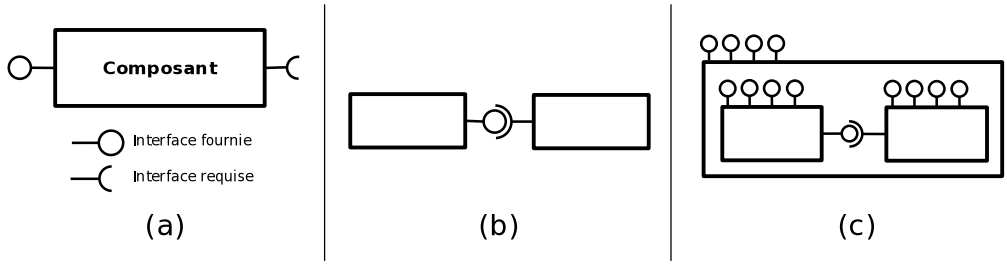

FIGURE 3.1 – Interfaces, connexions et composition des composants

Par rapport à l'approche orientée objets, les composants apportent plusieurs améliorations. Ils permettent de définir des besoins non-liés au code métier (au niveau de la sécurité, la fiabilité, la gestion du cycle de vie, etc.) et explicitent mieux l'architecture des applications. L'approche par composants permet un déploiement sur une plate-forme complexe (p. ex. embarquée), de manière relativement simple et facilement modifiable.

Nous identifions quatre étapes nécessaires pour la mise en œuvre et pour la gestion du logiciel à base de composants logiciels. Il s'agit de la définition, l'implémentation, le déploiement et l'exécution des composants [Mar03].

3.1.1 La définition

Cette étape consiste à décrire l'architecture du logiciel en termes de composants et de liens établis entre eux. Pour cela, plusieurs modèles à composants, comme par exemple Fractal [BCL+06] et CCM [Obj06], utilisent un *langage de description d'architecture* (ADL) et un *langage de description d'interfaces* (IDL). L'*ADL* permet de décrire explicitement la structure de l'application en termes de composants. L'*IDL* permet de définir les fonctions fournies pour chaque interface, et donc les services que les composants offrent les uns aux autres. Parmi les interfaces fournies par les composants, les modèles définissent souvent des interfaces d'***introspection*** et de ***contrôle***. Les interfaces d'introspection révèlent des informations internes au composant comme sa structure, les fonctionnalités offertes, son état, etc. Les interfaces de contrôle varient d'un modèle à un autre et peuvent concerner la gestion du cycle de vie ou la configuration (valeurs de paramètres, choix de services systèmes utilisés, etc.) des composants.

3.1.2 L'implémentation

L'implémentation concerne le développement du code métier des composants, la configuration des services système utilisés, ainsi que l'assemblage des composants dans une application. L'assemblage peut être fait selon un modèle de composition plat ou hiérarchique. Si le modèle est plat, le logiciel se base sur des *composants primitifs*, qui sont des éléments indivisibles de composition et déploiement. Si l'organisation suit des hiérarchies nous pouvons, comme dans le modèle Fractal [BCL+06], définir des composants composés d'autres composants, appelés *composites*. Ces derniers peuvent être décomposées ou réutilisés intégralement.

Le code métier est pris en charge par un développeur de l'application à composants. Les services utilisés sont typiquement fournis dans des canevas et leur utilisation est faite à l'aide d'une chaîne d'outils de génération du code.

3.1.3 Le déploiement

Le déploiement d'une application comprend l'installation de ses composants, la configuration par rapport au contexte d'exécution et le démarrage. Les environnements à composants, par exemple OpenCCM [BCM04] ou EJB [Sun98], fournissent explicitement des fonctions de déploiement.

3.1.4 L'exécution

L'exécution des composants est possible grâce à des *supports d'exécution*, qui permettent le lancement des composants et fournissent des *mécanismes de contrôle* pour la gestion de leur exécution. Le contrôle de l'exécution est aussi connu comme la *gestion du cycle de vie*, qui consiste à contrôler les phases comportementales du composant, comme le démarrage et l'arrêt.

Les supports d'exécution servent à la création d'***instances*** du composant, autrement dit, des entités d'exécution concrète créées à partir de types de composants. L'exécution est possible grâce à des environnements, comme Comete [Ger08], qui fournissent les services nécessaires pour le lancement sur une plate-forme donnée. Parmi les services, nous citons le nommage, la communication entre composants, la sécurité, etc.

3.2 Critères d'évaluation des projets portant sur les composants

Nous nous intéressons à l'utilisation des composants dans les systèmes embarqués ainsi qu'à leur application pour l'observation des systèmes informatiques. Pour cette raison, nous avons structuré notre état de l'art en considérant les critères suivants :

– *Utilisation dans les plates-formes embarquées* : Nous regardons si les travaux ont

déjà été appliqués dans le domaine des systèmes embarqué. Le lien avec le monde embarqué peut être fait soit en modélisant des aspects typiques comme le temps réel ou la consommation énergétique, soit en implémentant l'environnement à composants pour une plate-forme embarquée.

- *Utilisation pour l'observation* : Pour chaque projet nous regardons s'il fournit des fonctionnalités d'observation ou s'il a été utilisé pour faire de l'observation de systèmes.
- *Possibilité d'utilisation et de modification* : Nous nous orientons vers une approche à composants pour définir et implémenter notre approche d'observation. Pour cette raison, il est important de savoir si les projets fournissent des implémentations, si elles sont disponibles et si les sources peuvent être modifiées. D'autre part, nous sommes plus intéressés par des développements en langage C, considéré comme le standard de développement de logiciel embarqué [ROC08].

3.3 Classification des travaux autour des composants logiciels

Dans cet état de l'art, nous choisissons un ensemble de travaux autour des composants logiciels. Nous nous sommes intéressés à trois types de travaux : les approches génériques de développement à base de composants, des travaux développés spécifiquement pour l'embarqué et des projets utilisant les composants pour l'observation.

Les travaux considérés sont listés dans la table 3.1. Cette table nous donne également un premier aperçu de la place de ces projets par rapport aux critères considérés.

	Projet	Embarqué	Observation	Disponibilité
Standards de facto	EJB [Sun98]			X
	CCM [Obj06]	X	X	X
	Fractal [BCL+06]	X	X	X
	`.NET` [TGG+05]	X		
	CCA [AAW+02]		X	X
Approches embarqué	Cecilia et Comete [Obj09a] [Ger08]	X		X
	Koala [vOvdLKM00]	X		X
	TAO [GS99]	X		X
	RUNES [CCM05]	X		X
Approches observation	Composite Probes [Dia07]	X	X	X
	PMM [RTA+04]		X	X
	Rainbow [GCH+04]		X	
	JADE [dPBH+08]		X	X

TABLE 3.1 – Travaux orientés composants considérés

3.3.1 Modèles génériques

Nous commençons notre état de l'art par des modèles génériques de composants c'est-à-dire, qui peuvent être utilisés dans différents domaines d'application. Nous considérons des modèles qui sont devenus des standards de facto : EJB [Sun98], CCM [Obj06], Fractal [BCL+06], `.NET` [TGG+05] et CCA [AAW+02].

3.3.1.1 EJB

Proposé par Sun Microsystems [Sun98], le modèle EJB est utilisé pour le développement d'applications industrielles largement reparties. Dans EJB, une application est constituée de composants, appelés des *Beans*, codés en Java. Les Beans possèdent seulement une interface fournie et ne définissent pas d'interfaces requises. Le type de l'interface requise et plus particulièrement la manière dont elle peut être appelée détermine le type de composant.

Les connexions entre composants ne sont pas établies de manière explicite, mais en récupérant la référence d'une interface de Beans par un service de nommage. Utilisant ces références, les Beans peuvent faire des appels synchrones via Java RMI [Inc] ou asynchrones via JMS [Inc02].

Les EJB utilisent un modèle d'interconnexion à plat. L'impossibilité d'établir des hiérarchies rend difficile leur utilisation dans le cadre de systèmes complexes et à grande échelle.

Les services système utilisés par les Beans sont fournis par le biais d'entités appelés *conteneurs*. Les conteneurs améliorent la réutilisation des composants en permettant leur exécution dans des contextes différents. Par contre, les services proposés sont fixés. Typiquement, ce sont des services de sécurité, de gestion de transactions et de persistance.

EJB compte plusieurs implémentations dans le langage Java, développées principalement pour des serveurs d'applications. Certaines de ces implémentations, comme JBoss [jbo10] et JOnAS [Con], sont disponibles et modifiables. Cependant, l'utilisation de Java ne permet pas son utilisation dans une grande partie de plates-formes embarquées. Les outils pour l'observation des applications EJB sont propriétaires et ne fournissent pas des versions d'essai.

3.3.1.2 CCM

Le modèle CORBA, connu aussi comme CCM, a été proposé par l'Object Management Group [Obj06]. Le modèle définit les composants par des attributs et par leurs interfaces fournies et requises, appelées respectivement des ***ports*** d'entrée et de sortie. Les attributs des composants sont utilisés pour la configuration au déploiement ou à l'exécution. Les ports sont utilisés pour les appels (synchrones ou asynchrones) et l'interconnexion de composants. Le bus CORBA est utilisé pour l'acheminement de toute communication entre les composants. CCM utilise un modèle de composition à plat qui ne permet pas l'établissement de hiérarchies.

Les services système sont gères par des *conteneurs*, qui gèrent les instances d'un type de composant. Ils disposent d'objets d'interposition pour les ports d'entrée et de sortie des instances de composant. Les conteneurs offrent deux interfaces particulières de gestion des objets d'interposition : l'une d'introspection et l'autre de gestion de ports. Le modèle oblige à implanter un nombre fixe de services système pour tous les composants. Cet inconvénient peut être assez coûteux sur des plates-formes ayant des ressources limitées.

La description des types de composants est faite avec des langages déclaratifs à l'aide d'une extension du langage IDL de CORBA. L'IDL support une programmation des composants dans différents langages.

L'implémentation des composants est décrite avec un langage de description d'implémentation des composants (CIDL) qui contient la spécification de la structure logicielle et les services utilisés par les composants.

Le modèle CCM dispose de plusieurs implémentations. Nous trouvons OpenCCM [BCM04], qui est développé en langage Java et permet le déploiement et l'exécution sur plusieurs configurations logicielles. De même, certains développements industriels [iCM10] [Hru01] ont été proposés. Récemment, CCM a proposé un support expérimental compatible avec des plates-formes embarquées ayant Windows CE [Cor10] comme système d'exploitation.

3.3.1.3 Fractal

Fractal [BCL$^+$06] est un modèle général, ouvert et réflexif 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 du consortium *OW2*[1] 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.

Fractal adhère à la définition commune de composant où ce dernier représente un module de code réutilisable et déployable de manière indépendante. Un composant Fractal définit des interfaces fournies et des interfaces requises qui sont utilisées pour les interconnexions.

Les composants Fractal fournissent deux types particuliers d'interfaces, qui sont les *interfaces de contrôle*, et l'*interface d'introspection*. Les interfaces de contrôle permettent la gestion et la configuration des composants. Elles gèrent le « cycle de vie » des composants et les connexions qu'ils ont avec d'autres composants. L'interface d'introspection est symétrique : elle fournit des informations sur les composants, le cycle de vie et les connexions.

Le modèle Fractal se démarque par sa *récursivité* et le *partage de composants*. La récursivité, concept au cœur de Fractal, permet la définition de *composants composites* obtenus par composition de composants. Les composites ont les mêmes fonctionnalités (interfaces d'introspection et de contrôle) que les *composants primitifs*, qui sont des « boîtes

1. Site Web : `http://www.ow2.org/`.

noires ». Les composants composites sont un moyen pour construire des hiérarchies entre les composants et ainsi organiser la complexité des architectures. Le partage de composants consiste à avoir une instance d'un même composant qui peut être partagée par plusieurs composants.

Le modèle Fractal ne spécifie pas d'implémentation, ni de modèle d'exécution. Plusieurs implémentations ont été définies dans plusieurs langages. Le modèle prévoit des possibilités d'interopérabilité entre ces différentes implémentations.

3.3.1.4 Le modèle à composants `.NET`

Le modèle `.NET` [TGG+05] est développé par Microsoft et inspiré par son modèle à objets COM [Box97]. Les composants diffèrent des objets par des fonctions élaborées d'introspection et de nommage.

Les composants `.NET` donnent la possibilité d'organiser les applications à l'aide de hiérarchies.

Les composants `.NET` sont conçus pour être indépendants du langage de programmation. Cependant, l'implémentation doit être faite dans les langages de programmation fournis par la suite de développement Visual Studio [VSN06]. Si les classes qui définissent les composants et les conteneurs peuvent être étendues, le code source des composants n'est pas disponible.

La chaine d'outils de `.NET` produit du code binaire générique supporté par l'environnement d'exécution de `.NET` (CLR). L'environnement charge, exécute et gère les types de composants dans un langage intermédiaire (IL) dans lequel toutes les langages supportées par `.NET` sont compilés.

Nous remarquons que le modèle `.NET` permet d'étendre la définition de composant. Cet aspect pourrait permettre une éventuelle utilisation des composants `.NET` pour l'observation. Un inconvénient est le fait que le modèle est programmable que dans des langages de programmation de Microsoft, compatibles avec la CLR. Nous cherchons à utiliser des langages ouverts et de préférences acceptés largement dans le développement embarqué : le langage C et maintenant dans une moindre mesure C++ et Java.

3.3.1.5 CCA

CCA (Common Component Architecture) est un modèle proposé pour la programmation des applications scientifiques [AAW+02]. Son objectif est de promouvoir la réutilisation du code et la collaboration interdisciplinaire dans la communauté de calcul de haute performance. Il utilise les principes des composants, des interfaces et de composition.

Une application CCA consiste en un ensemble de composants reliés entre eux à l'aide des interfaces fournies et requises. Les ports et les connexions sont définis dans le langage *SIDL*[2] établi dans le cadre de CCA. Une fois que la définition est faite, le canevas du

2. Acronyme anglais : *Scientific Interface Definition Language*

CCA utilise la description pour créer, configurer et assembler les composant de l'application. À l'exécution, le modèle définit les composants comme des entités indépendantes d'exécution.

Les auteurs de CCA ont fourni une implémentation de référence, développée en C++, appelée CCAFFEINE [AAW+02]. Elle permet la programmation d'applications Fortran et C et a été conçue pour des applications à mémoire partagée . Le modèle d'exécution parallèle suivi par CCAFFEINE est *Single Component Multiple Data* (*SCMD*). Il s'agit d'une extension de *SPMD*[3], qui au moment de démarrer le calcul parallèle instancie autant composants CCA, que de processeurs utilise pour le calcul. Puis *SCMD* assigne la portion de données correspondante à chaque composant.

CCA ne fournit pas de fonctions d'observation et son implémentation n'est pas utilisable sur des systèmes embarqués. Néanmoins, le modèle propose des composants avec leur propres flots d'exécution. Ce dernier point peut être intéressant à considérer lors de la construction d'une infrastructure d'observation où il faudrait organiser les traitements d'observation de manière séparée des traitements applicatifs.

3.3.1.6 Synthèse sur les modèles à composants génériques

Les modèles à composants génériques comme EJB, CCM ou CCA sont devenus des standards de facto. Cependant, ils ne ciblent pas le domaine des systèmes embarqués et ne peuvent donc pas être directement appliqués. D'ailleurs, ils ne considèrent pas explicitement le problème de l'observation et sont difficiles à étendre.

3.3.2 Utilisation des composants dans les systèmes embarqués

La plupart de projets orientés composants dans le domaine de l'embarqué ont été conçus pour gérer l'évolution et le portage du code entre les différentes versions des plates-formes. Nous trouvons des propositions généralistes, ainsi que des implémentations ciblant des plates-formes embarquées spécifiques.

3.3.2.1 Cecilia et Comete

Cecilia [Obj09a] est l'implémentation de référence de Fractal pour le langage C. Elle a été utilisée dans le cadre des systèmes embarqués par STMicroelectronics à travers un intergiciel spécifique appelé Comete [Ger08]. Dans la suite, nous présentons ces deux environnements.

Cecilia : Il s'agit d'un canevas de développement pour la programmation de composants Fractal en langage C [Obj09a]. Ce canevas fournit une chaîne de développement basée sur des descriptions d'architecture. La chaîne d'outils est implémentée au-dessus de la chaîne de compilation ADL de Fractal. Le rôle de cette chaîne d'outils est de :

3. Acronyme anglais : *Single Program Multiple Data*

- lire un ensemble des fichiers de description ADL et IDL ainsi que d'implémentation en langage ThinkMC. Ce dernier a pour objectif d'étendre le langage C standard, à l'aide des macros, pour permettre aux développeurs l'implémentation des composants Cecilia ;
- générer le code C des composants correspondants ;
- produire le code nécessaire à la configuration requise par l'application ;
- permettre la compilation de l'ensemble des fichiers générés ainsi que la production des binaires exécutables des applications.

En ce qui concerne l'exécution, Cecilia propose un modèle simple qui utilise un seul flot pour l'exécution des applications.

Cecilia offre deux avantages majeurs, si nous visons son utilisation pour servir de support à la définition d'une solution d'observation. D'une part, le projet met à disposition le code source du canevas, ce qui permet une éventuelle extension pour l'observation. D'autre part, vu que Cecilia est une implémentation de Fractal, il est muni de caractéristiques d'introspection, qui peuvent servir de base pour la définition de l'observation.

Comete : Cet intergiciel permet le déploiement et l'exécution d'applications Cecilia sur différentes plates-formes, notamment des plates-formes embarquées [Ger08]. Il comporte environ de 12 500 lignes de code plus 2 500 des lignes ajoutées à Cecilia. L'architecture de la plate-forme est cachée par le concept de *Processing Element* (PE), qui correspond, selon l'architecture choisie, à un cœur, à un processeur ou à une machine sur lequel le code du composant peut s'exécuter.

Comete affecte les niveaux plate-forme d'exécution, intergiciel Comete et utilisateur. Au niveau plate-forme d'exécution, Comete effectue la correspondance entre les PEs et les unités de calcul physiques. Les plates-formes supportées sont soit indépendantes du matériel, dont *procnet* (réseau TCP/IP) et *pthread*, soit dépendantes du matériel, dont la carte Traviata de STMicroelectronics.

Au niveau intergiciel, Comete gère les protocoles de communication, l'allocation mémoire, le déploiement et l'ordonnancement des composants. Les protocoles de communication sont abstraits grâce à des composants communs aux protocoles, qui gèrent les appels entre les composants Cecilia.

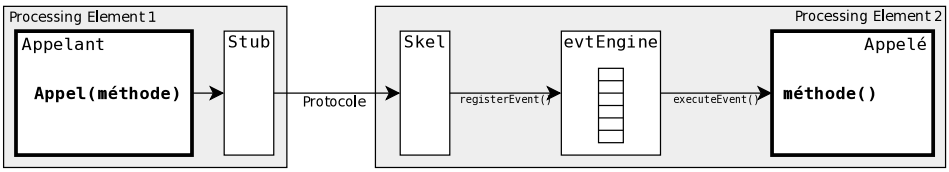

FIGURE 3.2 – Schéma de communication dans Comete

Nous montrons ces composants dans la figure 3.2. Nous voyons que les composants `Stub` et `Skel` sont intercalés entre les composants l'`Appelant` et l'`Appelé`. Un ordonnan-

cement est effectué au niveau de chaque PE afin de gérer les appels concurrents. Ceci est obtenu par l'utilisation d'un `evtEngine`, qui empile et dépile des appels et les exécute un par un.

Les protocoles de communication supportés à l'heure actuelle par Comete sont le *RPC* (appels de méthodes inter-processus), le *CBST* (fonctionnant par évènements [STM07a]), et le *SHMCBST* (spécialisation de *CBST* sans recopies mémoire, pour systèmes à mémoire partagée).

Enfin, au niveau utilisateur, il permet de définir la correspondance entre les PEs et les ressources disponibles de la plate-forme ainsi que de sélectionner le protocole de communication à utiliser.

Nous considérons que cet intergiciel est très intéressant, car il permet d'abstraire les aspects de communication et déploiement de la plate-forme. Son utilisation à STMicroelectronics augmente notre intérêt d'utiliser une infrastructure de type Fractal (modèle, chaîne de développement et de déploiement en langage C) pour la définition de notre approche d'observation. Par ailleurs, nous pensons qu'une solution d'observation organisée à l'aide de composants peut interagir plus naturellement avec une pile logicielle à base de composants. Enfin, le projet fournit une version ouverte du code source qui supporte des plates-formes génériques (machines multi-cœur, machines distantes communicant par RPC).

3.3.2.2 Koala

Koala est un modèle à composants développé par Philips pour la programmation du logiciel embarqué de leurs téléviseurs [vOvdLKM00]. L'objectif est de permettre la réutilisation, sans surcoût considérable, de code entre les différentes évolutions des plates-formes.

Le modèle suit les principes de base des modèles à composants : des composants comme des unités d'encapsulation et de réutilisation, des interfaces pour fournir et requérir des fonctionnalités, et la composition pour construire des applications. Le modèle définit un type de composants sans interfaces appelé un *module* qui est responsable de la connexion entre les interfaces requises et offertes.

Dans Koala, les auteurs ont cherché à séparer le code métier de la configuration spécifique à la plate-forme. Ainsi, les développeurs du code métier encapsulent le code dans des composants et ne font aucune hypothèse sur les configurations dans lesquelles leurs composants sont utilisés. De même, les concepteurs de la configuration ne sont pas autorisés à changer le fonctionnement interne d'un composant en fonction des spécificités de la plate-forme.

La séparation proposée par Koala est représentée par trois niveaux logiques (cf. figure 3.3). Le premier niveau correspond à l'application. Le deuxième niveau adresse la communication et des fonctions d'audio et de vidéo. Le troisième niveau s'occupe des configurations spécifiques de la plate-forme. La séparation selon trois niveaux nous semble intéressante puisque, elle permet de se focaliser sur un niveau d'observation particulier.

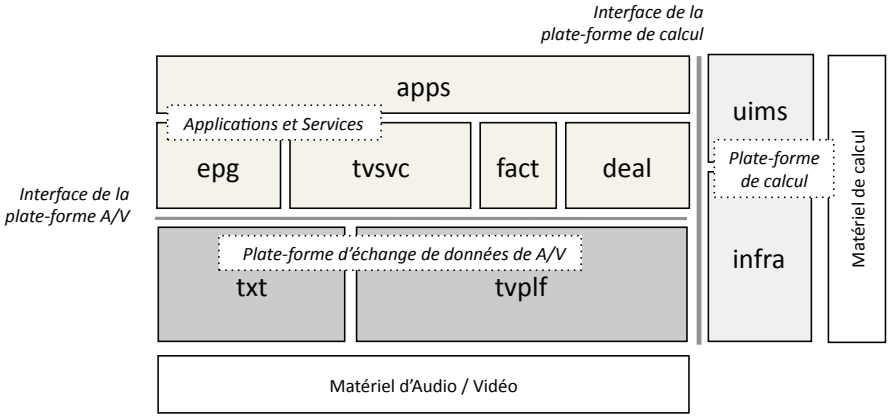

FIGURE 3.3 – Niveaux d'abstraction gérés par Koala

L'implémentation de Koala a été faite en langage C. La plupart des connexions entre les composants sont statiques et connues au moment de la configuration. L'architecture et les interfaces des composants sont décrites respectivement à l'aide des fichiers ADL et IDL. Un langage de description de composants (CDL) est utilisé pour décrire les composants composites.

Malheureusement, Philips ne fournit pas l'accès au code source du logiciel de ses plates-formes de télévision. Cependant, il fournit une version générique du modèle à composants ainsi qu'un compilateur *source to source*[4].

3.3.2.3 TAO

TAO est une implémentation du modèle à composants CORBA [GS99], utilisable sur des systèmes temps réel. Son objectif est de fournir un canevas qui contrôle les politiques et les mécanismes utilisés par CORBA en assurant un ordonnancement adapté. Cette ordonnancement sert à des systèmes ayant des contraintes temporelles soit fortes, comme dans l'avionique (ordonnancement statique), soit souples, comme dans les communications et la multimédia (ordonnancement dynamique) [Sch06] [LL73].

Le modèle à composants CORBA a été optimisé afin d'assurer son fonctionnement sur des systèmes sur puce. Les optimisations cherchent à réduire l'empreinte mémoire et le temps d'exécution. En particulier, les améliorations portent sur la diminution de la taille des structures de composants, principalement sur la réduction de code des talons et des squelettes[5], ainsi que sur des optimisations du compilateur d'IDL.

Afin d'aborder le temps réel, TAO ajoute à CORBA un ordonnanceur temps réel et une API pour la qualité de service. L'ordonnanceur temps réel détermine la priorité des invocations aux composants. L'interface de programmation permet aux applications et aux services de haut niveau de CORBA de spécifier des paramètres de qualité de service

4. Disponibles à l'adresse `http://www.program-transformation.org/Tools/KoalaCompiler`
5. Respectivement, des parties requises et fournies d'une communication entre composants CORBA

à l'aide d'un modèle de programmation orienté objet.

L'implémentation de TAO en langage C++ est ouverte et disponible. À l'heure actuelle, il est utilisé dans des systèmes embarqués industrielles dans différents domaines d'application. Par exemple, il fait partie du logiciel de communication dans des stations de transmission UMTS [6], produites par la compagnie Siemens. Toutefois, il ne propose pas d'extensions permettant l'observation, ni au niveau modèle, ni au niveau implémentation.

3.3.2.4 RUNES

RUNES, basé sur OpenCOM [CBG+08], vise la gestion du logiciel pour des plates-formes avec des ressources très limitées (p. ex. les réseaux de capteurs) ou ayant une configuration logicielle complexe (p. ex. téléphones portables avec un système d'exploitation Linux) [CCM05]. RUNES est par exemple utilisé dans le développement de protocoles réseaux (IP [Def81], UPnP [iso08]) ou la gestion de la consommation énergétique.

Dans RUNES, les composants sont interconnectés à l'aide d'interfaces fournies (appelées interfaces tout court) et d'interfaces requises (appelées réceptacles).

Les composants dans RUNES sont organisés dans des *canevas*. Ce dernier est un groupe de composants ayant une contrainte commune et est géré comme une unité indépendante de déploiement et d'exécution. Les canevas peuvent être reconfigurés lors de l'exécution (ajout et suppression des composants). La description de telles modifications est faite à l'aide de langages comme OCL [7].

RUNES propose deux implémentations pour les plates-formes embarquées : l'une en langage Java et l'autre en langage C. Le code source des implémentations est disponible dans le site Web du projet [8].

L'intérêt de ce modèle dans notre contexte est son utilisation pour l'embarqué. L'implémentation C de ce modèle pourrait nous servir d'infrastructure pour la définition de notre modèle d'observation. Malheureusement, des aspects comme les hiérarchies, définies dans OpenCOM, ne font pas partie de l'implémentation de RUNES.

3.3.2.5 Synthèse sur l'utilisation des composants dans les systèmes embarqués

Nous constatons que les approches à composants sont déjà présentes dans le domaine embarqué. Elles sont utilisées pour des applications différentes, dont la multimédia et la communication, et abordent des aspects divers, dont le déploiement et la qualité de service. Malheureusement, aucune des approches ne cible l'observation comme l'un de ses objectifs.

Les approches étudiées mettent à disposition au moins une version de leur code source.

6. Acronyme anglais : *Universal Mobile Telecommunications System*
7. Object Constraint Language [Gro10]
8. Le projet RUNES : `http://runesmw.sourceforge.net/`

Ce code source est écrit en langage C dans la plupart de cas. Vu que ces implémentations possèdent déjà des optimisations pour l'embarqué, nous pensons qu'il serait possible de réutiliser et d'étendre l'une de ces implantations pour la définition de notre approche d'observation. Nous considérons que les travaux qui permettent une extension plus facile, en termes de modèle et d'implémentation, sont Cecilia-Comete et TAO.

3.3.3 Observation à base de composants

Dans cette section, nous nous intéressons à des approches d'observation à base de composants pour l'observation du logiciel à composants, ainsi que des systèmes distribués. Dans les systèmes à base de composants, il est plus naturel en mettre en œuvre l'observation avec des composants. Dans les systèmes distribués, les défis d'observation peuvent être abordés par les capacités d'établissement de hiérarchies et de déploiement fournies par les composants. En ce qui concerne l'observation des systèmes distribués, nous faisons référence aux approches d'administration de systèmes à base de composants. La gestion de systèmes comporte des activités d'observation, qui supportent la surveillance et le contrôle des systèmes administrés. Les travaux étudiés sont présentés ci-après.

3.3.3.1 PMM

PMM (*Performance Measuring Modeling*) est ciblée sur l'observation d'applications écrites avec des composants CCA (cf. section 3.3.1.5). Il permet de récupérer différents types de données comme, par exemple, le temps d'exécution des fonctions des composants, les temps pour les appels MPI, les paramètres initiaux du calcul, etc.

L'architecture d'observation comporte trois types de composants : un composant *TAU*, plusieurs composants *proxy* et un composant *Mastermind*. Les composants travaillent ensemble pour collecter et mettre à disposition les données observées à l'utilisateur.

Le composant *TAU* [RTA+04] est un composant CCA indépendant de l'application qui utilise la boîte à outils TAU (cf. section 2.5.2.1). Ce composant implémente une interface appelée le *MeasuringPort*, qui définit des fonctions de temps, de gestion de données collectées ainsi que d'évaluation de requêtes de mesure. Les composants *proxy* sont en charge d'intercepter les échanges de messages et les appels à des méthodes entre les composants de l'application. Quand l'observation est démarrée, *proxy* envoie les données collectées vers *Mastermind*. Ce dernier est responsable de la collecte et de l'enregistrement des données observées. Pour chaque fonction observée, une entrée dans la trace est créée. Les estampilles et l'ordre des entrées dans la trace sont fournis par le composant TAU.

Nous illustrons l'architecture d'observation dans la figure 3.4. Nous montrons une application simple de 4 composants (`C1` `C2`, `C3` et `C4`), des composants proxy (`P1` `P2`, `P3` et `P4`), un composant Mastermind (`M`) et un composant TAU (`T`). Les lignes continues indiquent la connexion entre composants et les lignes pointillées les liens entre les *proxies* et le *Mastermind*.

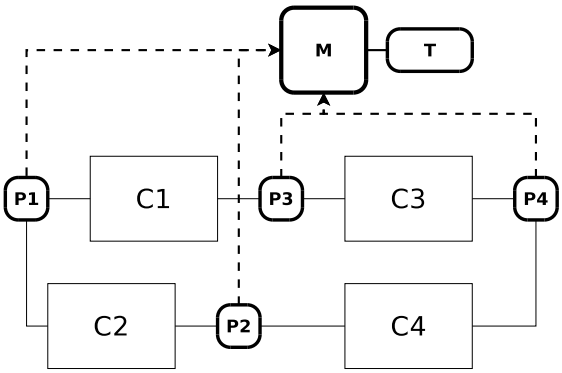

FIGURE 3.4 – Observation d'une application CCA avec PMM

Nous soulignons deux aspects importants de cette approche. Premièrement, l'architecture d'observation définit l'interface *MeasuringPort* pour le contrôle, la gestion et l'accès aux observations. Une interface de ce type peut offrir un point d'entrée unique aux observations du système sous étude. Deuxièmement, chaque type de composant a été spécialisé pour une tâche bien définie : l'interception, l'enregistrement ou le contrôle. Une définition des types de composants nous permettrait de gérer séparément les différentes étapes de l'observation.

3.3.3.2 Composite Probes

Dans le modèle Fractal, les interfaces d'introspection permettent d'obtenir des informations sur la structure des applications construites avec des composants Fractal. Cette capacité d'introspection a permis à d'autres approches, basées sur Fractal, de cibler l'observation. C'est le cas de Composite Probes [Dia07].

L'approche, inspirée par CLIF [9], consiste en l'utilisation des composants pour collecter et traiter des données sur l'exécution des systèmes distribués (p. ex. une grappe de calcul). Il définit une architecture générique qui permet l'observation de différents éléments du système à l'aide d'un ensemble défini d'interfaces. Il permet aussi l'observation des systèmes disposant d'un grand nombre d'éléments.

Composite Probes définit deux types de composants : les uns, des *sondes basiques*, sont chargés de la collecte de données brutes directement sur le système observé ; les autres, des *sondes composites*, permettent de recueillir les données collectées par les basiques et d'effectuer des traitements sur ces données. Nous montrons dans la figure 3.5 que les sondes basiques et les sondes composites sont organisées dans une hiérarchie pour l'échange des données, puis leur traitement.

Nous voyons dans la figure que les données observées sont collectées et ensuite acheminées vers le niveau supérieur dans la hiérarchie. Nous pouvons faire correspondre certaines étapes de l'observation avec des traitements faits par les sondes. Par exemple, la préparation de la cible à observer et la collecte des données brutes correspondent aux trai-

9. Il s'agit d'un canevas pour l'injection de charge sur des systèmes distribués [Dil09].

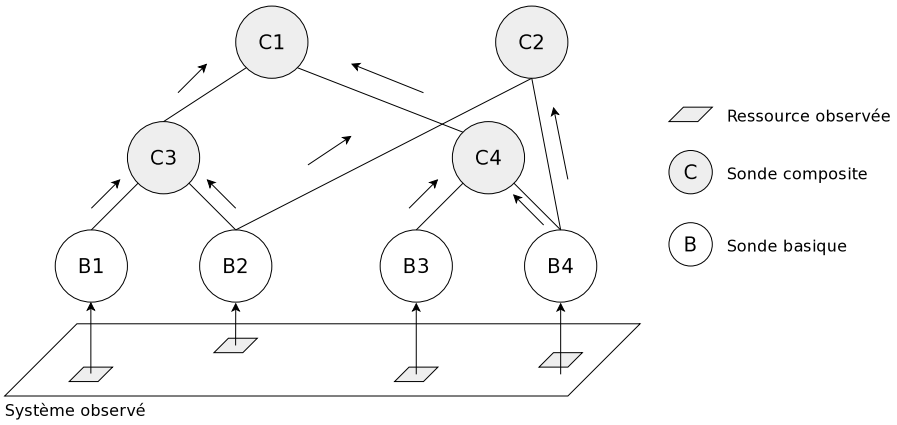

FIGURE 3.5 – Hiérarchies dans Composite Probes

tements faits par la sonde basique. De même, le pré-traitement des données correspond aux traitements effectués par une sonde composite.

Cette approche est développée en utilisant l'implémentation de référence de Fractal en Java et elle est disponible sur le site Web du projet. Malheureusement, même si les principes de gestion des sondes et de hiérarchies de Composite Probes donnent des pistes très intéressantes, leur implémentation ne peut pas être utilisée sur des plates-formes embarquées sans effectuer des adaptations considérables.

3.3.3.3 Rainbow

Cette approche propose un canevas d'administration de systèmes logiciels auto-adaptables, à l'aide d'architectures logicielles réutilisables [GCH+04]. L'un de ses objectifs est de favoriser la portabilité du canevas entre systèmes. Cela est abordée en divisant le canevas en couches, l'une générique à l'architecture, et l'autre spécifique au système (cf. figure 3.6). La correspondance entre ces couches est gérée par un service de traduction, fourni aussi par Rainbow.

La couche du système est composée des sondes pour observer le système en exécution, des effecteurs pour modifier sa configuration du système et d'un élément de découverte de ressources.

La couche d'architecture définit les contraintes, les réglés et les stratégies pour l'adaptation de Rainbow. Cette couche comporte un agrégateur des données des sondes ; un administrateur du modèle, qui gère l'accès au modèle ; un évaluateur de contraintes, qui notifie les non conformités ; et un moteur d'adaptation, qui détermine l'action à prendre suite à l'évaluation, puis le moteur exécute l'action.

Rainbow implémente la couche d'architecture en langage Java. Le langage d'implémentation de la couche du système varie en fonction du système observé. Cependant, le projet ne met pas à disposition son code source. Enfin, Rainbow a été utilisé dans une étude de cas, présentée dans [GCH+04], pour l'administration d'un système de vidéo

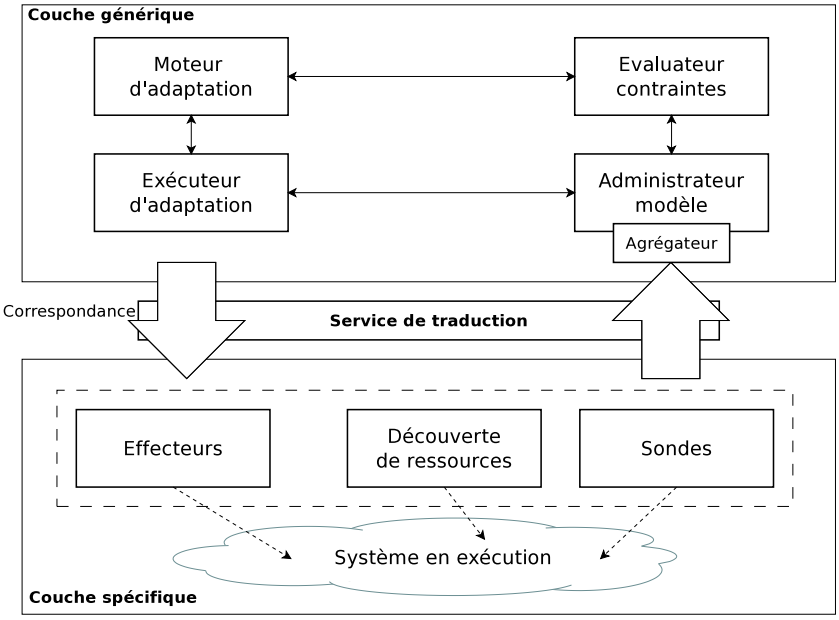

FIGURE 3.6 – Le canevas de Rainbow

conférence, qui utilise des dispositifs embarqués. Malheureusement, cette utilisation ne fait pas ressortir des particularités de l'embarqué pour les intégrer au canevas.

3.3.3.4 JADE

Basé sur Fractal, JADE vise l'administration autonome d'infrastructures logicielles [dPBH+08]. Leur objectifs principaux sont la réparation et l'optimisation automatiques de l'infrastructure. Le premier objectif, cherche à rétablir l'infrastructure dans un état cohérent après une panne. Le deuxième objectif, cherche à maintenir la performance de l'infrastructure malgré les variations de charge. JADE cible principalement les systèmes reparties, dont leurs éléments ont un couplage faible, comme les serveurs d'applications Web.

JADE fournit essentiellement deux fonctionnalités : il encapsule des ressources administrées, en fournissant une interface d'administration uniforme, et il permet la construction de gestionnaires autonomes, qui administrent à l'exécution un ensemble de ressources suivant une politique particulière.

L'encapsulation de ressources est possible grâce à l'application du concept de *wrapper*. Cela consiste à encapsuler des ressources dans des composants, appelés « élémentaires », qui peuvent être connectés pour représenter le lien entre les éléments d'une infrastructure logicielle. Nous montrons dans la figure 3.7, l'administration d'un Serveur Web à l'aide de JADE.

Nous voyons que la couche d'administration dispose d'un ensemble de composants élémentaires, qui encapsulent les éléments du Serveur Web pour son administration. Nous

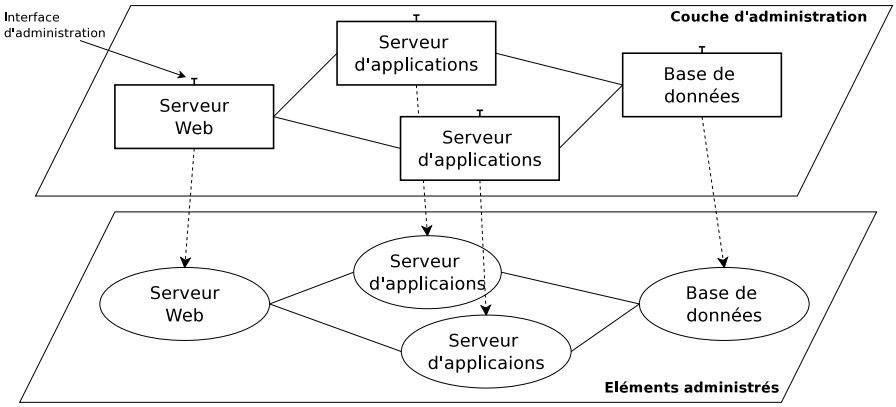

FIGURE 3.7 – Administration des éléments d'un Serveur Web

voyons aussi que les composants élémentaires établissent entre eux les mêmes liens existants entre les éléments administrés.

La construction de gestionnaires est fournie par l'établissement de politiques d'administration, qui sont représentés par des « boucles de contrôle ». Ces boucles observent que la politique pour laquelle elles ont été crées soit respectée. Si ce n'est pas le cas, elles déclenchent des actions de contrôle du système, définies par la politique.

Le support de JADE pour les systèmes embarqués n'est pas encore considéré, car JADE cible des plates-formes et des objectifs assez différents. Cependant, nous pensons que les principes d'encapsulation d'éléments pour leur administration et observation peuvent être très intéressants pour l'observation d'entités dans un système embarqué.

Par rapport à l'implantation, les auteurs proposent une implémentation en langage Java s'exécutant sur une plate-forme J2EE. Cette implémentation, qui reflet les objectifs de JADE, n'est pas adaptée pour son utilisation sur les MPSoC courants.

3.3.3.5 Synthèse sur les travaux autour de l'observation à base de composants

Les approches d'observation étudiées se caractérisent par la définition et l'établissement d'un moyen d'accès aux informations du système sous étude. Si le système est organisé dans des composants logiciels, le moyen d'accès comporte l'ajout d'interfaces d'observation aux composants. Dans le cas contraire, l'approche d'observation crée des composants pour encapsuler les éléments du système. Dans les deux cas, l'objectif est de fournir un niveau de base structuré, qui fournit les observations uniquement à travers un ensemble d'interfaces de composants.

Nous constatons que les approches étudiées utilisent les observations, fournies par ce niveau de base, soit pour les enregistrer directement, soit pour construire des architectures et effectuer des traitements sur les observations. Ces architectures peuvent suivre une organisation hiérarchique, qui permet d'acheminer les résultats des traitements vers des niveaux supérieurs dans cette hiérarchie.

3.4 Synthèse

Dans ce chapitre, nous avons brossé un état de l'art sur les composants logiciels. Nous pensons que les composants peuvent répondre aux besoins d'observation des systèmes embarqués. D'autant plus que les composants trouvent une application croissante dans le domaine.

Nous avons organisé les travaux étudiés en trois groupes en fonction de leurs applications. Nous avons considéré les travaux généralistes, les projets appliquant les composants dans le domaine de systèmes embarqués et enfin, les travaux utilisant les composants pour observer des systèmes informatiques.

Les modèles généralistes explicitent les principes des composants à reprendre dans notre approche d'observation. Cependant, les implémentations offertes par la plupart, restent inutilisables ou insuffisantes pour les systèmes sur puce. En effet, des modèles comme EJB ou `.NET` offrent moins de possibilités pour étendre leurs modèles avec des services système, très nécessaires pour l'observation des embarqués. Parmi les modèles génériques, nous trouvons que CCA n'est pas une possibilité envisageable à court terme, car il est fortement lié aux environnements d'exécution parallèle, qui ne sont pas encore définies pour les plates-formes sur puce. Entre CCM et Fractal, nous considérons que le modèle Fractal définit des concepts nécessaires pour notre approche d'observation, comme la composition d'applications à l'aide des hiérarchies, que nous ne trouvons pas dans CCM.

Les modèles conçus pour l'embarqué partagent des objectifs comme la réutilisation du code entre les évolutions des plates-formes, ainsi que des optimisations pour fournir des implémentations plus performantes pour les systèmes sur puce. Un aspect très intéressant est le fait que pour permettre la réutilisation du code, les approches proposent plusieurs niveaux d'abstraction dans les composants de la plate-forme : un premier niveau où les spécificités matérielles sont décrites ; un deuxième niveau qui fournit une vue de haut niveau sur les fonctions offerts par le matériel (p. ex. décodage et communication) ; et un troisième niveau applicatif qui exploite la plate-forme, mais tout en restant indépendant du matériel. Cela nous donne des idées pour effectuer l'observation dans différents niveaux d'abstraction.

Les approches d'observation à base de composants cherchent à utiliser des interfaces génériques pour le contrôle de l'observation, ainsi que pour la récupération de données. Les travaux Composite Probes et PMM, proposent des composants spécialises et l'utilisation des hiérarchies pour l'organisation et le traitement des observations. Nous pensons que ces deux aspects sont fondamentaux pour l'observation d'un grand nombre d'éléments dans un système embarqué.

Deuxième partie

Contribution

Chapitre 4

Proposition

La conception d'outils pour le développement d'applications embarquées est, dans la plupart des cas, dirigée par des besoins spécifiques des plates-formes. En effet, nous avons constaté, grâce à notre étude de l'état de l'art et à l'expérience acquise avec les outils de développement logiciel chez STMicroelectronics, qu'à chaque génération des plates-formes, la chaîne d'outils doit évoluer sensiblement.

Étant donné que les plates-formes embarquées augmenteront encore plus leur complexité, en termes du nombre de processeurs et d'organisation de la mémoire, les approches courantes d'observation ne passeront pas à l'échelle. Nous considérons qu'au delà des techniques de collecte de données permettant le débogage de systèmes sur puce, nous avons besoin d'une méthode pour la structuration des informations produites lors de l'observation. Cette méthode doit permettre la gestion des différentes étapes de l'observation, d'une part, et d'autre part, la structuration de grands volumes de données.

Ce chapitre est consacré à la présentation de notre proposition d'observation pour les systèmes embarqués. Il s'appuie sur les conclusions que nous avons tirées des travaux étudiés dans l'état de l'art sur l'observation et sur les composants logiciels.

4.1 Synthèse des travaux étudiés

L'étude des travaux sur l'observation nous donnent les pratiques courantes pour la collecte des données et leur analyse dans différents contextes. Cette étude montre que les objectifs des contextes analysés ont des besoins différents, mais qu'ils utilisent souvent les mêmes techniques pour l'observation. Cependant, la réalisation de ces techniques deviennent de plus en plus dépendantes des plates-formes sous-jacentes, particulièrement dans le contexte embarqué. Ces derniers posent en plus des contraintes importantes pour les solutions d'observation dues à leur évolution rapide et aux ressources limitées.

Nous présentons, ci-après, les points les plus importants que nous avons trouvé dans l'observation des systèmes informatiques :

- *Étapes de l'observation* : Nous avons présenté une suite d'étapes pour observer un système informatique. Nous considérons qu'une approche pour l'observation des systèmes embarqués, doit tenir compte des étapes affectant directement la performance du système embarqué. Ces étapes sont la sélection d'entités, l'instrumentation, la collecte, le pré-traitement et le stockage.

- *Observation des systèmes embarqués* : Nous avons étudié des approches d'observation spécifiques aux plates-formes, ce qui permet une observation très précise et performante aux niveaux matériel et système. Cependant, cette spécificité rend difficile la réutilisation des approches pour d'autres familles de plates-formes embarquées. D'ailleurs, la plupart des approches ne considèrent pas les aspects liés aux niveaux intergiciel et applicatif de la pile logicielle, qui deviennent partie de la pile logiciel des systèmes embarqués courants.

- *Observation des systèmes parallèles* : Nous constatons que l'observation des applications parallèles est dirigée principalement par les environnements de programmation. Ainsi, les solutions d'observation de programmes codées avec des Threads, en MPI ou en OpenMP sont différents. La dépendance des environnements fait que les éléments à observer sont prédéfinis dans la plupart de cas.
  Les approches d'observation des systèmes parallèles ont mis au point des mécanismes pour la gestion de grands volumes de données, ainsi que pour la synchronisation des observations. Nous pensons que les évolutions dans l'embarqué nous amèneront vers des plates-formes sur puce massivement parallèles, ce qui veut dire que nous devons tenir compte de leurs propositions d'observation pour les appliquer à l'embarqué. Néanmoins, pour les rendre utilisables sur les MPSoC, les approches doivent être capables de supporter les environnements de programmation embarqué, ainsi qu'optimisées, en termes d'empreinte mémoire et d'intrusion ajoutée.

- *Observation des systèmes distribués* : Dans ce contexte, l'observation distribuée doit faire face à des problématiques de systèmes largement répartis, comme la difficulté du calcul d'un état global et la gestion d'un grand nombre d'éléments. Cette dernière est adressée par les approches d'observation à l'aide de l'établissement de hiérarchies entre les éléments du système. Comme nous pensons que les prochaines plates-formes embarquées pourront être aussi distribuées, nous considérons qu'une approche hiérarchique pour la gestion d'observations facilitera le traitement de données.

Les conclusions fournies par l'état de l'art sur l'observation, nous ont amené à chercher des solutions pour nous abstraire de la hétérogénéité des techniques d'observation, ainsi que des contraintes des plate-formes embarquées. C'est pour quoi, nous avons étudié l'approche à composants logiciels.

Les travaux sur les composants ont été dirigés par des besoins d'organisation et de gestion du logiciel. Les approches étudiées ont considéré, d'une part, des modèles génériques permettant la définition de toutes les étapes du développement. D'autre part, nous avons étudié des travaux spécifiques, répondant à l'évolution des plates-formes d'une même famille embarquée.

En résumant, les concepts, les propriétés et l'utilisation de l'approche à composants que nous pouvons reprendre pour concevoir une solution d'observation sont :

- *L'encapsulation* : nous pensons qu'elle nous permettra d'abstraire des éléments du système sous observation, avec des composants, afin de les traiter de manière indépendante. D'autre part, les composants peuvent nous permettre de grouper des ensembles de fonctionnalités consacrées à l'observation. Par exemple, un ensemble dédié à la collecte des observations, l'autre à l'enregistrement de données, etc.
- *Les interfaces* : comme les interfaces sont le point d'entrée et de sortie d'un composant, nous pensons que elles serviront à fournir les observations de l'entité observée.
- *Le placement de traitements* : étant donné qu'un composant peut grouper des fonctionnalités dédiées à une étape de l'observation, nous pouvons cibler le placement et l'exécution des composants spécifiques sur des ressources du MPSoC, et d'autres composants sur des machines de développement distantes.
- *L'utilisation pour l'observation* : parmi les travaux sur les composants appliqués à l'observation, nous avons constaté que certains utilisent l'introspection pour offrir des informations sur leur structure. De même, les composants regroupent des fonctions, comme la collecte de données et le traitement de données. Enfin, les composants établissent des hiérarchies afin d'organiser les observations.
- *Leur utilisation dans l'embarqué* : nous constatons que l'approche à composants devient une solution pour la réutilisation du logiciel dans les plates-formes embarquées, notamment dans des produits Philips et STMicroelectronics. Dans cette évolution vers des solutions modulaires et réutilisables, nous pensons qu'une approche d'observation basée sur des composants peut interagir, plus naturellement, avec du logiciel organisé via des composants.

4.2 Objectifs de l'approche

Nous proposons dans cette thèse une solution d'observation pour l'observation des systèmes multiprocesseurs sur puce. L'un des nos objectifs fondamentaux est de démontrer l'utilité des composants logiciels, comme une solution pour adresser les limitations des solutions existantes, en particulier le manque de généricité. Nous définissons plus précisément les objectifs suivants :

- ***Généricité :*** Il doit être possible d'utiliser notre solution pour observer différentes plates-formes matérielles et logicielles. Pour cela, la solution devra permettre l'observation d'éléments matériels et logiciels, en les accédant à travers un ensemble unique d'interfaces.
- ***Observation multi-niveaux :*** Notre solution doit être en mesure d'assurer une observation au niveau matériel, mais aussi de fournir des informations sur les niveaux logiciels supérieurs, y compris l'intergiciel et les applications embarqués. Il

devra fournir des mécanismes de corrélation de l'information produite à différents niveaux. Par exemple, nous pouvons effectuer les observations a tous les niveaux en ajoutant une donnée commune, comme l'estampille, afin de établir une relation de temps entre les observations.

- ***Observation partielle :*** Observer ce qui se passe lors de l'exécution à chacun des niveaux dans un système embarqué n'est pas réaliste, en particulier dans le contexte des architectures *many-core*. Notre solution devra permettre l'observation d'une partie spécifique d'un système embarqué, qu'il s'agisse d'un composant matériel, comme un cœur ou un groupe des cœurs, un niveau du logiciel, comme le niveau de communication, ou un module d'application, comme un décodeur.

- ***Passage à l'échelle*** : L'observation d'un grand nombre d'éléments embarqués doit pouvoir s'organiser en suivant des hiérarchies pour l'acheminement des observations ainsi que proposer des composants qui traitent et, si possible, réduisent ou fusionnent les données collectées.

- ***Configurabilité :*** Il doit être possible de configurer notre solution générique afin de tenir compte des caractéristiques spécifiques de la plate-forme cible. Par exemple, il convient d'être en mesure d'utiliser une référence de temps fournie par la plate-forme, de déployer les fonctionnalités d'observation sur les processeurs disponibles d'un système multi-cœur ou d'utiliser un mécanisme de stockage de traces supporté par le matériel.

- ***Intrusivité minimale :*** Dans toutes les solutions consacrées à l'observation, une préoccupation majeure est le contrôle de l'intrusivité ajoutée au système par les mécanismes d'observation. La solution d'observation devra avoir un impact minimal sur le comportement des systèmes embarqués et devra également fournir les moyens pour mesurer et contrôler l'intrusion de l'observation.

4.3 Proposition

Nous proposons un modèle à base de composants logiciels pour l'observation des systèmes embarqués, respectant les propriétés définies ci-devant. Notre choix d'utiliser des composants logiciels a été motivé par le fait qu'ils se sont avérés être une solution efficace dans le développement, le déploiement, la gestion et la maintenance de grands systèmes logiciels [Obj06] [Sun98]. De plus, ils ont une popularité croissante dans le contexte industriel et en particulier dans le domaine des systèmes sur puce [Crn04].

Dans le modèle proposé, nous visons premièrement à fournir une couche générique pour abstraire la hétérogénéité des plates-formes matérielles et logicielles. C'est pourquoi nous proposons d'encapsuler les entités embarquées spécifiques et de définir un ensemble d'interfaces génériques pour l'observation.

Nous fournissons une observation multi-niveaux à l'aide de la capacité naturelle des composants à établir des hiérarchies. L'observation partielle est obtenue par l'instancia-

tion de composants d'observation correspondant uniquement aux entités qui sont observées.

Nous pensons que le passage à l'échelle est possible en faisant de l'observation partielle, autrement dit, en tenant compte que d'une sous-partie des éléments du système. De même, si nous avons beaucoup d'éléments à observer produisant un grand volume de données, nous établissons des hiérarchies entre les composants, afin de réduire la quantité de données en les agrégeant ou en les filtrant.

De manière à permettre la configurabilité, nous définissons des extraits de code dans lesquels, certaines parties doivent être complétées ou réécrites pour intégrer des traitements spécifiques. Ceux-ci correspondent principalement à l'implémentation des aspects propres à la plate-forme, comme l'utilisation de l'horloge ou d'un port de trace.

Enfin, pour assurer le contrôle de l'intrusivité, nous envisageons une mise en œuvre efficace pour la plate-forme sur puce, ainsi que la minimisation de l'intrusion due à l'instrumentation lors de l'intégration de notre solution avec la plate-forme à observer. De plus, nous pensons réaliser l'implantation de notre modèle à base des composants à partir de zéro, en implémentant seulement une infrastructure minimale dans un langage de programmation optimisé pour l'embarqué. Une implémentation de ce type réduit l'empreinte mémoire du modèle à l'utilisation.

Dans le chapitre suivant, nous présentons EMBera, notre approche à base de composants logiciels pour l'observation des systèmes embarqués.

Chapitre 5

EMBera : un modèle à base de composants logiciels pour l'observation de systèmes embarqués

Dans ce chapitre, nous présentons *EMBera*, notre approche à base de composants logiciels pour l'observation de systèmes embarqués. Nous décrivons le modèle à composants utilisé et sa spécialisation pour l'observation. Nous présentons les aspects génériques de mise en œuvre du modèle suivis d'une suite de recommandations pour son utilisation.

5.1 Le modèle à composants EMBera

Nous adoptons un modèle à composants et le spécialisons pour l'observation. Notre modèle à composants est fondé sur le modèle Fractal [BCL+06], introduit dans la section 3.3.1.3.

Une application EMBera est un ensemble de composants connectés entre eux. Dans le modèle, un composant est une entité logicielle avec une fonctionnalité bien définie. Les composants EMBera suivent les principes de base des composants (cf. section 3.1) : ils offrent leurs fonctionnalités à travers d'*interfaces fournies* et accèdent à des fonctionnalités d'autres composants par le biais d'*interfaces requises*. Les *connexions* entre composants sont établies en reliant les interfaces requises aux interfaces fournies du même type.

En outre, les composants EMBera sont munis d'une interface prédéfinie de *contrôle*. Pour la création des composants, l'interface définit la fonction `createComponent`. Pour l'interconnexion des composants, elle crée des interfaces à l'aide de `createInterface` et les connecte par le biais de `componentConnect`. Le cycle de vie du composant est contrôlé (contrôle du lancement et arrêt) par les fonctions `startComponent` et `componentWait`. La figure 5.1 illustre les interfaces et la connexion entre des composants EMBera.

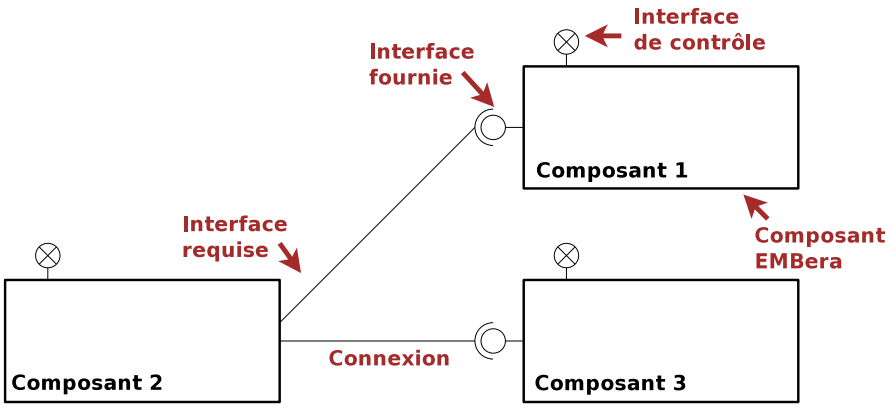

FIGURE 5.1 – Le modèle à composants EMBera

Les composants EMBera fournissent une interface d'introspection, héritée du modèle Fractal (cf. section 3.3.1.3). Cette interface fournit une fonction `getComponentInterfaces`, qui offre une sous-partie des informations de l'interface Fractal, notamment, la liste d'interfaces avec leur nom, leur type et les méthodes fournies.

Dans EMBera, les composants d'une application communiquent à l'aide de messages. Les interfaces peuvent être vues comme des points de communication qui sont utilisés pour l'envoi et la réception de ces 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.

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.

5.2 Le modèle d'observation EMBera

Nous nous servons de notre modèle à composants comme une couche générique sur laquelle nous définissons l'observation. Dans cette définition, nous cherchons à modéliser les étapes de l'observation des systèmes sur puce (cf. section 2.2). Nous visons à modéliser plus précisément la sélection d'entités à observer, l'instrumentation de la cible, la collecte des données brutes, le pré-traitement de données, le formattage et le stockage. Ces étapes sont modélisées en spécifiant des interfaces dédiées à l'observation et des types de composants, qui regroupent les traitements d'une ou de plusieurs étapes.

Pour la définition du modèle d'observation, nous abordons premièrement l'encapsulation des éléments du système à l'aide de composants EMBera, ce qui nous permet de représenter séparément les entités afin de les observer et qui permet la sélection d'entités

à observer. Puis, nous spécifions les interfaces spécialisées du modèle d'observation EMBera. Enfin, nous présentons trois types de composants qui abordent les traitements des étapes de l'observation. Ces composants se chargent : le premier, l'instrumentation du système et la collecte des données, le deuxième, des traitements initiaux sur les données collectées et, le troisième, de l'enregistrement des données résultantes.

5.2.1 L'encapsulation d'entités

Le principe de base de l'observation est l'identification d'éléments d'intérêt du système sous observation (*entités*). L'encapsulation pour l'observation consiste à représenter ces entités par des composants qui reflètent leurs caractéristiques et suivent leur évolution. Autrement dit, un composant va se charger de l'observation de l'entité, en collectant les informations nécessaires et en les transmettant pour leur traitement ultérieur.

Sur les plates-formes embarquées courantes, nous pouvons identifier des entités présentes à différents niveaux :

- *Niveau logiciel* : nous avons utilisé trois niveaux d'observation pour placer les entités, notamment, le système d'exploitation, l'intergiciel et l'application. Dans la table 5.1, nous montrons des exemples d'entités par niveau.

Niveau d'observation	**Entité**	**Informations observées**
Système d'exploitation	Flots d'exécution	Échantillons d'utilisation de la mémoire
Intergiciel	Primitives de communication entre flots	Mesure du temps d'attente
Application	Fonctions utilisateur	Valeurs de retour des fonctions

TABLE 5.1 – Exemples des entités et des évènements par niveau d'observation

  Le composant peut récupérer des informations (par exemple des fichiers système), soit en utilisant des mécanismes externes à l'application, soit en accédant (modifiant) directement au code de l'application. Le choix du mécanisme d'encapsulation code dépend principalement de la disponibilité du code source et de la possibilité de modification.

- *Niveau matériel* : l'encapsulation d'entités materielles (cœurs, bus de communication, niveau de cache) se fait en accédant à des interfaces logicielles spécifiques, fournies par le constructeur de puce.

L'encapsulation d'entités est le premier pas pour l'observation des systèmes embarqués avec des composants EMBera. Nous considérons que la représentation faite des éléments du système par des composants permet d'abstraire la hétérogénéité des entités et des mécanismes pour la récupération d'information.

5.2.2 Les interfaces spécialisées

Suite à l'encapsulation, les interfaces spécifiques établissent le niveau de base pour l'observation. En effet, elles fournissent un point d'entrée unique et défini pour l'observa-

tion des plates-formes pourvues d'un grand nombre d'éléments. Les interfaces spécialisées font partie de la structure de tous les composants EMBera. Les interfaces que nous avons définies pour l'observation sont les interfaces de *description*, de *contrôle* et des *données* (cf. figure 5.2).

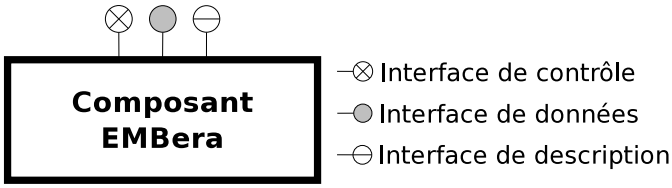

FIGURE 5.2 – Interfaces d'observation d'un composant EMBera

Interface de description Cette interface est une interface d'introspection avec une seule fonction (`ObservableList* getObservables()`) fournissant de l'information sur les évènements qui peuvent être observés. Par exemple, si nous voulons surveiller les threads appartenant à un processus, nous aurons besoin d'instancier un composant EMBera pour ce processus. La fonction `getObservables` retournera, par exemple, parmi d'autres résultats, `< <type=int>, <name="NB_Threads"> >`. Ce dernier résultat permettra de connaître le nom de l'attribut et son type.

Interface de contrôle Il s'agit d'une interface dont le rôle est de choisir les évènements à observer par le composant. Elle fournit des fonctions qui activent ou arrêtent le traitement de l'observation d'un évènement donné.

Dans les sections consacrées à la spécialisation des composants de ce chapitre, nous montrons que les composants possèdent une mémoire tampon pour l'enregistrement temporaire des observations et implémentent des mécanismes pour collecter les données à une fréquence donnée. Ces caractéristiques sont paramétrables à l'aide de l'interface de contrôle.

Les fonctions de l'interface sont les suivantes :

- `void enable(event* observable)` et `void disable(event* observable)` : ces fonctions permettent d'activer ou de désactiver l'observation d'un évènement de l'entité encapsulée. En reprenant l'exemple du processus cité auparavant, si nous avons accès à un évènement du type `event`, appelé `NB_Threads`, il est possible d'appeler la fonction `enable(NB_Threads)` pour surveiller l'évolution des threads appartenant au processus.
- `void setBufferSize(int value)` : cette fonction sert à établir la taille du tampon interne pour l'observation.
- `void setSamplingPeriodicity(int value)` : nous utilisons cette fonction pour fixer la fréquence d'échantillonnage avec laquelle les données brutes sont collectées ou traitées. Le paramètre `value` indique une fréquence en millisecondes.

Interface de données Cette interface est chargée de fournir les données observées par le composant. La fonction principale de cette interface est `Buffer getObservationData()`, laquelle retourne toutes les données observées collectées par le composant.

Dans l'observation que nous souhaitons effectuer, nous estimons que deux de nos cibles sont les programmes parallèles embarqués et les applications organisées avec des composants. Pour la première, nous proposons des fonctions retournant une sous-partie des données disponibles à l'aide de `getObservationData`, et ciblant l'observation des flots d'exécution. Les informations fournies par l'interface de données peuvent être étendues selon les plates-formes et les entités observées. Les fonctions définies sont les suivantes :

- `getExecutionTime` : Donne le temps d'exécution du flot d'exécution observé [1].
- `getMemoryUsed` : Fournit la valeur de la mémoire utilisée par le flot d'exécution sous observation [2].
- `getCreationTime` : Retourne le temps de création des structures du flot d'exécution.

Pour les applications basées sur des composants, nous utilisons, à l'heure actuelle, des informations disponibles dans les structures à composants observées (p. ex. la liste d'interfaces ou l'état du composant).

5.2.3 Les composants basiques

Les composants basiques s'occupent des aspects d'instrumentation et de collecte de données brutes. Quand une entité ou un évènement est choisi pour être observé, nous considérons qu'il doit y avoir un composant basique responsable de l'instrumentation nécessaire du système et de la production des données. L'information à produire est, plus précisément, composée d'une estampille (quand ?), d'un attribut pour identifier le type d'évènement (quoi ?) et des informations additionnelles (quoi exactement ?).

Afin d'illustrer le composant basique, considérons que nous avons besoin de surveiller l'évolution de la mémoire d'un processus. Initialement, nous utilisons un composant basique pour encapsuler le processus, comme nous illustrons dans la figure 5.3. Pour observer le processus, nous utilisons les interfaces de la manière suivante :

1. Nous demandons la liste de caractéristiques observables du processus à l'interface de description, grâce à la fonction `getObservables()`.
2. Nous utilisons l'interface de contrôle pour activer l'observation de la mémoire (`enable(UsedMemory)`) et fixons la périodicité d'échantillonnage (`setSamplingPeriodicity(1000)`).
3. Nous récupérons les données observées en invoquant la fonction `getObservationData()` de l'interface de données.

1. Il s'agit du temps mesuré entre la création et la destruction du flot d'exécution.
2. Comporte la quantité totale de la mémoire utilisée par le flot d'exécution.

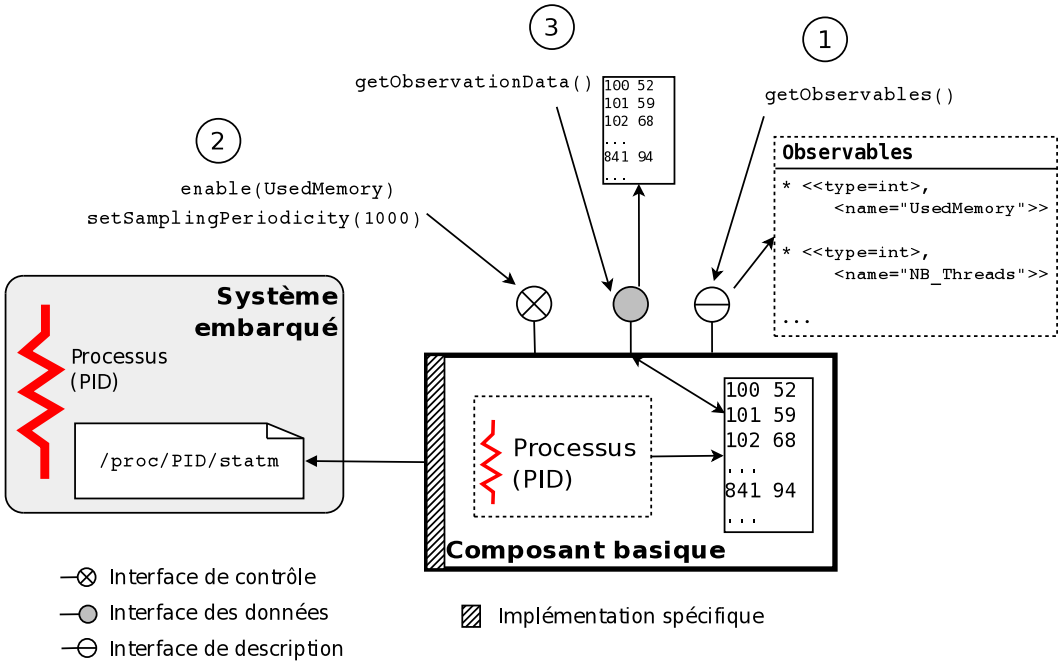

FIGURE 5.3 – Exemple de composant basique : observation de la mémoire utilisée par un processus

Pour obtenir les données brutes, le composant doit implémenter un mécanisme pour demander les valeurs concernant la mémoire au système d'exploitation. Par exemple, le composant basique peut utiliser le fichier du système `/proc/PID/statm`, dans les systèmes Linux.

5.2.4 Le composants de traitement des données

Les composants de traitement de données sont utilisés dans le but de réduire le volume d'informations d'observation à traiter ou à stocker. Pour cela, ils effectuent des traitements sur les données brutes collectées. Nous avons défini deux types de composants, qui sont respectivement les composants de *filtrage* et d'*agrégation*.

Composant de filtrage

Ce composant applique une fonction de filtrage sur les données produites par un ou plusieurs composants basiques. Le filtre à appliquer est défini à travers la fonction `void setFilter(char* observable, Filter* filterOp)`, ajoutée à l'interface de contrôle. Le paramètre `observable` spécifie le nom de la caractéristique à filtrer ; le paramètre `filterOp`, de type `Filter`, est une structure de données référençant la fonction à appliquer ainsi que les paramètres requis, par exemple, la valeur d'un seuil. Les données filtrées résultantes sont stockées dans un tampon géré par le composant et son contenu est accessible à travers l'interface de données.

Dans notre exemple du processus, si nous voulons détecter des valeurs de la mémoire qui dépassent 65 Ko, nous ajoutons un composant de filtrage pour traiter les données produites par le composant basique, tel que présenté dans la figure 5.4.

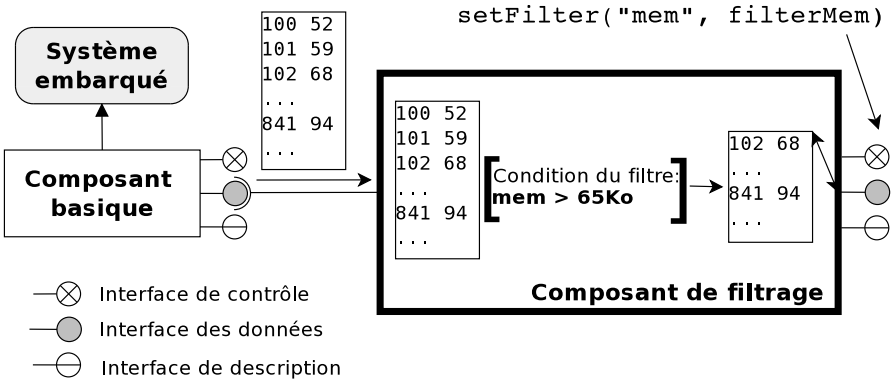

FIGURE 5.4 – Exemple de composant de filtrage

Composant d'agrégation

Ce composant est conçu pour appliquer une fonction d'agrégation sur les données générées par un ou, par plusieurs composants qui peuvent être basiques, de filtrage ou d'agrégation. Pour la définition des fonctions d'agrégation, dans chacun des cas, nous avons ajouté deux fonctions à l'interface de contrôle. Dans les deux cas, les résultats de l'agrégation sont stockées dans le tampon interne du composant.

Si le composant agrège des données pour un seul composant, nous utilisons la fonction `void setAggUnique(char* observable, Aggregator* aggFunc)`. Le paramètre `observable` indique la caractéristique sur laquelle l'agrégation est effectuée ; `aggFunc`, de type `Aggregator`, est une structure de données, qui contient le type d'agrégateur à appliquer ainsi que des paramètres de l'agrégateur, comme le nombre d'observations sur lesquelles l'agrégation est appliquée.

Si le composant agrège les données parvenant de plusieurs composants, il applique la fonction d'agrégation aux données ayant un identificateur commun, par exemple, les observations sur des flux de données correspondant à la même trame dans une séquence vidéo. Pour définir l'agrégation dans ce cas, une deuxième fonction a été ajoutée à l'interface de contrôle. La signature de cette fonction est : `void setAggMultiple (ListAggFunc* list)`. Le paramètre `list`, de type `ListAggFunc` contient une liste des champs et des opérateurs. Par exemple, si nous avons besoin d'agréger trois champs `A`, `B` et `C` avec l'opérateur `AND`, la variable `list` doit être égale à `A AND B AND C`.

Pour illustrer le composant d'agrégation, nous considérons l'exécution d'une application de décodage audio qui produit deux flux en sortie pour deux haut-parleurs. Nous voulons détecter les cas où le son est coupé.

Pour observer cette application, nous utilisons deux composants basiques, un par

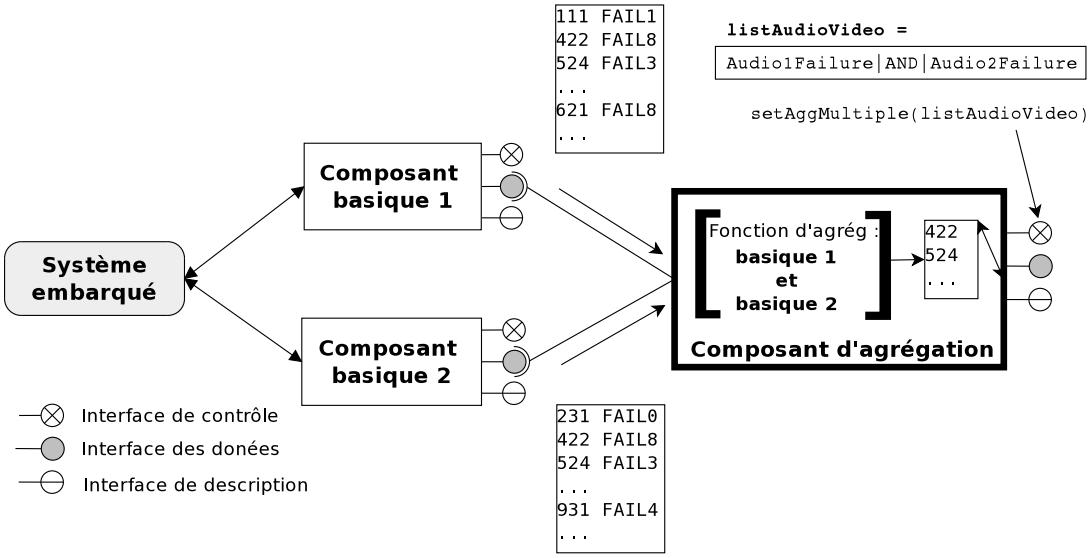

FIGURE 5.5 – Exemple de composant d'agrégation

flux audio, plus un composant d'agrégation, comme il est présenté dans la figure 5.5. Chacun des composants basiques collecte des observations sur les décodages ratés des trames audio pour chacun des flux. Puis, les données sont envoyées vers le composant d'agrégation. Ce dernier applique une fonction d'agrégation $Audio_1 Failure_{Basique1}\ AND\ Audio_2 Failure_{Basique2}$ sur des trames ayant le même identifiant. Le résultat de la fonction remplit le tampon du composant d'agrégation.

5.2.5 Composant de stockage

Ce composant est en charge d'enregistrer les données transmises depuis des composants basiques, de filtrage ou d'agrégation. Les fonctions du composant de stockage comportent principalement le formattage des données et leur sauvegarde sur un support persistant.

Les composants de stockage utilisent deux des trois interfaces définies par le modèle EMBera, à savoir, les interfaces de description et de contrôle. L'interface des données n'est pas utilisée, car les données sont écrites sur un support persistant et ne sont donc pas transmises aux autres composants dans le modèle. L'interface de contrôle implémente la fonction `void writeData()`, qui est chargée de l'écriture des données sur le support persistant et la remise à zéro du tampon. Comme les mécanismes de stockage sont très hétérogènes (des fichiers, port de trace, etc.), nous utilisons une implémentation différente du composant par mécanisme de stockage supporté.

En ce qui concerne le format des données, l'interface fournit, par exemple, la fonction `setFormatType(char* formatName)` pour établir le format à utiliser parmi une liste des formats prédéfinis, comme Pajé [dKdOS00] et OTF [KBB+06], et la fonction `setFieldFormat(char* fieldName, void* fieldType, void* paramsList)`, pour définir

le format d'un type d'évènement. Ces fonctions sont utilisables parce que nous connaissons à priori tous les types de données fournies par les composants.

Pour illustrer le composant de stockage, nous continuons avec l'exemple précédemment. Si nous avons besoin de formatter les données produites par leur composant de filtrage et de les enregistrer dans un fichier de trace, nous utilisons un composant de stockage et le connectons à l'interface de données du composant de filtrage. Cette configuration est présentée dans la figure 5.6.

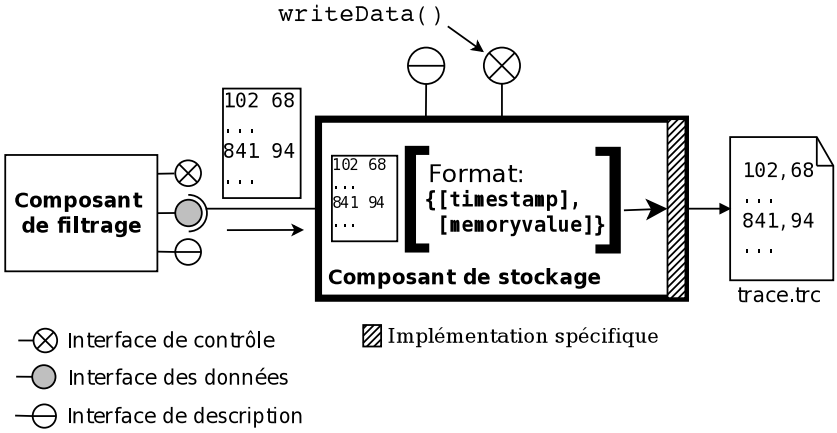

FIGURE 5.6 – Exemple de composant de stockage

5.3 Implémentation du modèle EMBera

Nous avons choisi de faire la mise en œuvre du modèle à composants EMBera en langage C parce qu'il s'agit du standard « de facto »pour les logiciels embarqués, en raison de ses performances, sa portabilité entre les différentes plates-formes et son support aux systèmes hérités [3] [ROC08].

Nous présentons dans ce qui suit les éléments les plus importants de l'implémentation du modèle à composants et de l'observation dans EMBera. Néanmoins, la mise en œuvre doit toujours être complétée avec une partie spécifique à la plate-forme observée. Une implémentation spécifique est requise principalement lors de la collecte des données brutes et lors de l'enregistrement (composants basique et de stockage). Cette implémentation spécifique est illustrée dans les études de cas des chapitres 6, 7 et 8.

5.3.1 Mise en œuvre du modèle à composants

L'implémentation du modèle à composants EMBera peut être considérée comme un support de base pour la définition des traitements d'observation. En effet, la structure de composants peut être vue comme un « code à trous », où les traitements d'observation

3. Terme anglais, *Legacy Systems*

sont définis. Cette structure est accompagnée de primitives de communication entre les composants.

La structure de composants est en particulier utilisée pour l'encapsulation du code métier d'une application observée. En effet, le code à trous des composants EMBera peut être rempli avec le code de l'application.

Un composant EMBera est composé d'une structure de données et de deux flots d'exécution. Le premier flot, que nous appelons *flot principal*, est en charge du fonctionnement du code du composant ainsi que du contrôle du cycle de vie du composant. Le deuxième flot est dédié à la gestion de l'observation. La structure de données contient principalement les références aux flots d'exécution et aux fonctions que ces flots exécutent.

La création de composants se fait avec la fonction `createComponent` qui initialise la structure de données et crée les flots d'exécution correspondants. Nous adaptons l'implémentation de cette fonction générique avec les primitives de création des flots et de synchronisation de la plate-forme observée. Par exemple, nous utilisons `pthread_create` dans Linux et `task_create` dans le RTOS OS21 de STMicroelectronics.

Les interfaces fournies et requises d'un composant sont créées avec la fonction `createInterface`. Le langage C ne fournissant pas le concept d'interface, nous implémentons une interface fournie comme une boîte aux lettres utilisant une structure FIFO, et les interfaces requises étant représentées par un pointeur vers la structure de l'interface fournie. Cette implémentation est utilisable directement sur des plates-formes à mémoire partagée. Si la plate-forme utilise une mémoire organisée d'une autre manière, nous adaptons la structure FIFO et les pointeurs aux structures de la plate-forme, en utilisant par exemple, des objets distribués pour les FIFOs et des ports de communication pour les pointeurs.

Les primitives de communication entre interfaces, `send` et `receive`, sont traduites par des appels à des fonctions de communication spécifiques à la plate-forme.

Nous établissons les liens entre les interfaces requises et fournies des composants à l'aide de la fonction `componentConnect`. Cette fonction consiste à affecter à l'interface requise la référence de l'interface fournie. L'implémentation de cette fonction est également spécifique et dépend principalement du déploiement (si les composants sont déployés sur des processeurs différents) et des mécanismes de communication.

Lorsque les connexions sont établies, la fonction `startComponent` fait démarrer les composants, c'est-à-dire, elle initialise l'exécution des flots d'exécution du composant. Pour attendre la fin de l'exécution des composants, nous fournissons la fonction `componentWait`. Comme nous gérons des flots d'exécution, la mise en œuvre de ces deux fonctions dépend des primitives de création et de destruction des ces flots ainsi que du mécanisme de synchronisation disponible sur la plate-forme.

Le déploiement d'une instance du modèle est fait par l'invocation explicite des fonctions de contrôle. La figure 5.7, présente un exemple de déploiement dans lequel deux composants sont créés (`createComponent`), connectés (`componentConnect`) et démarrés (`startComponent`). Lors de la création, nous donnons un nom et un code à exécuter (une fonction C) par le composant. Avant d'établir une connexion, nous créons les interfaces fournies, puis les requises impliquées dans la connexion (`createInterface`).

```
#include "embera.h"
...
int main ( void *args [ ] ) {
  ...
  Component basic = createComponent ( "basic", basicCompFunction );
  Component storage = createComponent ( "storage", storageCompFunction );
  ...
  interface storageBasicItf = createInterface( storage,
                                               "storageBasic", ... );
  ...
  interface basicDataItf = retrieveInterface ( basic, "basicDataItf", ... );
  ...
  componentConnect ( storage,          /* Composant de stockage */
                     storageBasicItf   /* Interface du composant de stockage */
                     basic,            /* Composant basique */
                     basicDataItf,     /* Interface de donnés du comp. basique */
                   );
  ...
  startComponent ( basic );
  startComponent ( storage );
  ...
}
```

FIGURE 5.7 – Exemple de déploiement des composants EMBera

5.3.2 Mise en œuvre du modèle d'observation

Le modèle d'observation EMBera implémente les interfaces spécialisées, ainsi que les composants d'observation.

Interfaces spécialisées Une interface spécifique est implémentée comme un couple d'interfaces, l'une fournie et l'autre requise. L'interface fournie reçoit des messages demandant de l'information tandis que l'interface requise renvoie l'information demandée.

Le flot principal du composant fournit un support à l'exécution du code du composant, ainsi qu'aux fonctions des interfaces de description et de contrôle. Ce flot sert de support au code métier de l'application, si celui-ci est implémenté à l'aide des composants EMBera. Le deuxième flot exécute les fonctions fournies par l'interface de données. Les données sont, dans la plupart des cas, enregistrées dans un tampon mémoire du composant (cf. figure 5.8).

Composants d'observation Comme l'implémentation d'un composant correspond principalement aux fonctions de ses interfaces fournies, nous avons fait une implémentation générique de ces interfaces. Par la suite, nous illustrons cette implémentation avec des exemples d'extraits de code, à compléter pour chacun des composants d'observation EMBera. L'implémentation précise des fonctions fournies par ces interfaces est spécifique à la plate-forme observée. Par exemple, nous utilisons des fonctions spécifiques du système pour obtenir les estampilles, ainsi que des statistiques sur l'utilisation de la mémoire.

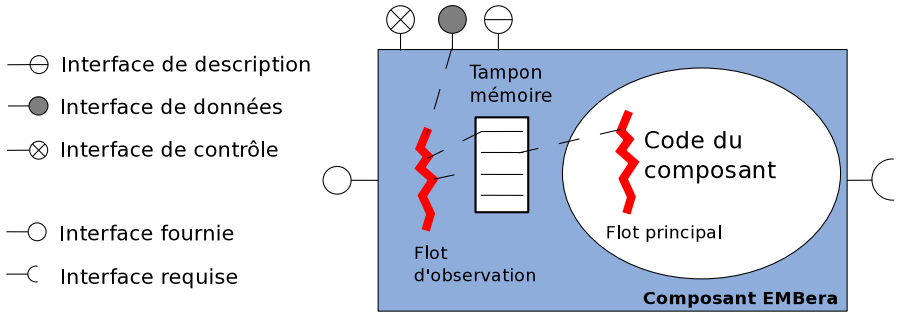

FIGURE 5.8 – Implémentation des composants d'observation EMBera

– *Composant basique* : La figure 5.9 montre un exemple d'un composant basique simple. La fonction qui définit son comportement (`basicCompFunction`) crée une mémoire tampon pour les données brutes et le remplit avec une périodicité donnée. Les données sont accessibles à travers la fonction `getObservationData` de l'interface de données du composant.

```
#include "embera.h"
...
Buffer buffer;                          /* Tampon pour enregistrer les données observées */
Event evt;
...
void *basicCompFunction (void *args) {  /* Fonction principale du composant */
  ...
  buffer = createBuffer ( bufferLength );
  ...                                   /* À compléter avec les évènements à observer */
  while ( ... ) {
    while ( index < bufferLength ) {
      ...
      buffer [ index ] = getRawData ();  /* Enregistre la valeur de l'observation courante
                                            dans le tampon */
      ...
    }
    ...
}
volatile Event getRawData ( ) {
    ...                                 /* À compléter avec l'identification de
                                           l'évènement à observer */
        return evt;
}
Buffer getObservationData ( ) {         /* Fonction de l'interface de données du composant */
  ...
  return buffer;
}
```

FIGURE 5.9 – Extrait de code d'un composant basique

– *Composants de traitement de données* : La figure 5.10 montre un extrait de code d'un composant de filtrage. La fonction principale du composant est la fonction `filterCompFunction`. Elle récupère la référence à l'interface (`itfData`) connectée au composant produisant les données à filtrer. et applique le filtre à chacun des évènements. Puis, elle stocke les évènements acceptés par le filtre dans le tampon

du composant.

```
#include "embera.h"
...
Buffer buffer;                          /* Tampon pour enregistrer les données filtrées */
Event evt;
...
void *filterCompFunction (void *args) {   /* Fonction principale du composant */
  ...
  evt = receive (itfData, ...);
  if ( evalFilter (evt, ... ) ) {    /* Évaluation du filtre */
     bufer[...] = evt;
  }
  ...
}
int evalFilter (evt, ...) {
    int flag = 0;
    ...
    return flag;
}
void setFilter (char* observable, Filter* filterOp) {
    ...
}
```

FIGURE 5.10 – Extrait de code d'un composant de filtrage

– *Composant de stockage* : Le composant de stockage, présenté dans la figure 5.11, est géré par la fonction `storageCompFunction`. Dans cette fonction, le composant remplit un tampon avec des données reçues des autres composants. Quand le nombre de données reçues `bufferSize` est atteint, le composant enregistre le tampon dans une mémoire persistante à l'aide de la fonction `writeData`.

5.4 Comment instancier le modèle d'observation EMBera ?

Pour utiliser le modèle EMBera, nous proposons de suivre les étapes suivantes :

1. ***Définition des objectifs d'observation et identification des entités*** : Il est important de bien définir la liste d'objectifs d'observation en premier lieu. Chacun des objectifs doit se baser sur l'une des trois actions suivantes :
   - la collecte spécifique d'information sur une entité donnée,
   - le pré-traitement des données et leur enregistrement,
   - l'établissement des paramètres initiaux pour configurer l'observation.

2. ***Identification des composants EMBera*** : Le deuxième étape consiste à identifier les composants EMBera nécessaires à l'observation. La liste d'objectifs et les entités identifiées dans l'étape 1, nous permettent de mieux choisir les composants à utiliser :

```
#include "embera.h"
...
Buffer recvBuffer;                      /* Tampon de réception des données observées */
int bufferSize;                         /* Taille du tampon de réception */
int availableBuffer;                    /* Espace disponible dans le tampon */
...
void *storageCompFunction (void *args) { /* Fonction principale du composant */
  ...
  availableBuffer = bufferSize;
  Interface itf = retrieveInterface (...);
  ...
  while  ( ... ) {
    ...
    recvBuffer [bufferSize - availableBuffer] = receive (itf, ...);
    availableBuffer -- ;
    if ( availableBuffer == 0 )  {
      ...
      writeData ( );                    /* Écriture de données dur le mécanisme */
      availableBuffer =  bufferSize;
      ...
    }
    ...
  }
}
void writeData( ) {
    ...                                 /* À compléter avec le code spécifique au
                                           mécanisme de stockage */
}
```

FIGURE 5.11 – Extrait de code d'un composant de stockage

- Les entités et les évènements indiquent les composants basiques à utiliser.
- Les objectifs comportant des traitements correspondent à des composants de filtrage et/ou agrégation.
- Les besoins d'observation correspondant à un enregistrement de données nécessitent l'utilisation d'un composant de stockage.
- Enfin, les objectifs liés à la configuration de l'observation déterminent les paramètres initiaux des composants.

3. ***Préparation du système pour l'observation*** : La préparation de la cible à observer est faite soit en recompilant le code applicatif avec des options spécifiques et en ajoutant des extraits de code EMBera, soit en accédant directement aux mécanismes existants de production de données (p. ex. des traceurs comme `DTrace` [CSL04] ou `systemtap` [sys]). La figure 5.12 montre un extrait de code EMBera qui peut être ajouté à la compilation de l'application observée afin de produire des données de profilage. Dans cet exemple, une fois compilé l'extrait de code produit à l'exécution un évènement à chaque fois qu'une fonction utilisateur définie est exécutée.

4. ***Spécialisation du modèle d'observation pour la plate-forme*** :
   Cette étape consiste à compléter le code à trous des composants d'observation

```
#include "embera.h"
...
void profile_function (...) {
  ...
  if (getFuncName(functionAddress) == "functionName") { /* Interception de l'appel
                                                           à la fonction */
    ...
    event evt = createEvent (...);                      /* Génération d'un évènement lors
                                                           de l'interception */
    sendEvent (evt, ...);                               /* Envoie de l'évènement vers
                                                           le composant basique */
    ...
  }
}
```

FIGURE 5.12 – Extrait de code d'une fonction de profilage

EMBera. En effet, il faut ajouter le code nécessaire à ces fonctions. Un exemple est la récupération des données produites pendant l'exécution, un deuxième est l'écriture des données sur un dispositif de stockage.

5. ***Instanciation et déploiement des composants EMBera*** : La dernière étape est l'instantiation des composants pour leur déploiement. Cette définition peut être faite soit en créant un programme, soit à l'aide d'un fichier de description.
   Dans le premier cas (cf. figure 5.7), il faut importer les bibliothèques d'EMBera et instancier, connecter et contrôler le cycle de vie des composants dans le programme. Autrement, il est possible d'utiliser un fichier de description XML pour définir les composants, leurs connexions et leur déploiement. La figure 5.13 illustre la définition et le contenu d'un fichier de description d'une instance d'EMBera ayant un composant basique, un composant de filtrage et un composant de stockage. Nous clarifions que nous n'avons pas implémenté un parseur pour des fichiers de configuration spécifiques à EMBera. Cependant, si nous utilisons une implantation à composants existante pour implémenter le modèle d'observation, comme Cecilia (cf. section 3.3.2.1), nous avons la possibilité de nous servir du parseur de ce modèle.

   Le choix entre ces deux options dépend de l'approche de programmation utilisée par l'application à observer. Par exemple, pour une application structurée seulement avec des fonctions, nous utilisons un programme pour instancier le modèle EMBera, alors que pour une application structurée par le biais de composants, nous utilisons un fichier de configuration.

   L'observation effectuée par EMBera suit le cycle de vie de l'application. C'est pourquoi le démarrage de l'exécution de l'application est fait en même temps que celui de l'instance d'EMBera. L'observation des autres niveaux d'abstraction est effectuée pendant l'exécution de l'application (p. ex. les interactions au niveau intergiciel).

```
        ...
        <!ATTLIST loader name CDATA #IMPLIED>
        <!ELEMENT embera  (component*)>
        ...
(a)     <!ATTLIST component type CDATA #IMPLIED>
        <!ATTLIST component name CDATA #IMPLIED>
        <!ATTLIST component processor CDATA #IMPLIED>
        <!ATTLIST component out CDATA #IMPLIED>
        ...
-------------------------------------------------------------------------------
        ...
        <embera>
         ...
         <component type="basic" name="basic0" out="filter0" processor="0"/>
(b)      <component type="filter" name="filter0" out="storage0" processor="0"/>
         <component type="storage" name="storage0" processor="1"/>
         ...
        </embera>
        ...
```

FIGURE 5.13 – Description d'une instance d'EMBera en format XML

5.5 Synthèse

Nous avons décrit dans ce chapitre un modèle à base de composants logiciels que nous avons défini et implémenté pour servir comme infrastructure de base pour l'observation. Dans ce modèle, nous reprenons des concepts du modèle Fractal tels que l'encapsulation, les interfaces et l'organisation des hiérarchies.

Nous avons mis l'accent sur l'adaptation du modèle d'observation à base de composants aux systèmes embarqués. Pour cibler ces derniers, le modèle tient compte principalement de la portabilité entre plates-formes, ainsi que du contrôle des mécanismes de collecte de données pour réduire l'intrusivité de l'observation. Le modèle repose sur des composants spécialisés ainsi que sur un ensemble d'interfaces consacrées à l'observation

Les composants spécialisés permettent l'encapsulation des éléments du système à observer, afin d'identifier leurs caractéristiques et les analyser séparément. La spécialisation des composants est faite en définissant des types de composants pour adresser les différentes étapes de l'observation.

Les interfaces spécialisées fournissent un point d'entrée unique pour découvrir, contrôler et accéder aux observations obtenues par les composants. À partir de ces interfaces nous pouvons construire des hiérarchies complexes pour le traitement des données. Le résultat de l'utilisation de telles hiérarchies sert à exposer succinctement le comportement de tout le système observé, tout en permettant d'étudier des données en détail si nécessaire.

Le chapitre contient également une suite d'étapes que nous considérons nécessaires et suffisantes pour la bonne utilisation de notre modèle d'observation à base de composants.

Notre approche est bien adaptée aux particularités des systèmes embarqués en permettant de configurer et de contrôler séparément les différentes étapes liées à l'observation. En effet, nous pouvons configurer l'exécution de chacune des étapes de manière indé-

pendante afin de réduire l'impact sur le système. De même, nous pouvons déployer les composants pour observer des ressources hétérogènes de la plate-forme étudiée.

Afin de valider notre approche, nous proposons d'appliquer le modèle EMBera à différents contextes d'observation. Pour cela, nous utilisons trois études de cas effectuées sur différentes configurations matérielles et logicielles. Nous présentons dans les chapitres qui suivent ces trois études : une application de décodage MJPEG (chapitre 6), une application multimédia industrielle (chapitre 7) et un décodeur du format SVC (chapitre 8).

Troisième partie

Expérimentation

Chapitre 6

Étude de cas : décodeur MJPEG

Dans ce chapitre nous présentons la première des trois études de cas proposées pour valider notre modèle d'observation EMBera. Cette étude de cas consiste en l'observation d'une application de décodage vidéo sous format MJPEG [iso07]. Nous avons adapté l'application pour l'utiliser sur une plate-forme embarquée construite par STMicroelectronics.

Notre choix s'est porté sur une application de décodage pour deux raisons. Premièrement, les décodeurs font partie des applications présentes dans des dispositifs embarqués grand public, comme les set-top boxes et les *smartphones* [Mar06]. Deuxièmement, l'application MJPEG est un décodeur facile à analyser et ses sources sont disponibles.

Dans la suite, nous décrivons les aspects généraux de l'architecture matérielle et logicielle utilisée. Ensuite, nous montrons les aspects les plus représentatifs de la mise en œuvre du modèle à composants EMBera et de l'application MJPEG sur la plate-forme embarquée. Puis nous nous concentrons sur l'utilisation du modèle d'observation EMBera pour l'application MJPEG, ainsi que sur l'analyse des observations obtenues.

6.1 Architecture de la plate-forme embarquée

Les expériences ont été effectuées sur la plate-forme *STi7200*[1], une plate-forme embarquée multiprocesseur construite par STMicroelectronics. La configuration matérielle et logicielle de la plate-forme est brièvement présentée, ci-après.

Configuration matérielle

La plate-forme contient une unité centrale de calcul générique RISC[2] *ST40* cadencée à 450 Mhz et quatre processeurs *ST231* à 400 Mhz (cf. figure 6.1). Le processeur central *ST40* a accès à toute la mémoire de la puce qui est une mémoire SDRAM de 2 Go. Chaque

1. Plate-forme de décodage multimédia de haute définition [STM06].
2. Acronyme anglais : Reduced Instruction Set Computer

processeur *ST231* a accès à un bloc de cette mémoire qui représente sa mémoire locale. La communication entre processeurs se fait en utilisant un 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*.

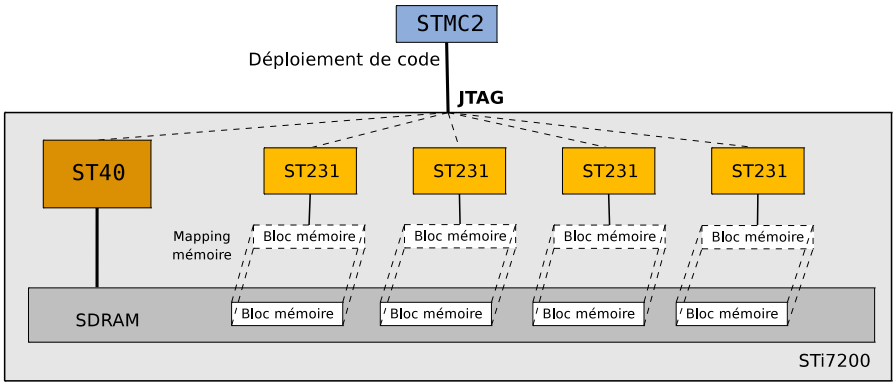

FIGURE 6.1 – Plate-forme STi7200

La plate-forme *STi7200* dispose d'un port JTAG à travers lequel il est possible de déployer du code binaire directement sur les différents processeurs. Le port JTAG est utilisé à l'aide d'un boîtier spécifique, appelé *STMC2*. Enfin, la plate-forme dispose des ports de sortie vidéo ainsi que des ports USB pour l'accès à des dispositifs de stockage (p. ex. une mémoire flash).

Configuration de la pile logicielle

Le décodeur MJPEG utilise une pile logicielle spécifique pour l'exploitation de la plate-forme matérielle. Cette pile logicielle est composée par l'application de décodage, un intergiciel, et un système d'exploitation temps réel.

La programmation sur *STi7200* est faite en utilisant une implémentation optimisée du standard ANSI C. Cette implémentation fait partie d'un ensemble d'outils STMicroelectronics comprenant des compilateurs, 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.

L'application de décodage MJPEG

Il s'agit d'une application de décodage de 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 [Huf52], un ré-ordonnancement de pixels et une transforma-

3. L'application a été mise en œuvre dans [APcDG05], dans le cadre du développement d'une plate-forme de simulation *cycle-accurate*

tion discrète inverse de cosinus (IDCT). À la fin, tous les blocs sont réordonnés afin de reconstituer les images originales et les envoyer vers une sortie pour leur affichage.

Aucune version embarquée de l'application n'étant disponible, nous l'avons porté sur la plate-forme *STi7200*. Nous avons encapsulé les différents traitements MJPEG dans des composants en gardant la structure de l'application. Nous avons défini un composant `Fetch`, un composant `Reorder` et plusieurs composants `IDCT`. `Fetch` s'occupe du découpage des images en blocs et de l'application de l'algorithme de Huffman. Les composants `IDCT` effectuent des calculs d'IDCT en parallèle et `Reorder` réordonne et reconstitue les images. L'application qui en résulte est schématisée dans la figure 6.2.

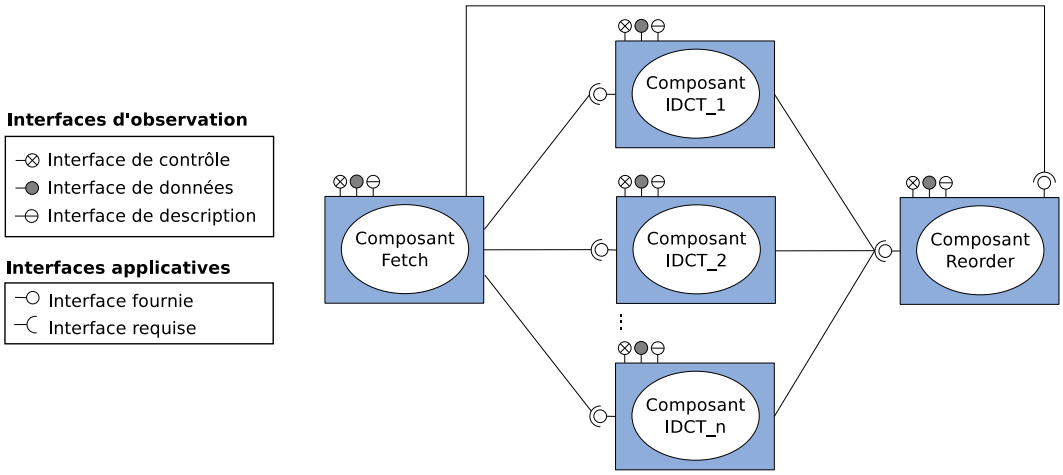

FIGURE 6.2 – Application de décodage MJPEG avec des composants EMBera

Intergiciel multimédia STMicroelectronics

Cet intergiciel est une couche qui fournit des interfaces de haut niveau pour l'accès aux fonctionnalités de la plate-forme multimédia ainsi que pour la communication entre les différents processeurs. Pour l'exploitation de ces derniers, les applications sont programmées à l'aide des flots d'exécution déployés sur les processeurs. Si les flots d'exécution s'exécutent sur un même processeur, ils communiquent en accédant au bloc mémoire de ce processeur. Les flots d'exécution, s'exécutant sur des processeurs différents, communiquent à l'aide d'un mécanisme de communication spécifique appelé `EMBX`. La communication est effectuée 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, un flot d'exécution 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 est 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`, nécessite l'établissement d'un canal de communication entre les processeurs concernés et la création de *ports* de communication. Si un flot d'exécution s'exécutant sur le processeur *ST40* doit communiquer avec un autre flot sur un *ST231*, il doit y avoir un port par processeur, qui va « acheminer » la communication entre les flots d'exécution.

Système d'exploitation temps réel STMicroelectronics

Les processeurs de la plate-forme *STi7200* exécutent un système d'exploitation multi-tâches temps réel [STM07b] développé par STMicroelectronics. Le RTOS fournit des modules de gestion des flots d'exécution (tâches) et de la mémoire. Le système fournit également tous les mécanismes nécessaires à la gestion des interruptions, des exceptions et de la synchronisation.

Les tâches du RTOS peuvent être considérées de manière analogue aux 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 RTOS est lui-même géré comme une tâche. Pour créer une nouvelle tâche du RTOS, il est nécessaire de spécifier la région de la mémoire (appelée *partition*) qui va être utilisée par la tâche ainsi que sa priorité parmi l'ensemble de tâches.

6.2 Mise en œuvre du modèle EMBera

Dans la suite, nous détaillons l'implémentation du modèle à composants EMBera sur la plate-forme *STi7200*. Puis nous montrons comment nous avons mis en œuvre l'application MJPEG grâce à ces composants.

6.2.1 Implémentation des composants EMBera

La structure d'un composant EMBera est implémentée comme une structure de données C et ses flots d'exécution comme des tâches du RTOS. Les fonctions du modèle EMBera (cf. section 5.3.1) sont implémentées comme décrit ci-après.

La fonction `createComponent` crée les tâches du composant à l'aide de la fonction du RTOS `task_create`. Après leur création, les tâches sont mises en attente à l'aide de la fonction du RTOS `task_suspend`. En effet, les composants ne doivent pas être lancés avant que leurs interfaces soient créées et connectées.

La fonction `createInterface` dépend du type d'interface créée. Pour une interface fournie, la fonction crée un objet distribué (cf. section 6.1), 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`.

Si les composants s'exécutent sur deux processeurs différents, la fonction `componentConnect` affecte à l'interface requise la référence de l'objet distribué représentant l'interface fournie. Dans ce cas, la récupération de la référence nécessite que le composant qui détient l'interface fournie fasse 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 6.1). Dans `componentConnect` s'effectuent l'attente de cette référence, la récupération et l'affectation à l'interface requise. Si les deux composants s'exécutent sur le même processeur, la récupération de la référence est triviale parce que elle est faite sur la mémoire accessible au processeur et la connexion n'a pas besoin d'objet distribué.

Étant donné que lors de la création du composant les tâches ont été mises en attente, la fonction `startComponent` débloque l'exécution des tâches. À cet effet, `startComponent` utilise la fonction du RTOS `task_resume`. La fonction `componentWait` utilise la primitive du RTOS `task_wait` dont le but est d'attendre la fin de la tâche principale du composant et de détruire les structures utilisées par le composant.

6.2.2 Implémentation de l'application MJPEG

Les modules de l'application MJPEG, introduits en section 6.1, ont été mis en œuvre à travers de composants EMBera. Nous avons réutilisé le code original de l'application sauf en ce qui concerne la communication entre les modules.

Pour des raisons liées au matériel, et explicitées dans la section suivante, nous avons utilisé trois processeurs sur les cinq fournis par la plate-forme. Nous avons donc créé trois composants pour les modules de l'application MJPEG : un composant `Fetch-Reorder` et deux composants `IDCT` (cf. figure 6.3). Les composants EMBera créés reprennent la structuration et les fonctions montrées dans la figure 6.2. Le composant `Fetch-Reorder` est une fusion des composants `Fetch` et `Reorder` montrés précédemment. Il est exécuté sur le processeur *ST40*, car il est plus performant pour l'exécution des opérations d'entrée et de sortie. Les composants `IDCT` sont déployés sur 2 *ST231* puisqu'ils sont conçus pour exécuter des fonctions de calcul intensif.

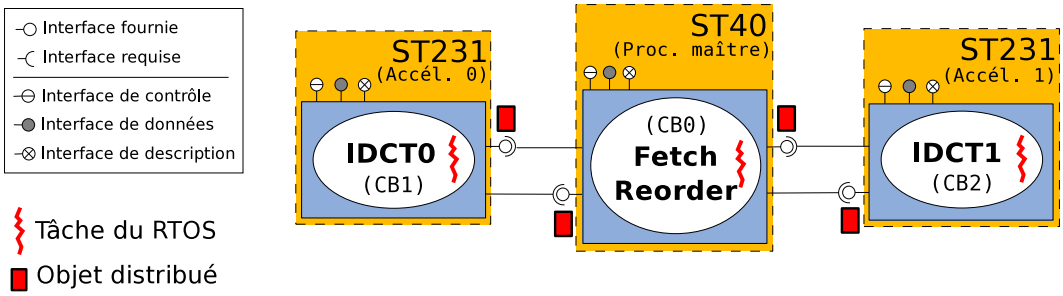

FIGURE 6.3 – Déploiement de l'application de MJPEG sur la plate-forme *STi7200*

La figure 6.4 présente un extrait simplifié de la fonction exécutée par un des composants `IDCT` de l'application MJPEG.

```
#include "embera.h"
#include "rtossupport.h"
void *idct_function (void *args) {
   ...
   msgBuffer = receive(provIDCT0Fetch);         /* IDCT0 reçoit des données en entrée */
   ...
   while (strcmp(msgBuffer->msgType, "EOF") {   /* Fin de la vérification des messages */
      ...
      computeIDCT(...);                         /* Calcul d'IDCT */
      ...
   }
   send (msgResult, reqIDCT0Fetch);             /* Envoi du résultât d'IDCT0 vers le composant
                                                   Fetch-Reorder */
}
```

FIGURE 6.4 – Extrait du code exécuté par le composant IDCT0

6.2.3 Déploiement de l'application MJPEG

Afin de déployer l'application MJPEG sur la plate-forme *STi7200*, nous avons généré un code exécutable différent pour chacun des processeurs. Chaque code binaire a été indépendamment chargé sur son processeur cible. Pour les expériences présentées dans la section 6.4, le chargement a été effectué manuellement. Chaque exécutable contient une fonction `main` permettant d'établir l'architecture en termes de composants déployée sur ce processeur.

```
#include "embera.h"
#include "rtossupport.h"
extern void *idct_function (void *args);
int main () {
   rtos_start();
   rtosPreDeployment("st231video0","st40");
   ...
   component *IDCT0 = createComponent (..., "IDCT0", idct_function);
   interface *provIDCT0Fetch = createInterface(IDCT0, "provIDCT0Fetch", "provided");
   interface *reqIDCT0Reorder = createInterface(IDCT0, "reqIDCT0Reorder", "required");
   ...
   componentConnect(reqIDCT0Reorder, provFetchIDCT0);
   startComponent(IDCT0);
   ...
   return(0);
}
```

FIGURE 6.5 – Extrait du code `main` exécuté sur le processeur ST231

Regardons le programme de déploiement d'un des composants `IDCT` (cf. figure 6.5). Dans ce programme, nous initialisons tout d'abord le noyau du RTOS (`rtos_start`) et établissons les canaux de communication entre les processeurs (`rtosPreDeployment`). La fonction `rtosPreDeployment` crée deux ports portant le même nom, l'un sur le processeur *ST40* et l'autre sur le processeur *ST231*. Ces ports seront utilisés pour toutes les communications effectuées à travers les primitives `EMBX_Send` et `EMBX_Receive` entre des tâches s'exécutant sur ces deux processeurs. En effet, l'établissement des ports est fortement lié à l'architecture de l'application et nous pouvons le considérer comme un pré-déploiement.

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 se termine par le lancement du composant (`startComponent`) et la mise en attente de sa terminaison (`componentWait`).

Pour charger le code sur les processeurs, nous avons utilisé la plate-forme en mode d'opération normal, en utilisant tous les processeurs pour les traitements exécutés (1 composant `Fetch-Reorder` sur le *ST40* et 4 `IDCT` sur les *ST231*). Cependant, ce mode nous permet de n'observer que des données du processeur maître. Pour cette raison, nous nous sommes servi du boîtier *STMC2*, de l'interface JTAG (cf. figure 6.1) et d'un outil de débogage. Cependant, 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.

À ce stade, nous avons les fonctionnalités de l'application MJPEG encapsulées dans des composants EMBera. Par la suite nous présentons l'application du modèle d'observation EMBera au décodeur MJPEG.

6.3 Application du modèle d'observation EMBera

Nous présentons dans ce qui suit la manière dont le modèle EMBera est appliqué pour l'observation de l'application MJPEG. Nous suivons à cet effet les étapes introduites dans la section 5.4.

6.3.1 Définition des objectifs d'observation et identification des entités

Vu que nous avons porté le décodeur MJPEG sur la plate-forme *STi7200*, nous avons voulu évaluer la qualité de décodage et l'optimalité de l'utilisation des ressources. Pour cela, nous nous sommes intéressés aux métriques suivantes :

1. *Qualité de la vidéo* :
   (a) Nombre total d'images décodées.
   (b) Quantité d'appels aux fonctions de communication entre les fonctionnalités majeures de MJPEG.
   (c) Temps de communication et taille des messages.
   (d) Temps d'exécution de toute l'application.
2. *Utilisation des ressources de la plate-forme* :
   (a) Temps d'exécution de chacune des tâches du RTOS utilisées par l'application.

(b) Durée de création d'une tâche.

(c) Évolution de l'utilisation mémoire du composant `Fetch` inférieure à 2 Mo.

(d) Produire un historique de l'utilisation de la mémoire, relevée toutes les secondes.

À partir de ces objectifs, nous identifions les entités et les évènements à observer. Les entités identifiées sont les flots d'exécution de l'application (des tâches du RTOS). Nous observons trois flots d'exécution, un flot par composant de l'application.

6.3.2 Identification des composants EMBera

Après l'analyse des objectifs de cette étude de cas, nous proposons l'instance du modèle d'observation suivante (cf. figure 6.6) :

- *Trois composants basiques* : Chaque composant basique est associé à un flot d'exécution de l'application MJPEG. Toutefois, comme l'application utilise déjà des composants EMBera, nous n'avons pas besoin d'en créer d'autres. En effet, les composants de l'application servent à encapsuler les flots d'exécution que nous voulons observer. Comme les fonctionnalités des composants basiques sont déjà présentes dans ces composants, il suffit d'activer l'observation à travers leurs interfaces de contrôle.
- *Un composant de filtrage* : le rôle de ce composant est de détecter quand un composant Fetch-Reorder occupe plus de 2 Mo de mémoire.
- *Un composant de stockage* : ce composant est utilisé pour stocker les données collectées.

6.3.3 Préparation du système pour l'observation

Le portage du décodeur MJPEG sur la plate-forme embarquée a été fait en utilisant le modèle EMBera. Par conséquent, il n'a pas fallu modifier l'application pour l'observer. En effet, la structure de composants implémente les mécanismes requis pour collecter des données du décodeur et de l'intergiciel. D'autre part, vu que nous nous servons d'une interface définie par le RTOS, qui permet de récupérer des mesures de performance de la plate-forme, nous n'avons pas modifié le système pour l'observation.

6.3.4 Spécialisation du modèle pour la plate-forme

Nous avons spécialisé l'implémentation de l'interface de données afin d'utiliser des fonctions fournies par le RTOS. La plus importante de ces fonctions est `task_status` qui donne des informations 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

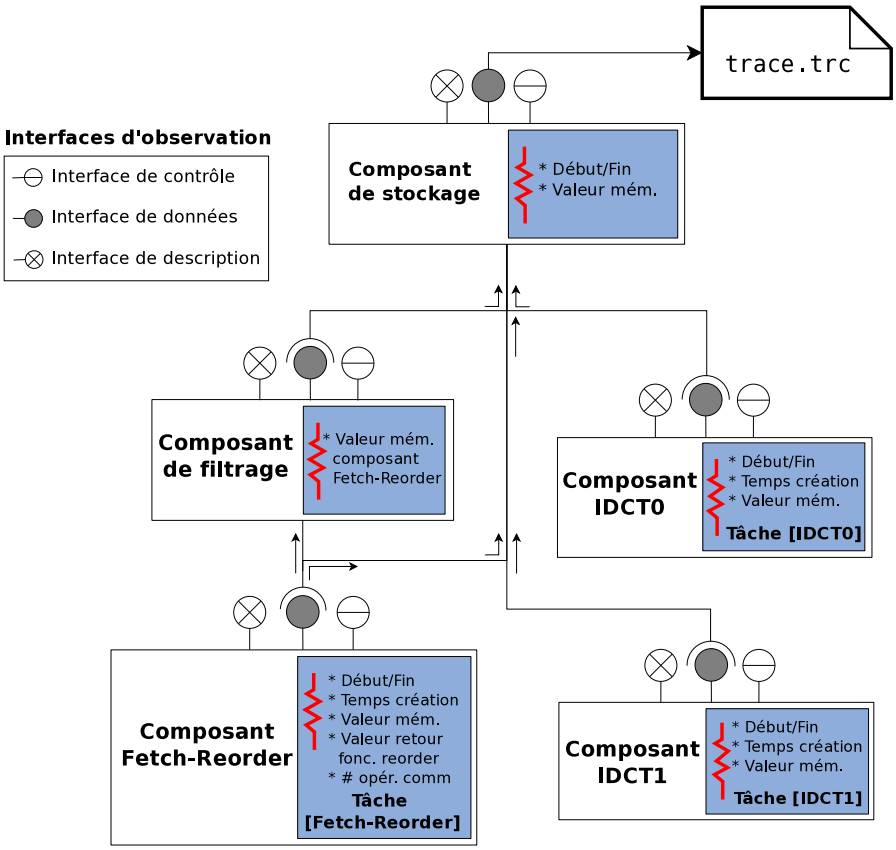

FIGURE 6.6 – Composants EMBera utilisés pour l'observation de l'application MJPEG

des autres tâches, nous pouvons récupérer les informations sur la tâche principale d'un composant depuis la tâche d'observation.

La fonction `getObservationData()` retourne principalement les valeurs disponibles à travers `task_status`. Nous avons organisé la récupération des données disponibles à travers une liste de sous-foncions, présentées ci-après :

- `getExecutionTime` : Nous définissons le temps d'exécution d'un composant comme le temps d'exécution de la tâche principale de ce composant.
- `getMemoryUsed` : La mémoire utilisée par un composant est caractérisée par la mémoire utilisée par sa tâche principale, la taille de la structure de données de composant étant négligeable.
- `getCreationTime` : Le temps de création d'un composant correspond à la création de sa tâche principale et l'allocation de sa structure de données. La mesure du temps de création est faite avec la fonction du RTOS `time_now`.
- `getSendAverageTime` et `getReceiveAverageTime` : Ces fonctions ont été rajoutées pour mesurer le temps moyen de communication entre tâches. Cette fonctionnalité est activée/désactivée explicitement à la compilation d'une application EMBera.

6.3.5 Déploiement des composants EMBera

Nous instancions les composants du modèle d'observation dans le même programme C utilisé pour le lancement de l'application MJPEG sur chacun des processeurs. Pour cela, nous ajoutons des primitives pour la création des composants et des interfaces ainsi que pour le contrôle du cycle de vie. Un exemple du code ajouté pour l'observation est présenté dans la figure 6.7.

```
#include "embera.h"
#include "rtossupport.h"
int main () {
   ...
   interface *reqFetchFilter = createInterface(FetchReorder, "reqFetchFilter", "required");
   component *filterComponent = createComponent (..., "filterComponent", filterFunction);
   interface *provFilterFetch = createInterface(filterComponent, "provFilterFetch", "provided");
   componentConnect(reqFetchFilter, provFilterFetch);

   interface *reqFilterStor = createInterface(filterComponent, "reqFilterStor", "required");
   component *storageComponent = createComponent (..., "storageComponent", filterFunction);
   interface *provStorage = createInterface(storageComponent, "provStorage", "provided");
   componentConnect(reqFilterStor, provStorage);
   ...
   startComponent(filterComponent);
   startComponent(storageComponent);
   ...
}
```

FIGURE 6.7 – Code de déploiement des composants d'observation sur le processeur *ST40*

Concernant le déploiement des composants EMBera, nous avons utilisé le processeur maître de la plate-forme pour exécuter les composants de filtrage (`filterComponent`) et de stockage (`storageComponent`). La figure 6.8 montre le déploiement effectué.

Nous avons fait ce choix, car le processeur *ST40* à un accès direct aux données observées par `Fetch-Reorder`. Le composant `storageComponent` a également été déployé sur le processeur maître car il offre un meilleur service pour la gestion et l'exécution de plusieurs tâches dans le système. Les composants `IDCT0` et `IDCT1` sont connectés au composant `storageComponent` à travers des connexions distantes, à l'aide d'objets distribués. Étant donné que la plate-forme n'a pas de système de fichiers, nous nous servons du composant `storageComponent` pour accéder au port JTAG. Les données transmises à travers ce port sont directement transférées sur le système de fichiers de la machine de développement.

6.4 Observations effectuées et problèmes détectés

Nous organisons les données collectées par niveaux d'observation le RTOS, l'intergiciel de communication et l'application MJPEG. Cette organisation permet d'analyser séparément les données et de trouver des éventuelles corrélations entre les différents niveaux d'abstraction. Nous montrons que l'observation aide dans la détection de problèmes de performance et dans une meilleure compréhension de l'application MJPEG.

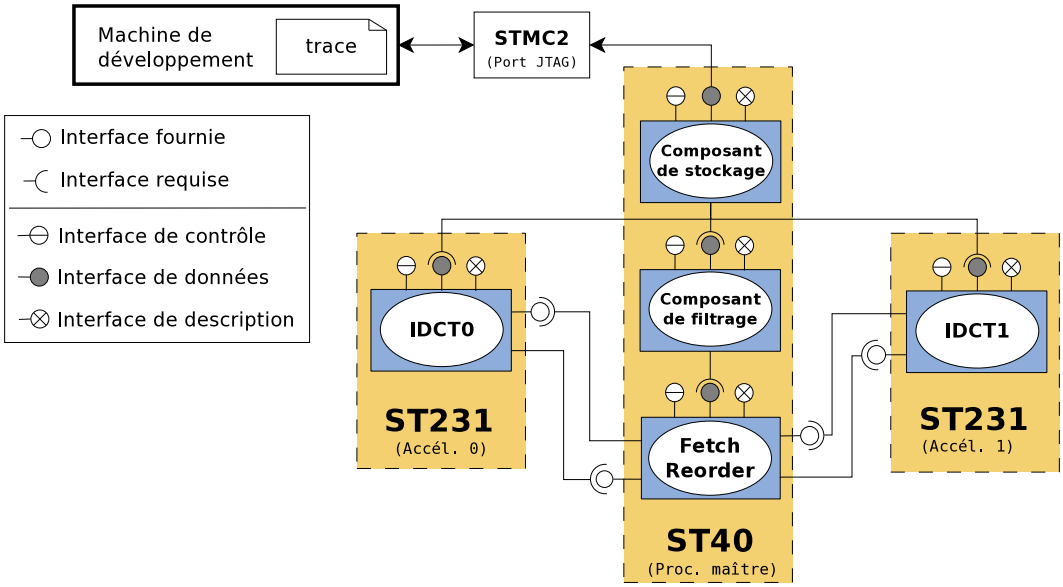

FIGURE 6.8 – Déploiement des composants d'observation EMBera

Pour nos expérimentations, nous avons observé l'exécution de l'application MJPEG, sur un cas de décodage d'un fichier de 577 *images* JPEG (taille du fichier 2.8 *Mo*). Les expérimentations effectuées ont été exécutées 10 fois, pour chaque niveau observé. Nous considérons ce nombre d'exécutions suffisant car les exécutions s'effectuent sur une plate-forme logicielle, qui respecte des contraintes temps réel. Le respect de ces contraintes permet aux exécutions d'avoir un temps d'exécution quasi identique (différences de moins de 1%).

6.4.1 Observations au niveau système

La table 6.1 présente les mesures concernant les temps d'exécution des composants de l'application. Nous pouvons constater 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 le traitement d'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	Exéc. non observée (s)
`Fetch-Reorder`	83.30
`IDCT0`	6.42
`IDCT1`	7.19

TABLE 6.1 – Temps d'exécution des composants du décodeur MJPEG

La grande différence en temps d'exécution entre ces composants nous suggère une optimisation : la parallélisation de `Fetch-Reorder`, qui permettrait la réduction du temps total d'exécution du décodeur. Par exemple, les fonctions de `Fetch` et `Reorder` pourraient être exécutées par des tâches indépendantes, même si elles sont déployées sur le même processeur. Par ailleurs, d'autres tâches `IDCT` pourraient être exécutées sur les processeurs *ST231* pour mieux exploiter sa capacité.

L'évolution de la consommation mémoire est également observée. Nous prenons toutes les secondes des mesures de la mémoire utilisée par les composants `Fetch-Reorder` et `IDCT0`. Nous observons que la mémoire utilisée par `Fetch-Reorder` (cf. figure 6.9 (a)) augmente rapidement dans un premier temps, puis ralentit, tout en continuant à croître. Nous pouvons faire correspondre cette empreinte mémoire au comportement du composant. Au début de son exécution, le composant initialise les données et ensuite effectue un calcul itératif pendant lequel il ne libère pas de mémoire.

L'évolution de la taille mémoire de `Fetch-Reorder` nous montre un léger problème de fuite mémoire, qui peut provoquer un remplissage inutile de la mémoire de la plateforme. Même si dans les expériences que nous avons effectuées nous n'avons pas épuisé la mémoire, le cas peut se présenter lors du décodage des séquences vidéo plus longues. Grâce à cette observation, nous avons corrigé le problème en libérant la mémoire quand tout les blocs d'une image sont transmises vers les composants d'`IDCT`.

L'observation de la mémoire peut permettre de mieux dimensionner la taille nécessaire pour une application. Dans les systèmes sur puce, le dimensionnement précis des ressources résulte en réductions significatives du coût de production de ces systèmes.

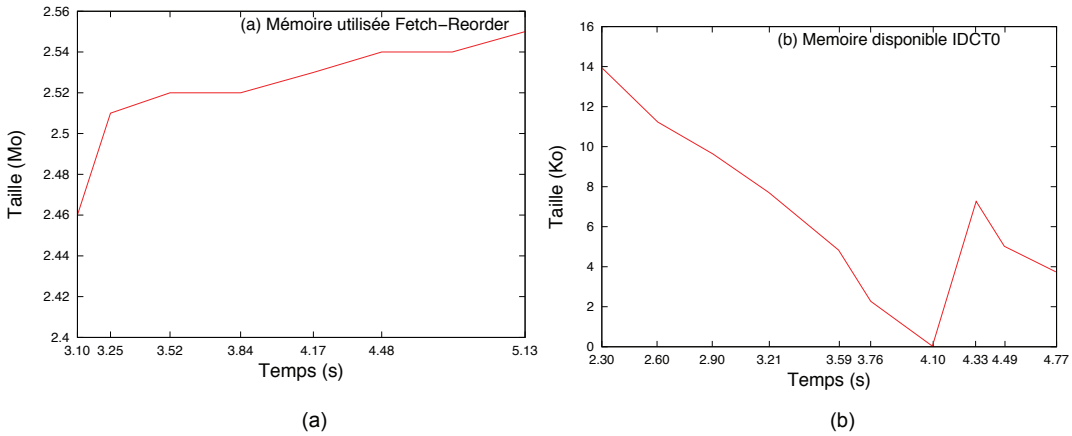

FIGURE 6.9 – Évolution de la mémoire des composants

L'utilisation de la mémoire par le composant `IDCT0` est présentée différemment. La figure 6.9 (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, le RTOS élargit dynamiquement la partition mémoire.

Cette information peut nous permettre, par exemple, de régler la taille de la portion mémoire additionnelle afin d'élargir moins souvent la mémoire, car des augmentations fréquentes peuvent induire un coût énergétique considérable.

6.4.2 Observations au niveau intergiciel

Nous avons mesuré le temps de création des composants. Nous avons constaté que ce temps est principalement consacré à la création des tâches. Pour le composant `Fetch-Reorder`, le temps de création est de 57 ms, séparé en 33 ms pour la création de la tâche principale (58%) et 23 ms pour la création de la tâche d'observation (42%). Pour les composants `IDCT`, le temps de création est plus long (81 ms) : 51 ms pour la tâche principale (63%) et 30 ms 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 de contrôle de périphériques, il est plus performant dans la création et ordonnancement des tâches que les accélérateurs. Cette observation nous indique que nous devons limiter le nombre de tâches sur les processeurs accélérateurs afin d'obtenir de meilleurs performances.

En ce qui concerne les mesures des temps de communication, nous observons dans la figure 6.10, que les temps d'envoi des messages augmentent exponentiellement avec la taille des messages pour le processeur *ST40*. Il est à noter que ces temps évoluent différemment sur les deux types de processeurs *ST40* et *ST231*. En effet, nous constatons que le processeur *ST231* prend moins de temps que le *ST40* pour envoyer un message de la même taille. Cela suggère que l'architecture favorise l'accès à la mémoire aux accélérateurs. Suivant cette observation, l'application peut utiliser le *ST40* pour envoyer des données aux accélérateurs avec des messages d'une petite taille et mettre de grosses données en mémoire partagée. Au contraire, les *ST231*s peuvent enregistrer le résultat des calculs dans une mémoire tampon locale et la transférer vers le *ST40* avec peu de messages de grande taille.

6.4.3 Observations au niveau application

Nous avons comptabilisé le nombre d'appels aux fonctions `send` et `receive` afin de quantifier la communication entre les composants. Nous avons obtenu 10 386 appels (autant de `send` que de `receive`) pour le composant `Fetch-Reorder` et 5 193 appels pour les composants `IDCT`. Le composant `Fetch-Receive` exécute en effet le double d'opérations comparé aux composants `IDCT` ce qui nous donne des indications sur les schémas algorithmiques utilisés dans l'application MJPEG.

Cette information nous a également servi pour le calcul de la quantité totale d'images décodées. En effet, chaque message échangé entre le processeur maître et un accélérateur, contient une portion d'une des images à décoder. Chacune des images est décomposée par la fonction `Fetch` en 18 portions. À partir de l'observation des messages échangés, nous avons observé 577 *images* décodées par l'application MJPEG.

Les informations obtenues par l'observation de l'application MJPEG permettent une

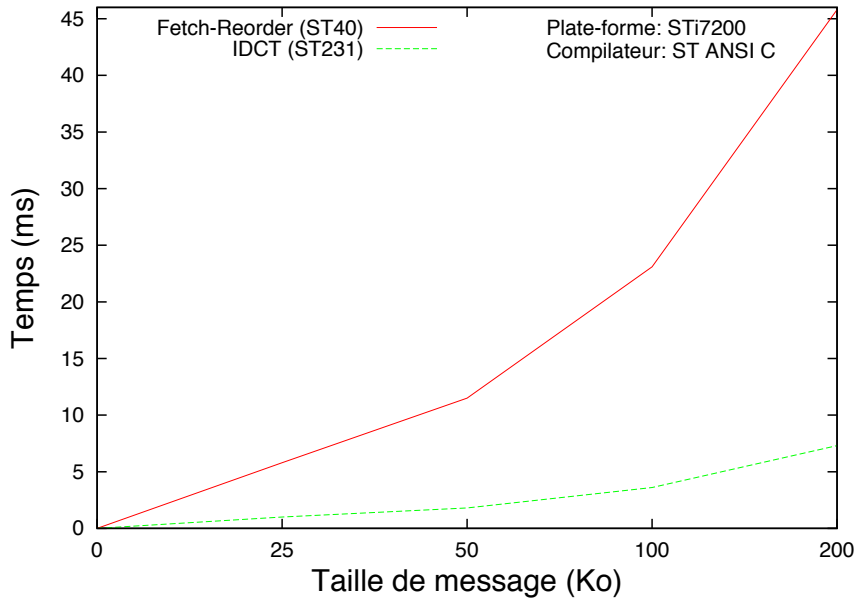

FIGURE 6.10 – Temps de communication en utilisant la primitive `send`

meilleure compréhension dus comportement des applications et contribuent à trouver une amélioration des performances. Par exemple, le temps d'exécution des composants indique que l'application MJPEG est bien équilibrée par rapport à la taille de l'entrée des images JPEG. Cependant, si la taille change, l'exécution des fonctionnalités `Fetch` et `Reorder` peuvent provoquer un goulot d'étranglement sur les composants `IDCT`.

6.5 Synthèse

Dans ce chapitre, nous avons présenté la première des études de cas pour notre modèle EMBera. Le décodeur MJPEG est une application ayant des modules basiques qui sont présents dans la plupart des algorithmes de décodage de vidéo, c'est-à-dire, la décomposition des trames encodées, la transformation des trames et la recomposition, ainsi que le rendu des images.

Nous avons utilisé notre modèle pour l'observation de ces modules, que nous avons encapsulés à l'aide de composants EMBera. Cette encapsulation nous a permis d'observer des paramètres comme la mémoire utilisée et le temps d'exécution, sans modifier les modules de l'application MJPEG. En outre, nous avons observé les interactions entre les modules déployés dans des processeurs différents. L'observation des interactions nous donne des informations sur la qualité de la vidéo décodée en termes d'images effectivement traitées.

L'observation de MJPEG sur une vraie plate-forme embarquée a servi à l'amélioration de l'implémentation du décodeur. Par exemple, nous avons pu corriger des problèmes concernant la communication entre les processeurs, ainsi que la gestion de la mémoire.

De plus, nous avons constaté l'avantage d'utiliser le processeur maître pour déployer des composants effectuant des opérations d'entrée et de sortie, et les processeurs dédiés pour le déploiement des composants exécutant la transformation des trames.

L'absence de contraintes temps réel de l'application MJPEG fait que notre modèle n'a pas pu être validé dans un contexte embarqué ayant des restrictions temporaires.

Le chapitre suivant présente une deuxième étude de cas d'utilisation du modèle EMBera, cette fois-ci, pour l'observation d'une application de décodage des multiples formats vidéo, utilisée dans des produits STMicroelectronics. Dans ce cas, le respect des contraintes de temps réel est déterminant pour assurer la qualité de la vidéo décodée.

Chapitre 7

Étude de cas : canevas industriel pour des applications multimédia

Nous avons effectuée une deuxième étude de cas pour la validation expérimentale de notre modèle d'observation EMBera. Elle concerne une application industrielle de décodage multimédia. Cette application se sert d'une pile logicielle et d'une plate-forme matérielle utilisées dans de nombreux produits de STMicroelectronics. Dans ce cas, la plate-forme est une *set-top box*, qui sert au décodage vidéo de haute définition.

Nous considérons que l'analyse de l'application de cette étude de cas est représentative du domaine embarqué. En effet, il s'agit d'une application embarquée industrielle présente aujourd'hui dans des produits finaux de STMicroelectronics. Par ailleurs elle est soumise aux contraintes de temps réel.

Dans ce chapitre, nous présentons en premier lieu les caractéristiques de la plate-forme matérielle et de la pile logicielle, nécessaires à la compréhension du système à observer. Puis nous présentons la mise en ouvre du modèle à composants EMBera sur cette plate-forme, ainsi que de l'application de notre modèle d'observation EMBera à cette étude de cas. Enfin, nous montrons et analysons les résultats obtenus lors de l'observation et nous concluons le chapitre par une synthèse.

7.1 Architecture de la plate-forme embarquée

Nous avons utilisé la plate-forme IDTV [STM07c] [AS08] : une plate-forme multiprocesseur spécialisée dans le décodage de la télévision numérique.

Configuration matérielle

La plate-forme IDTV appartient à la même famille que la *STi7200*, utilisée dans l'étude de cas précédente. La configuration matérielle est montrée sur la figure 7.1. Les différences principales entre cette plate-forme et la *STi7200* sont le nombre de proces-

seurs accélérateurs (deux *ST231* contre quatre), la vitesse d'horloge du processeur *ST40* (265 MHz contre 400 MHz) et la taille de la mémoire SDRAM (128 Mo contre 2 Go).

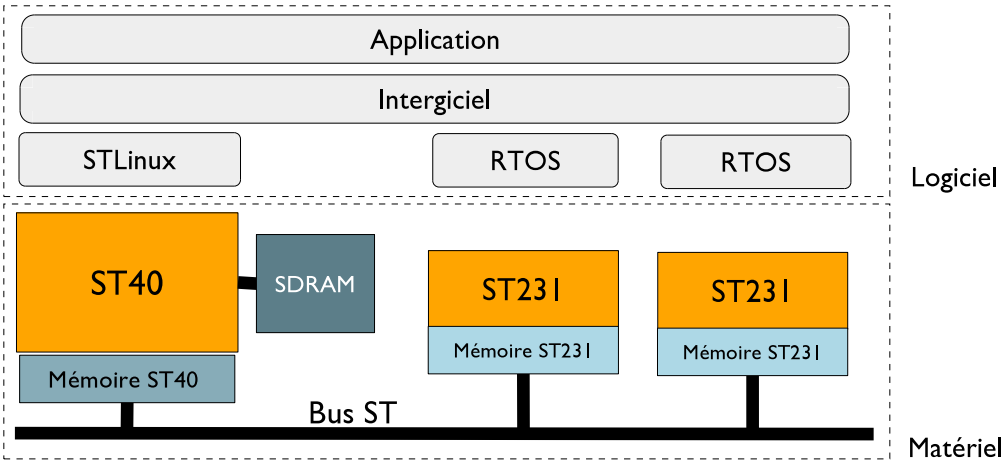

FIGURE 7.1 – Plate-forme *IDTV*

Configuration logicielle

L'application multimédia de utilise différents couches logicielles pour son exécution. Nous décrirons ci-après l'application multimédia, l'intergiciel de communication et les systèmes d'exploitation.

Application multimédia

Il s'agit d'une couche qui abstrait l'intergiciel multimédia et qui fournit des commandes à l'utilisateur pour le contrôle de fonctions de décodage de l'intergiciel. Les commandes permettent principalement d'accéder aux flux multimédia, ainsi que de démarrer, d'arrêter et de mettre en pause le décodage. Le code de l'application est exécuté entièrement sur le processeur maître.

Intergiciel multimédia STMicroelectronics

L'intergiciel multimédia[1] offre les fonctionnalités de décodage offertes par la plate-forme ainsi que les primitives de communication entre les processeurs.

Les fonctions offertes par l'intergiciel sont utilisées au niveau de l'application et appellent directement les fonctions fournies par la plate-forme. Par exemple, si nous implémentons au niveau applicatif une fonction `PLAY(...)`, cette fonction fait appel à la fonction de l'intergiciel `STMediaPlayer::Play(...)`. Cette dernière fait, par la suite, appel à la fonction correspondante pour démarrer la lecture de trames de la vidéo ainsi

1. Nous appellerons cet intergiciel par la suite aussi *IMST*.

que pour le décodage sur les accélérateurs et le rendu sur le dispositif de sortie. Pour la communication, l'intergiciel utilise les primitives de EMBx, décrites dans le chapitre précédent.

Systèmes d'exploitation STLinux et RTOS de STMicroelectronics

Deux systèmes d'exploitation sont utilisés pour l'exécution de l'application multimédia (cf. figure 7.1). L'un est STLinux [STM09b], une version embarquée du noyau Linux 2.6 s'exécutant sur le processeur *ST40*. L'autre est un RTOS, développé par STMicroelectronics, s'exécutant indépendamment sur chacun des processeurs accélérateurs. Le RTOS utilisé dans ce chapitre a été précédemment introduit dans la section 6.1.

Les deux systèmes d'exploitation utilisent des mécanismes pour gérer simultanément plusieurs flots d'exécution. Pour STLinux nous parlons de *processus* et *threads*, dont nous pouvons obtenir leur temps de création et d'exécution ainsi que l'utilisation de ressources (p. ex. le processeur et la mémoire). Pour le RTOS, nous avons vu que les flots d'exécution, appelés des « tâches », fournissent une interface du système qui donne accès sur les temps de création et de destruction, les priorités, le temps d'inactivité et l'utilisation de la mémoire disponible.

Les mécanismes fournis par les systèmes nous serviront pour la production des données brutes pendant l'observation de l'application avec notre modèle EMBera.

7.2 Mise en œuvre du modèle EMBera

Nous présentons dans cette section les aspects les plus représentatifs de l'implémentation du modèle à composants EMBera pour la plate-forme *IDTV*. Nous avons décidé d'observer la plate-forme depuis le processeur *ST40*. Pour cette raison, nous avons implémenté le modèle à composants de façon à être exécuté sur le système d'exploitation STLinux sur le processeur maître. Les interactions avec les accélérateurs sont observées en utilisant la couche intergicielle qui gère accélérateurs.

Vu que l'exécution de l'application à observer est faite sur un processus utilisateur STLinux, nous avons décidé qu'une instance d'EMBera doit aussi utiliser un processus utilisateur Linux pour son exécution. Cela permet au modèle d'observation de s'exécuter indépendamment à l'application mais aussi de communiquer avec le processus applicatif à l'aide de sockets.

Le processus qui exécute EMBera crée des flots d'exécution, qui sont rattachés à chaque composant (des threads), et de gère les interfaces d'observation. Un composant EMBera est composé d'une structure de données et deux threads, à l'exception du composant de stockage, qui utilise un seul. Le premier de ces threads exécute les fonctions des interfaces de description et de contrôle. Le second thread exécute les fonctions définies par l'interface de données.

Une interface fournie est représentée par une structure de données FIFO que nous

avons nommée une « boîte aux lettres ». À travers cette dernière, le composant reçoit des messages. Une interface requise est un pointeur vers une interface fournie, qui est utilisé pour envoyer des messages. Une connexion est effectuée en établissant le pointeur de l'interface requise vers une interface fournie donnée.

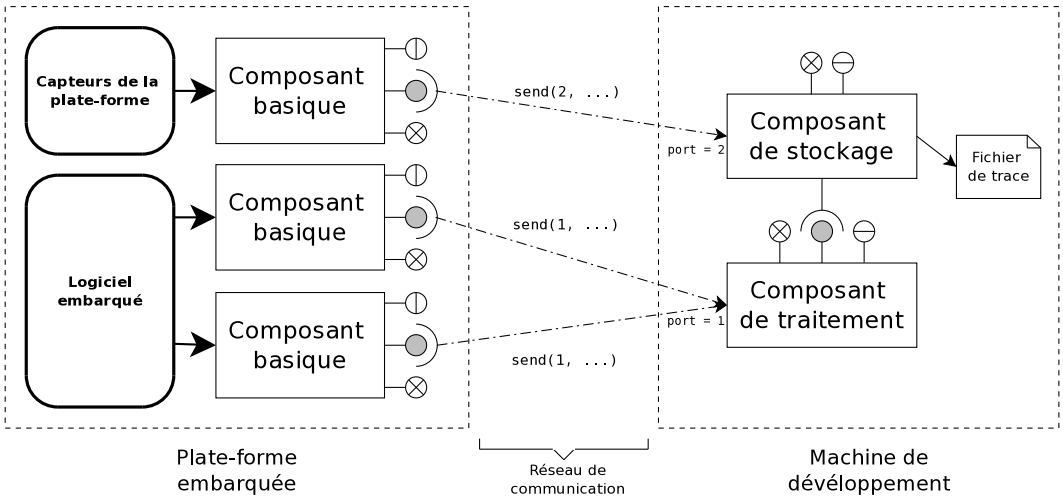

FIGURE 7.2 – Exemple de déploiement des composants EMBera

Deux implémentations des primitives de communication ont été expérimentées. La première utilise une lecture/écriture sur une zone de mémoire partagée du *ST40*. La seconde est basée sur des sockets pour permettre la communication entre composants s'exécutant sur des plates-formes distantes, reliées par un réseau de communication. Dans le cas précis de l'observation, cette communication permet un déploiement plus flexible. Par exemple, nous pouvons envisager une configuration du modèle où les composants basiques sont déployés sur la plate-forme embarquée et les composants de traitement et de stockage sur des processeurs faiblement utilisés ou dans des machines distantes (cf. figure 7.2).

Le déploiement d'une instance du modèle est faite par l'invocation explicite des fonctions de contrôle dans une fonction `main` du langage C, de la manière présentée dans la figure 5.7.

7.3 Application du modèle d'observation EMBera

Nous organisons les observations effectuées selon trois niveaux d'abstraction : l'application, l'intergiciel et le système.

Nous définissons une instance du modèle d'observation par niveau logiciel. Les observations obtenues et leur analyse sont présentées dans la section 7.4.

7.3.1 Définition des objectifs d'observation et identification des entités

Les objectifs d'observation ont été définis après étude de l'application et suite aux discussions faites avec des utilisateurs de l'application à propos de leurs besoins.

1. *Évolution des threads utilisés par l'application* : nous avons besoin de connaître le nombre de threads utilisés tout au long de l'application. Cette information est collectée toutes les secondes.

2. *Utilisation de la mémoire* : nous avons besoin de connaître l'historique d'utilisation de la mémoire toutes les secondes.

3. *Taux de réussite lors du décodage de la vidéo* : il s'agit de détecter les trames mal décodées afin de raisonner sur la qualité de la vidéo.

4. *Historique des commandes utilisateur* : nous avons besoin de connaître l'historique de l'exécution des fonctions de l'application. Cette information nous permet de mieux situer une exécution lors d'une étape d'analyse, en la corrélant avec des données obtenues depuis les différents niveaux logiciels.

Nous listons, dans la table 7.1, les entités que avons décidé d'observer.

Objectif	Application	Intergiciel	Système	Entité identifiée
1. Évolution des threads utilisés par l'application			X	Processus de l'application
2. Utilisation de la mémoire			X	
3. Échecs de décodage de la vidéo		X		Fonctions de décodage
4. Historique des commandes utilisateur	X			Fonctions de l'application

TABLE 7.1 – Objectifs d'observation organisés par niveau logiciel

L'observation du *processus* de l'application nous donne l'évolution de la quantité des threads et de la mémoire utilisée. L'observation des *fonctions de décodage* fournit l'information sur les échecs de décodage des trames de la vidéo. Enfin, l'observation des *fonctions de l'application* nous donne l'information sur l'interaction de l'utilisateur avec l'application.

7.3.2 Identification des composants EMBera

Dans la suite, nous montrons l'instanciation des composants EMBera pour chaque niveau logiciel observé.

Observation de l'application

Pour collecter les données concernant les commandes applicatives, nous définissons un composant basique. Il encapsule l'application et relève les appels aux fonctions utilisateur. Les données sont enregistrées à l'aide d'un composant de stockage.

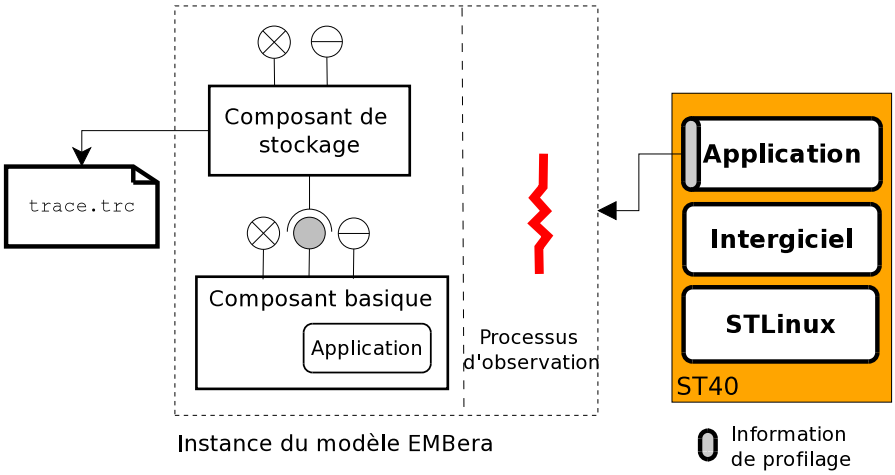

FIGURE 7.3 – Composants EMBera utilisés pour observer l'application de décodage

Observation de l'intergiciel

Cette deuxième instance du modèle d'observation met l'accent sur la qualité de décodage, un facteur crucial pour une application multimédia. Pour cela, nous observons deux fonctions : `AudioDec(frame)` et `VideoDec(frame)`. Ces fonctions décodent respectivement des trames audio et vidéo dans un flux vidéo.

Nous avons utilisé quatre composants : deux composants basiques, un composant d'agrégation et un composant de stockage (cf. figure 7.4).

Les composants basiques interceptent les valeurs de retour des fonctions de décodage à l'aide de fonctions de profilage (cf. figure 7.6). Le composant d'agrégation applique une fonction $failure(DecAud_i)\ AND\ failure(DecVid_i)$, où i est le numéro de la trame décodée. Le résultat de la fonction indique si la trame i a été correctement décodée. Le composant de stockage écrit les données produites par le composant d'agrégation sur un fichier local.

Observation du système d'exploitation

Pour l'observation du système, nous créons un composant basique et un composant de stockage, comme nous le montrons sur la figure 7.5.

Le composant basique observe le système d'exploitation en accédant à des fichiers du système. Pour cela, il lit les informations du fichier `/proc/PID/stat`. À partir de ce

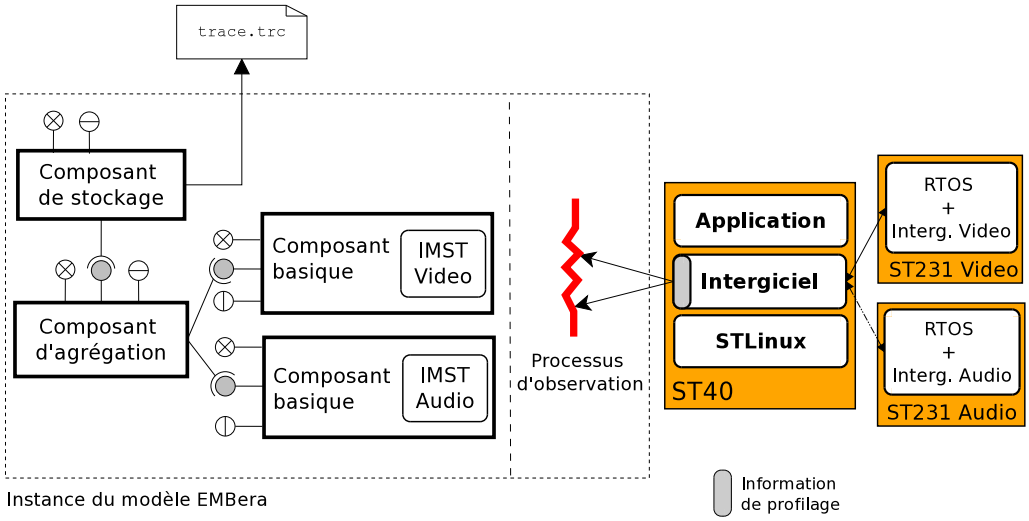

FIGURE 7.4 – Composants EMBera utilisés pour observer l'intergiciel multimédia

fichier, le composant récupère les valeurs de la mémoire utilisée et le nombre de threads. L'interface de contrôle est utilisée pour définir la taille initiale du tampon interne du composant basique, et la périodicité à 1 $échantillon/s$. Chaque échantillon est estampillé avec la fonction `gettimeofday`, puis enregistré dans le tampon.

Le composant de stockage implémente l'enregistrement des données dans des fichiers locaux et est configuré à travers l'interface de contrôle. Avec cette dernière, nous définissons le format $\{[timestamp], [memoryvalue], [threadnumber]\}$ pour les évènements observés.

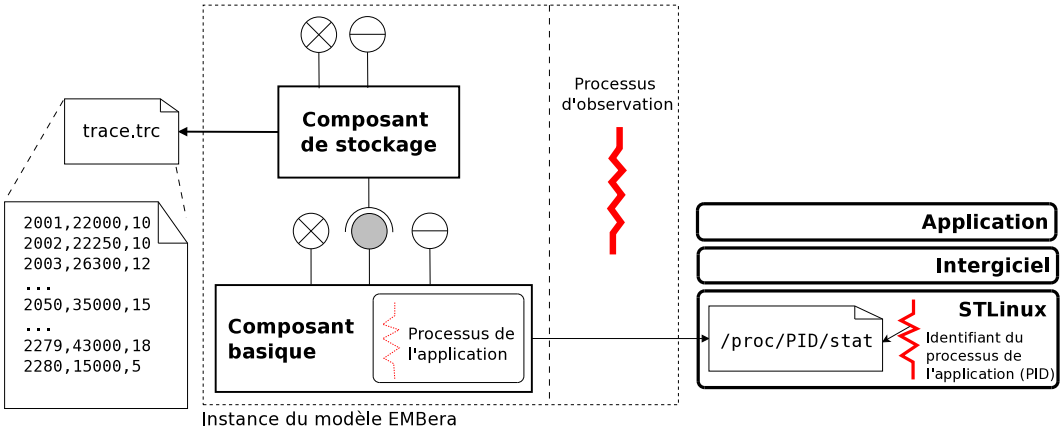

FIGURE 7.5 – Composants EMBera utilisés pour observer le processus de l'application

7.3.3 Préparation du système pour l'observation

Pour l'observation des entités de l'application et de l'intergiciel (respectivement des fonctions utilisateur et de décodage), nous utilisons une technique fournie par le compilateur `gcc` afin de ne pas modifier le code source de l'application. Il s'agit d'une fonctionnalité de profilage offerte par le compilateur [IBM09]. L'appel et le retour d'une fonction de l'application compilée sont remplacés par le compilateur pour une séquence faisant appel à une fonction de profilage suivi de l'appel ou du retour lui-même. Les fonctions de profilage peuvent être redéfinies pour un usage particulier. Dans notre cas, nous ajoutons des extraits de code EMBera pour la génération de données. La figure 7.6 montre un des ces extraits.

```
#include "embera.h"
...
void profile_func_enter(void* func_address, void* call_site) { /* Fonction de profilage appelée à
                                                                 l'entrée d'une fonction de
                                                                 l'application */
...
 if (getFuncName(func_address) == "userFuncName") {            /* Interception des appels */
  ...
  Event evt = createEvent (timeStamp, "userFuncName", ...);    /* Génération d'une structure avec
                                                                 l'information de l'évènement */
  port = getBasicPort(...);
  sendEvent (evt, port, ...);                                  /* Envoi de la structure vers
                                                                 un composant basique */
  ...
 }
}
```

FIGURE 7.6 – Extrait de code d'une fonction de profilage

La fonction `profile_func_enter` verifie si nous appelons une fonction déterminée (`userFuncName`). Puis nous créons une structure de données (`evt`) qui contient les données de l'évènement et leur estampille. Enfin, nous envoyons la structure vers un composant basique (`sendEvent`) à travers un port accessible par le composant.

Le code des fonctions de profilage est écrit dans des fichiers séparés de ceux du code source de l'application. Pour ajouter ces fichiers, nous recompilons l'application en ajoutant des options de débogage et de profilage ainsi qu'en indiquant l'emplacement des fichiers.

Cette technique est utilisée pour observer les niveaux intergiciel et application. Pour observer le système, nous nous nous servons des informations fournies par le système STLinux sur les processus dans l'espace `/proc/PID`.

7.3.4 Spécialisation du modèle d'observation pour la plate-forme

Nous présentons, dans ce qui suit, la spécialisation faite du modèle pour chaque type de composant du modèle d'observation.

Composant basique L'application et le logiciel produisent des données sur leurs exécutions grâce à l'ajout des extraits de code (cf. figure 7.6).

Nous programmons le code des composants basiques de manière à récupérer les données produites par ces extraits. L'exécution de l'instance du modèle et de l'application sont faites dans des processus indépendants que nous faisons communiquer en utilisant des sockets.

Pour l'observation du système, nous avons également spécialisé la fonction `getRawData` afin de récupérer des données depuis le fichier virtuel `/proc/PID/stat`. Le composant basique qui observe le processus de l'application implémente la lecture périodique des valeurs, pour obtenir la mémoire utilisée et le nombre des threads créés par le processus.

Composant de traitement de données Nous configurons un composant d'agrégation pour agréger les données collectées. Cette configuration a été effectuée à l'aide de l'interface de contrôle, sans modification du code source du composant.

Composant de stockage Le composant de stockage a été modifié pour supporter l'écriture des données sur un système de fichiers. Pour cela, nous avons spécialisé la fonction `writeData()` du composant de stockage. Les lignes du code ajoutées à la fonction comportent la mise en format des données et la gestion du fichier de trace (ouverture et fermeture des fichier et enregistrement des données observées).

7.3.5 Instanciation et déploiement des composants EMBera

Une instance du modèle d'observation EMBera est définie dans un programme C, en invoquant directement des fonctions de contrôle pour la création des composants du modèle d'observation EMBera et la gestion de leur cycle de vie (cf. figure 5.7).

Afin de connecter l'instance du modèle d'observation EMBera au niveau logiciel observé, les composants basiques, après leur démarrage, créent le processus qui exécute l'application de décodage (cf. figure 7.7). De cette manière, le composant basique connaît l'identifiant du processus observé et peut accéder au fichier système correspondant. Le cycle de vie des composants d'observation est terminé une fois que le processus de l'application de décodage se termine.

Dans l'étude de cas, les composants EMBera sont entièrement déployés sur le processeur *ST40*. Cependant, il serait possible d'établir un déploiement différent en utilisant la communication par *sockets* entre la production de données brutes et les composants de traitement. En effet, nous pouvons par exemple déployer les composants basiques sur la plate-forme embarquée et les composants de traitement et stockage sur une machine de développement, si nous avons un réseau d'interconnexion entre les plates-formes.

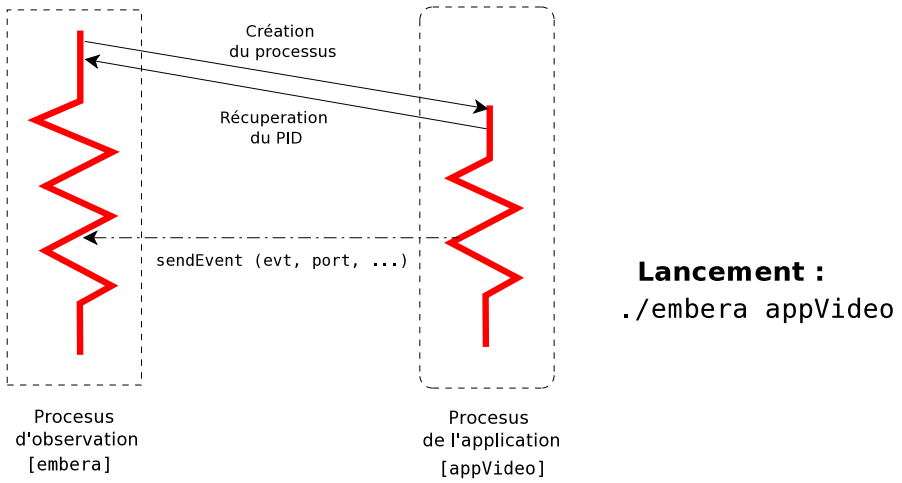

FIGURE 7.7 – Flots d'exécution créés pour l'observation de l'application

7.4 Observations effectuées et problèmes détectés

Nous présentons dans cette section les résultats obtenus suite à l'observation de l'application de décodage multimédia. Nous discutons d'abord les observations des niveaux intergiciel et système. Puis nous utilisons les résultats au niveau de l'application pour effectuer une corrélation entre les différents niveaux observés.

7.4.1 Observation au niveau de l'intergiciel

Nous avons observé le décodage de deux flux vidéo différents (d'une durée de 75 *s* et 1 300 *s* respectivement). La table 7.2 montre, en deuxième colonne le nombre d'images qui n'ont pas été décodées pour chacun des cas.

Durée de la vidéo (s)	# des trames non décodées	# total de trames
75	24	1 800
1 300	212	31 200

TABLE 7.2 – Nombre d'échecs de décodage des trames

La plupart des échecs sont dûs soit à des données de trames corrompues, soit à des échéances non respectées. Nous avons constaté que la plupart des trames non décodées pour la vidéo de 75 *s* est due principalement à des erreurs dans les données des trames. Pour la plus longue vidéo, les trames non décodées correspondent à la fois à des trames corrompues et à des délais non respectés. Nous considérons que dans des systèmes d'exploitation pourvus d'un ordonnanceur préemptif (tels que STLinux), l'exécution des tâches du système peut interrompre des tâches de décodage. Autrement dit, plus l'exécution de l'application de décodage est longue, plus est probable que le système interrompe l'exécution de l'application.

Afin de prouver notre hypothèse sur l'ordonnancement, nous avons utilisé l'outil KPTrace (cf. section 2.5.1.3) pour le traçage détaillé du système d'exploitation. Nous présentons dans la figure 7.8 l'exécution des fonctions de décodage `STVID.Produc` et `STVID.H264Pa`, qui ont une forte dépendance entre elles. Dans la partie gauche de la figure, nous pouvons voir que les deux fonctions sont exécutées l'une après l'autre. La partie droite de la figure montre un cas où d'autres tâches ont été ordonnancées entre les deux fonctions. Ces tâches introduisent des délais à l'exécution de `STVID.H264Pa` et sont l'une des causes des échéances non respectées. Ce genre de comportement constitue un cas d'erreur qui peut être détecté facilement à l'aide d'EMBera.

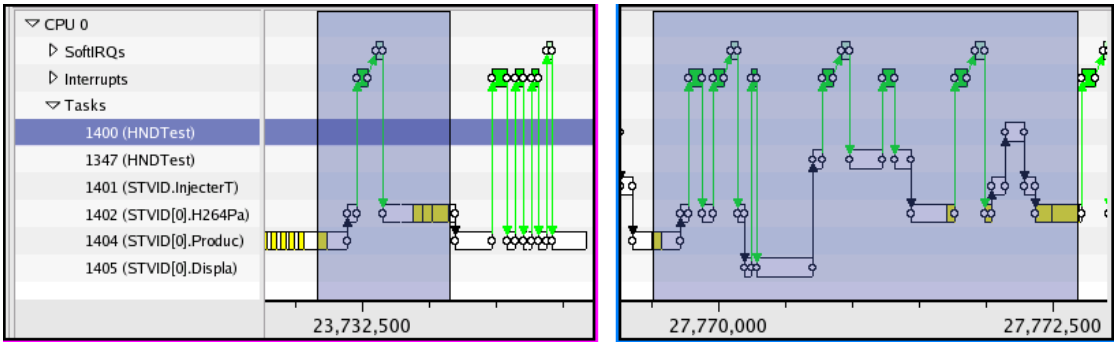

FIGURE 7.8 – Différences des temps d'exécution des fonctions de décodage

7.4.2 Observation au niveau du système

Pour illustrer l'information obtenue au niveau du système, nous avons décodé une suite de 3 vidéos, de 75 s chacune, codées en format MPEG-2. La figure 7.9 présente l'évolution de l'utilisation de la mémoire (ligne continue) et du nombre de threads (ligne pointillée).

La figure montre de manière consécutive l'évolution de la mémoire et des threads lors des 3 décodages vidéo. Nous distinguons 5 zones dans la figure : ① l'initialisation, ②, ③ et ④ les exécutions du décodeur et ⑤ la terminaison.

Nous pouvons observer en ①, lors du démarrage de l'application, qu'une quantité importante de la mémoire de l'application est allouée, mais aussi que 10 threads sont créés. Les deux opérations s'effectuent avant que l'application démarre le décodage d'un flux multimédia. Vers le temps 110 s (fin de ②), nous pouvons observer une décroissance de la quantité de la mémoire allouée et du nombre de threads utilisés. Cette variation correspond la fin d'un décodage du flux. La zone ③ alloue une taille de mémoire plus importante que ②. Comportement similaire pour ④ par rapport à ③.

Le début de la zone ⑤ montre qu'après trois exécutions du décodeur, la mémoire allouée a augmentée de 5 Mo, tandis que le nombre de threads utilisé est le même pour les trois exécutions du décodeur. Ce comportement correspond à une fuite mémoire, car des zones allouées entre deux exécutions du décodeur n'ont pas été libérées.

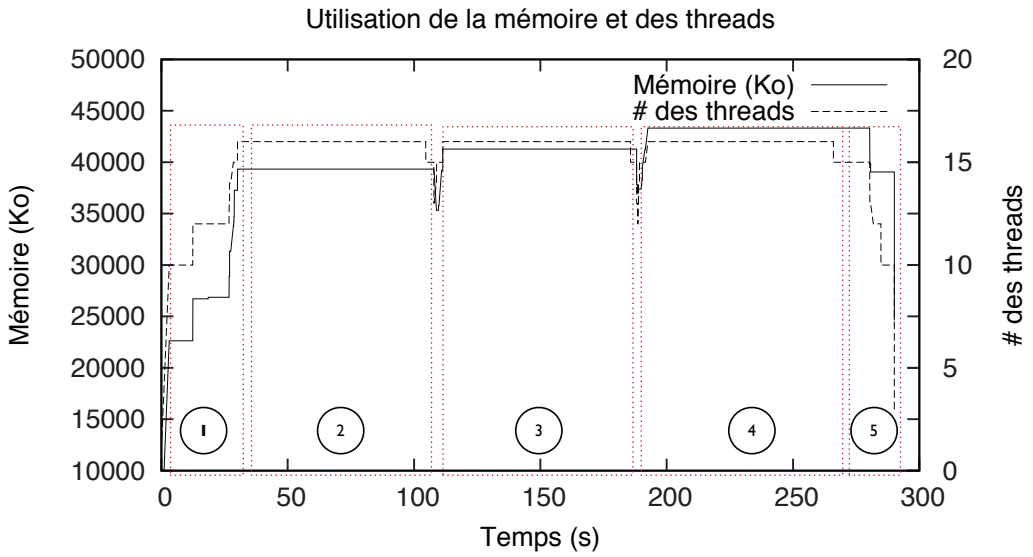

FIGURE 7.9 – Évolution de l'utilisation de la mémoire et des threads utilisées par l'application sur trois exécutions

Cette information est très utile dans la conception et la mise au point des plates-formes embarquées car elle offre des précisions pour le dimensionnement des plates-formes.

7.4.3 Corrélation entre les observations

Afin de présenter une vue sur les trois niveaux observés, nous déployons en même temps les trois instances du modèle d'observation EMBera (cf. section 7.3.5). Chacune des instances produit un fichier de trace avec les données observées. Étant donné que nous avons effectué les observations sur le même processeur, les estampilles des traces possèdent la même base de temps, et donc, nous pouvons corréler leurs évènements observés à différents niveaux.

Pour la génération des données à trois niveaux logiques, nous avons effectué 10 exécutions de l'application de décodage, avec les mêmes données d'entrée. Nous considérons que le nombre d'exécutions suffit parce que la pile logicielle garantit des contraintes de temps réel et, par conséquent, assure un temps d'exécution identique de l'application. Pour chaque exécution, nous initialisons la plate-forme et décodons un flux vidéo de 75 s en format MPEG-2. Nous avons choisi de redémarrer la plate-forme à chaque décodage afin d'avoir les mêmes conditions initiales d'exécution. Nous présentons dans la figure 7.10 les valeurs moyennes des observations obtenues ainsi que la corrélation que nous pouvons en tirer.

Les observations obtenues permettent de comprendre plus précisément le fonctionnement de l'application et servent également à trouver des défauts de programmation.

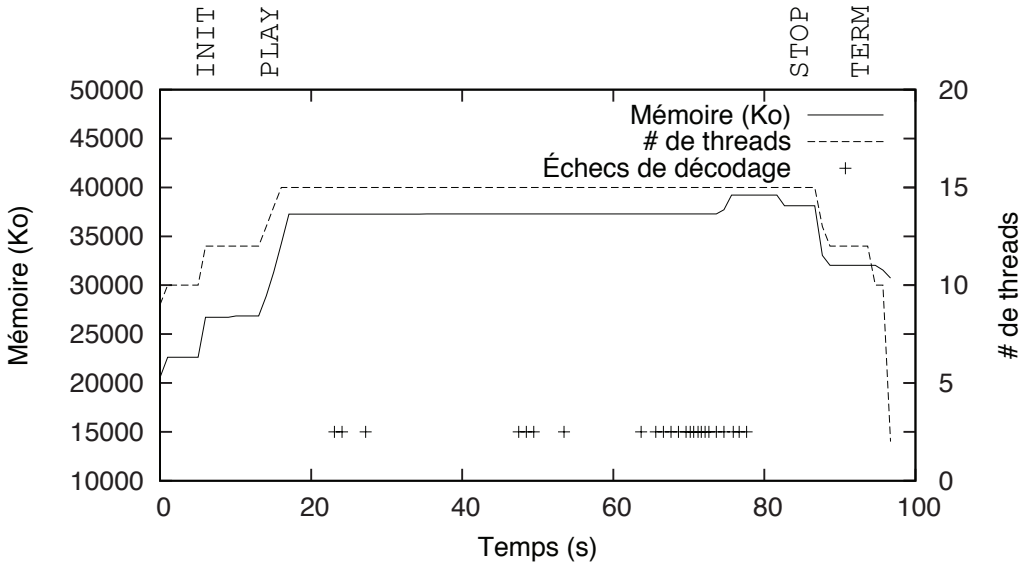

FIGURE 7.10 – Corrélation des évènements applicatifs avec l'intergiciel et le système

En ce qui concerne la compréhension de l'application, nous pouvons constater dans la figure que l'exécution de la commande utilisateur `INIT` (vers 7 s) déclenche la création des threads additionnels ainsi que l'allocation supplémentaire de mémoire. Lorsque la commande `PLAY` est exécutée (vers 13 s), des créations des threads et des allocations de la mémoire additionnels sont encore effectuées.

Pour ce qui est de la détection de défauts, nous nous intéressons uniquement ici aux échecs de décodage. Dans la figure 7.10, nous observons un nombre plus important d'échecs vers le temps 75 s. De plus, nous pouvons percevoir une allocation supplémentaire de mémoire. Vers le temps 80 s, l'exécution de la commande `STOP` libère une partie de la mémoire allouée pour arrêter le décodage et l'affichage de la vidéo. L'exécution de la commande `TERM` libère la mémoire réservée pour l'application et termine l'exécution des threads ajoutés pour la lecture de la vidéo.

L'augmentation du nombre d'échecs indique des trames corrompues dans le flux vidéo en entrée. L'utilisation de mémoire supplémentaire suggère que l'algorithme de décodage stocke des données sur les trames corrompues jusqu'à ce qu'il trouve une trame correcte. Nous pouvons induire que lors d'une longue suite de trames impossibles à décoder, le décodeur peut essayer d'utiliser plus que la mémoire totale disponible de la plate-forme, en produisant une terminaison abrupte de l'exécution du décodeur.

Notre observation suggère des solutions possibles. Par exemple, l'empêchement de l'allocation de la mémoire en présence d'un flux de données corrompues, pourrait éviter la fin inattendue de l'exécution. De la même manière, l'application pourrait réserver un tampon mémoire supplémentaire, à utiliser uniquement dans le cas de lecture de trames incorrectes, pour gérer les échecs.

7.5 Synthèse

Nous avons présenté dans ce chapitre la deuxième étude de cas pour la validation du modèle d'observation EMBera. Il s'agit de l'observation d'une plate-forme multimédia utilisée dans des produits STMicroelectronics.

Nous avons instrumenté le logiciel, sans introduire des modifications au code source de l'application, grâce aux options offertes par la compilation. Les composants EMBera ont été complètement indépendants de l'application observée. En effet, l'intégration du modèle à la plate-forme embarquée a été simple à effectuer : le lancement des composants d'observation a démarré l'application et a observé cette dernière jusqu'à la fin de son exécution.

Nous avons obtenu des observations intéressantes sur l'exécution de l'application dans les différents niveaux logiques choisis. Ces observations nous ont permis, d'une part, de comprendre le comportement de l'application et, d'autre part, de détecter des problèmes de fonctionnement. Nous avons vérifié des comportements attendus de la plate-forme, comme une utilisation plus importante des ressources lors de l'invocation des fonctions utilisateur. Par rapport aux problèmes détectés, nous pouvons souligner les suivants :

- Des fuites de mémoire ont été détectées lors des exécutions consécutives du décodeur.
- Des problèmes potentiels de non-respect des échéances de décodage multimédia dus à l'ordonnancement des tâches par le système d'exploitation.
- Des échecs de décodage mal gérés ce qui a induit une utilisation inattendue des ressources de la plate-forme.

Nous avons communiqué ces observations aux ingénieurs de l'équipe en charge du développement et la mise au point de l'application. Ils considèrent que les observations sont utiles, et pensent que le modèle d'observation peut être utile dans d'autres contextes d'analyse, par exemple, dans des tests de régression.

Chapitre 8

Étude de cas : décodeur SVC

Ce chapitre présente la troisième étude de cas qui porte sur une application de décodage SVC [1], programmée à l'aide des composants. En effet, les méthodes de développement d'applications embarquées ont récemment commencé à évoluer vers l'utilisation des stratégies plus modulaires avec des éléments réutilisables ou reconfigurables. C'est pourquoi, les composants logiciels ont été adoptés comme une alternative au développent du logiciel des futurs systèmes sur puce.

L'application de cette étude de cas est un décodeur configurable, c'est-à-dire, qui permet l'adaptation à la volée des paramètres du décodage (affichage et calcul), quand les caractéristiques de fonctionnement varient (type de plate-forme, bande passante) [SW08]. Nous considérons qu'elle est représentative aux applications embarquées pour les futurs MPSoC.

Dans cette étude de cas, nous ne visons pas à changer l'infrastructure à composants utilisée par l'application pour son observation. Au contraire, nous proposons d'utiliser cette infrastructure pour implémenter et appliquer le modèle d'observation EMBera.

Pour nos expérimentations, nous utilisons une plate-forme multi-cœur homogène non-embarquée. Ce choix a été fait pour deux raisons : d'une part, l'application a été conçue pour exécuter certains de ses composants en parallèle en utilisant une mémoire partagée ; d'autre part, à l'époque des expériences, nous n'avions pas accès à des plates-formes embarquées SMP [2]. Par ailleurs, au moment de l'étude, le support du canevas de composants, pour les plates-formes embarquées de nos expériences, n'était pas disponible.

Dans ce chapitre, nous présentons premièrement les caractéristiques matérielles et logicielles de la plate-forme d'expérimentation utilisée pour notre étude. Puis nous présentons l'application du modèle d'observation EMBera sur une implémentation du décodeur SVC. Enfin, nous montrons et discutons les observations obtenues.

1. Acronyme anglais : *Scalable Video Coding*
2. Acronyme anglais : *Symmetric Multiprocessing*

8.1 Architecture de la plate-forme multiprocesseurs

Les expériences de cette étude de cas ont été effectuées sur une machine SMP avec des processeurs généralistes. Nous montrons les caractéristiques matérielles et logicielles de la machine que nous utilisons pour l'observation avec le modèle EMBera.

Configuration matérielle

Il s'agit d'une plate-forme Intel Xeon x7460 [Int08] de 64 bits à quatre processeurs ayant 6 cœurs chacun, cadencés à 2.6 GHz. La plate-forme est munie de 64 Go de mémoire vive. Elle possède trois niveaux de mémoire cache dont le niveau 3 (`L3`) est partagé entre tous les cœurs et le niveau 2 (`L2`) partagé entre deux cœurs. Le niveau 1 (`L1`) est propre aux cœurs. L'organisation des cœurs et des caches de l'un des processeurs est présentée dans la figure 8.1.

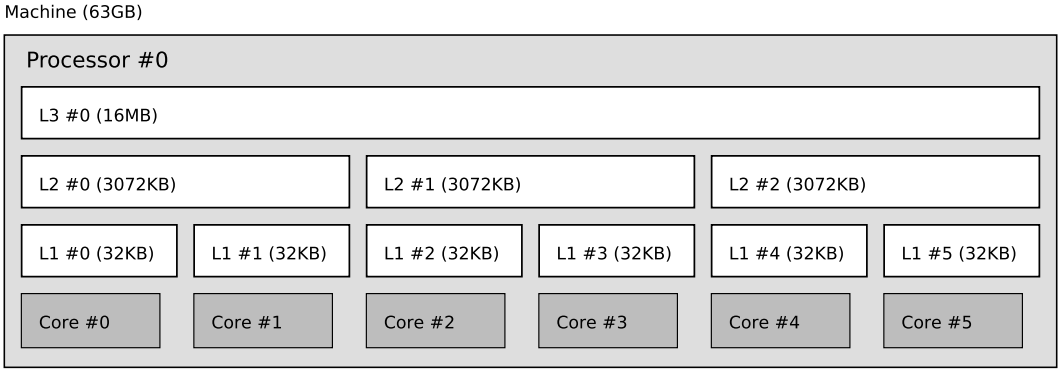

FIGURE 8.1 – Organisation des cœurs et de la mémoire cache dans le processeur Intel x7460

Pour nos expérimentations nous avons utilisé seulement l'un des processeurs de la plate-forme. Nous avons constaté des meilleures performances si l'application et le modèle d'observation sont déployés sur des cœurs du même processeur. Cela est dû au fait que des cœurs confinés dans un même processeur profitent de la localité du cache.

Configuration logicielle

Le logiciel est présenté selon trois niveaux logiques bien différenciés, à savoir, le canevas à composants Cecilia et l'application de décodage SVC, l'intergiciel de communication et déploiement Comete, et le système d'exploitation Linux.

Le canevas Cecilia et le décodeur SVC

Le niveau applicatif est composé, d'une part, d'un canevas de composants logiciels et, d'autre part, de l'application de décodage, programmée à l'aide du canevas. Ci-après,

nous rappelons et approfondissons des concepts de Cecilia (cf. section 3.3.2.1), avant d'introduire le décodeur SVC.

Le canevas Cecilia : L'environnement Cecilia, développé par l'INRIA et France Telecom, fournit un canevas pour le développement d'applications basées sur Fractal en langage C [Obj09a].

Pour la construction d'applications, Cecilia utilise trois langages : le ADL, pour définir l'architecture de l'application, l'IDL, pour définir les interfaces et ThinkMC [Obj09b], qui décrit la définition de composants et permet leur implémentation en langage C. En effet, ThinkMC permet l'implémentation des composants en C, en utilisant des macros prédéfinies pour gérer l'interaction avec le code généré par la chaîne d'outils à partir des fichiers ADL et IDL.

La chaîne d'outils utilisée dans Cecilia fonctionne en deux étapes : la première étape est une étape de compilation des fichiers ADL et IDL en fichiers C. Pour cela les fichiers sont analysés par une chaîne de traitement Java, qui crée une représentation interne de l'application. Des fichiers sont ensuite générés à partir des informations de cette représentation : des fichiers `.adl.c` pour les composants, et des fichiers `.idl.h` pour les interfaces. Ces fichiers définissent également des macros ThinkMC. La seconde étape de traitement est la compilation finale de ces fichiers avec les fichiers d'implémentation ThinkMC, afin de générer un exécutable. Un compilateur `gcc` est utilisé pour cela (cf. figure 8.2).

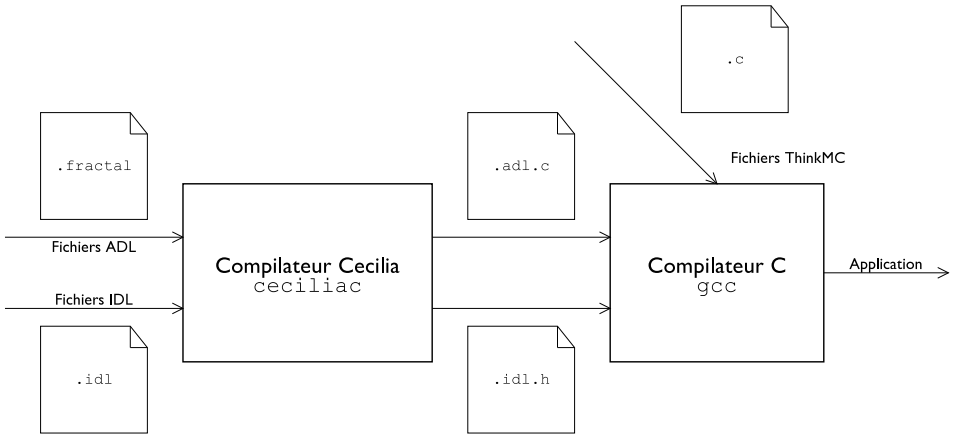

FIGURE 8.2 – Chaîne d'outils Cecilia

Le modèle d'exécution de Cecilia établit que tout appel entre composants est un appel direct à la fonction de l'interface fournie du composant lié au composant appelant. Ce modèle ne fonctionne que pour des plates-formes à mémoire partagée. Cecilia utilise un seul flot d'exécution qui parcourt toutes les fonctions appelées de manière synchrone (cf. figure 8.3).

Décodeur SVC : Le *Scalable Video Coding* [SW08] est un amendement de la norme H.264/AVC [VWS11]. Un décodeur qui supporte SVC est capable d'adapter ses résultats

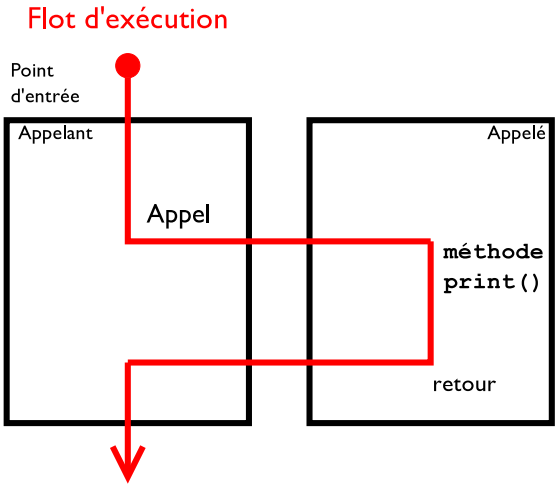

FIGURE 8.3 – Communication et exécution dans Cecilia

de décodage aux changements dans la vidéo d'entrée, tels que le nombre d'images par seconde, ainsi que la dimension et la qualité des images. Autrement dit, une vidéo encodée une fois en SVC, peut offrir différents débits avec différentes qualités. Les trames de la vidéo sont organisées dans des groupes d'images, qui possèdent des données communes entre elles. La vidéo d'entrée peut être lue à partir soit d'un fichier, soit d'un flux réseau. S'il s'agit d'un fichier, l'information des changements des paramètres de la vidéo est incluse dans le fichier.

Une version du décodeur SVC a été développée par des chercheurs du groupe *Image Communication* à l'Institut *Heinrich Hertz* à Fraunhofer (HHI)[3]. Le décodeur a été organisé dans des composants logiciels Cecilia. Il est important à noter que cette version du décodeur n'implémente pas des mécanismes pour garantir le temps réel. La figure 8.4 présente l'architecture simplifiée du décodeur SVC. Les composants que nous y voyons sont :

- `SvcTest` : ce composant composite fournit le point d'entrée principal de l'application (fonction `main`). Le composant est en charge de la gestion des flux vidéo en entrée et en sortie. Les flux en entrée sont découpés dans des paquets, puis envoyés vers le composant `SvcDecoder`.
- `SvcDecoder` : il s'agit d'un composant composite qui extrait l'information du flux vidéo et permet de contrôler le décodage en termes de taille, qualité des images et nombre d'images par seconde. La gestion de ces paramètres est faite par les composants `Sei`, `Sps` et `Pps`. Une fois que les paramètres sont établis, le paquet reçu est envoyé vers le composant `Slice`.
- `Slice` : ce composant composite fournit des paquets de données de la vidéo et contrôle la plupart des opérations de décodage. Ces opérations sont effectuées

3. `http://ip.hhi.de/imagecom_G1/savce/index.htm`

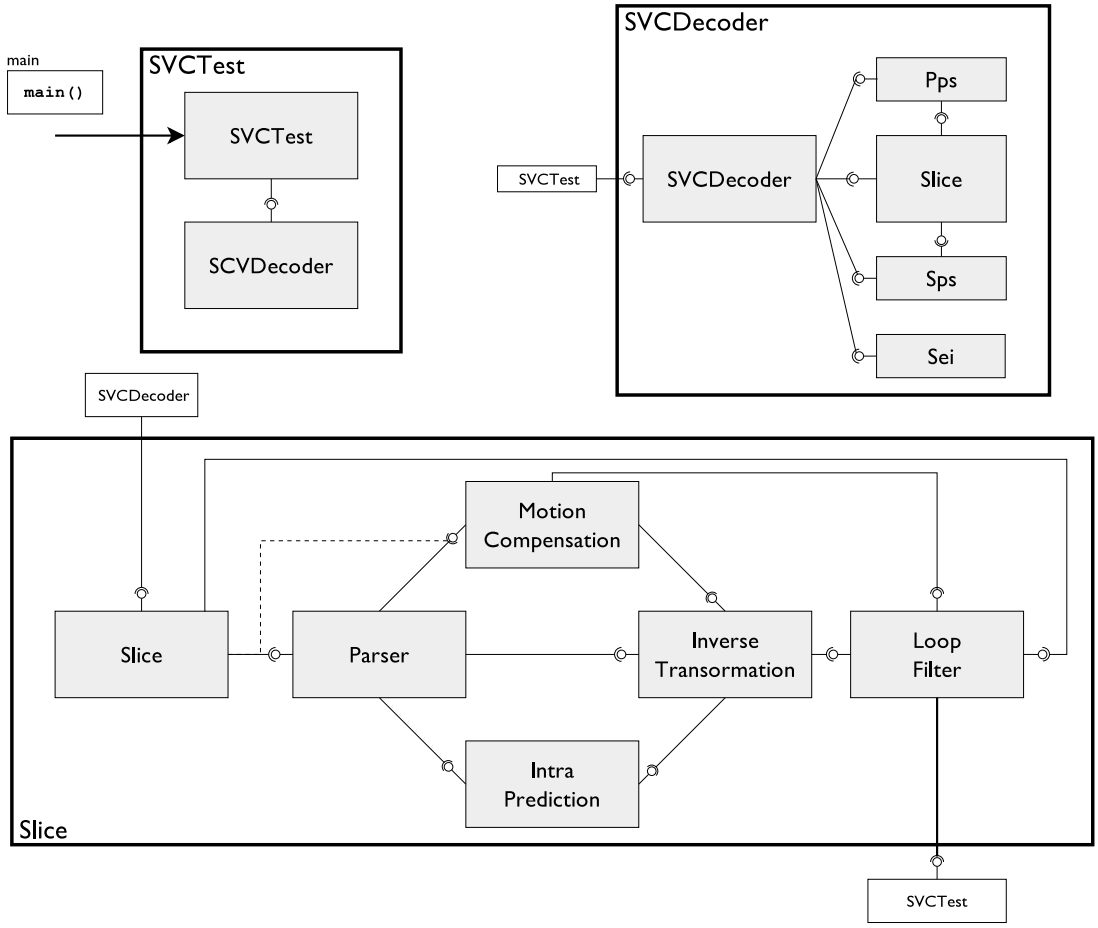

FIGURE 8.4 – Architecture de l'application de décodage SVC implémentée avec des composants Cecilia.

par les composants `Parser`, `MotionCompensation`, `IntraPrediction` et `InverseTransformation`.
Lorsqu'un paquet est traité, `Parser` utilise soit `MotionCompensation`, soit `IntraPrediction` pour une première étape dans le décodage. Puis il applique les traitements de `InverseTransformation` pour produite des paquets décodés. Le résultat du traitement de paquets de données est envoyé vers le composant `LoopFilter`.

– `LoopFilter` : ce composant reçoit le résultat du décodage géré par `Slice`, et applique la dernière étape du traitement. Puis il délivre des images vers le composant `SvcTest` pour leur affichage ou leur enregistrement dans des fichiers.

Intergiciel à composants Comete

Comete est un projet STMicroelectronics qui fournit un support pour le déploiement dynamique et la communication de composants Cecilia sur différentes plates-formes. L'in-

tergiciel Comete est lui-même organisé en termes de composants Cecilia. Comete se sert de la bibliothèque Minus [STM08], utilisée pour le développement des intergiciels pour des plates-formes MPSoC.

Nous rappelons que Comete définit le concept de *Processing Element* ou *PE*, comme une abstraction d'une unité de calcul, tel qu'un processeur ou un cœur (cf. section 3.3.2.1). Comete permet aux composants d'être déployés sur un PE ou redéployés d'un PE à un autre pendant leur exécution. Les composants peuvent s'exécuter en parallèle si les PEs abstraient des unités de calcul indépendantes.

L'utilisateur de Comete définit ses besoins pour l'application : la plate-forme de destination, les protocoles à utiliser pour les communications, et le déploiement des composants de l'application sur les différents PEs.

À partir de ces données, Comete sélectionne le compilateur adapté, rajoute les composants requis pour la communication selon le protocole choisi, rajoute les composants pour l'exécution de Comete sur les différents PE (ordonnancement, allocations), et assure la gestion du déploiement des composants. Dans cette étude de cas, nous utilisons le protocole de communication de Comete pour les systèmes à mémoire partagée connu comme *SHMCBST*. Nous soulignons que ce protocole est asynchrone et n'assure pas un retour de fonction.

Pour le déploiement et l'exécution, Comete effectue deux étapes :

1. *Pré-déploiement de Comete* : celui-ci est fait par les composants `ComponentManager` et `ComponentLoader`. Ces composants respectivement gèrent et chargent les composants de l'application, puis lancent l'exécution des composants. Le `ComponentManager` utilise le composant `ApplicationDB` pour enregistrer dans une base de données les informations relatives à la création des composants et à leurs connexions. Ensuite, un flot d'exécution est créé ou réservé par PE défini (p. ex. un thread par cœur). Dans cette étude de cas, nous utilisons le support de *pthread* pour l'exécution du décodeur SVC.

2. *Déploiement des composants de l'application* : il consiste à attribuer l'exécution d'un composant à l'un des PEs. Nous montrons dans la figure 8.5 deux PEs exécutant deux composants.

Système d'exploitation Linux

Nous avons utilisé le système d'exploitation Linux, noyau version 2.6-32. Ce noyau offre la gestion de processus et une bibliothèque de threads. Il fournit également des interfaces pour l'allocation de la mémoire, comme `malloc` et `calloc`. L'accès à ces allocateurs est fait à l'aide des composants Minus, qui fournissent une couche d'abstraction du système d'exploitation, fondée sur des composants Fractal. Cette abstraction est utilisée par l'intergiciel Comete pour la création et la migration des composants.

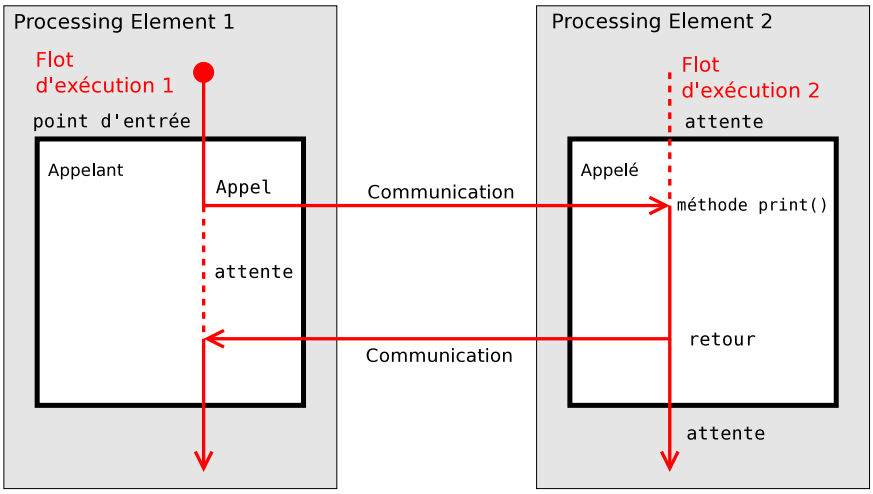

FIGURE 8.5 – Modèle d'exécution de Comete

8.2 Mise en œuvre du modèle EMBera

Nous avons décidé de nous servir du modèle à composants Cecilia pour implémenter le modèle EMBera. Ceci a été possible puisque nous avons eu accès au code source de Cecilia et de Comete. Ce choix nous a permis d'intégrer facilement les aspects d'observation dans le décodeur SVC et l'intergiciel. Ce travail d'implémentation a été effectué en majeur partie pas Augustin Degomme.

La mise en œuvre du modèle EMBera consiste en l'ajout des interfaces d'observation, définies dans la section 5.2.2, aux composants observés, ainsi qu'en l'implémentation des composants spécialisés d'EMBera à l'aide de Cecilia. Nous avons regroupé la mise en œuvre des interfaces et des composants dans un paquetage, appelé `observation`. Ce dernier est intégré à Cecilia comme l'une de ses dépendances requises, à l'aide du gestionnaire de paquetages d'installation Maven [Fou10].

Interfaces spécialisées

Pour rajouter les interfaces d'EMBera, nous affectons l'infrastructure à composants Cecilia. Pour cela, nous rajoutons la définition IDL des interfaces de description, de contrôle et de données d'EMBera ainsi que l'implémentation de leurs méthodes en langage C. Nous montrons dans la figure 8.6(a), la définition de l'interface de données, et dans la figure 8.6(b), la signature de la méthode qui retourne les données collectées.

L'intégration des interfaces aux composants applicatifs est réussie en étendant la définition de ces composants dans le fichier ADL, comme nous le montrons dans la figure 8.6(c). Grâce à cette définition, les composants applicatifs deviennent observables, c'est-à-dire, les interfaces d'observation EMBera sont créées pendant la compilation de l'application. L'intégration des interfaces d'observation aux composants de Comete uti-

```
 (a) package observation.treatment.api;

     interface DataItf {
      int getObservationData(uint64_t timestamp, string eventType, string obsType, list data);
     }
-----------------------------------------------------------------------------------------------
 (b) int METHOD(dataItf, getObservationData)
         (void *_this, uint64_t timestamp, char* type, char* obsType, intptr_t** dataList) {...}
-----------------------------------------------------------------------------------------------
 (c)  <definition name="components.slice.Slice" extends="observation.obs.lib.ObservedComponent">
```

FIGURE 8.6 – Modifications effectuées à Cecilia pour l'ajout d'EMBera

lise la même définition montrée dans la figure 8.6(a) et 8.6(b), mais elle est codée en dur au code de Comete.

Les modifications effectuées à la structure des composants Cecilia sont faites une seule fois et elles servent à l'observation de tous les composants programmés dans Cecilia, qui incluent la bibliothèque d'observation lors de leur définition. Suite aux modifications, nous recompilons l'environnement Cecilia pour intégrer notre paquetage à Cecilia. Pour Comete, les interfaces d'observation sont toujours présentes, mais afin de produire des données, l'observation doit être activée.

Le rajout automatique des interfaces EMBera permet aux composants de fournir une observation minimaliste par défaut. Les informations offertes par cette observation sont la liste d'interfaces, de méthodes et d'attributs ainsi que le temps d'exécution du composant. Nous clarifions que l'introspection proposée dans Fractal, fournissant les listes d'interfaces et de méthodes, n'était pas fonctionnel à l'époque dans Cecilia. C'est pourquoi, nous avons implémenté des mécanismes pour obtenir correctement ces informations dans le canevas Cecilia.

Implémentation des composants

Pour la mise en œuvre des composants d'observation EMBera, nous utilisons des composants Cecilia. Dans le paquetage, nous fournissons des implémentations par défaut des composants d'observation, qui peuvent être modifiées en fonction des besoins de l'observation.

La génération du code objet des composants EMBera est faite à la compilation de l'application. Ces composants s'intègrent à l'application en se connectant aux interfaces d'observation ajoutées sur les composants applicatifs.

Composant basique Nous avons montré que l'implémentation de Cecilia a été modifiée principalement pour ajouter les interfaces spécialisées du modèle EMBera. De cette manière, tous les composants applicatifs Cecilia créent, lors de leur instanciation, leurs interfaces d'observation. Cependant, afin de ne pas introduire d'intrusivité aux composants non-observés, l'observation n'est pas activée par défaut.

Comme les composants possèdent désormais des interfaces d'observation, nous pou-

vons nous servir de n'importe quel composant Cecilia comme composant basique, une fois que l'observation est activée.

Nous avons aussi implémenté des composants basiques indépendants de l'application, qui visent principalement l'observation du système et de l'intergiciel. Pour l'intergiciel Comete, nous observons les composants `ComponentLoader`, `ComponentManager` et `ApplicationDB`, ainsi que l'échange de messages entre composants Cecilia à l'aide de Comete. Nous montrons dans la figure 8.7 un composant basique observant la communication.

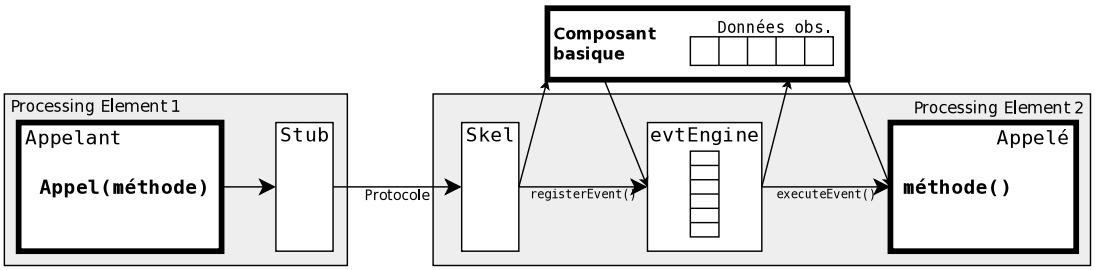

FIGURE 8.7 – Composant basique observant la communication dans Comete

Nous rappelons que les composants `Skel` et `evtEngine` de la figure gèrent la communication du coté du composant appelé. Nous connectons un composant basique aux composants `Skel`, `evtEngine` et au composant appelé. Ces connexions permettent au composant basique d'obtenir les temps entre l'envoi du message et la réception dans `evtEngine`, puis le temps de traitement, qui correspond au dépilage et à l'exécution de la méthode appelée.

Composants de filtrage et d'agrégation Ces composants fournissent les mêmes fonctionnalités décrites dans la section 5.2.4 : des méthodes pour définir les fonctions de filtrage et d'agrégation ainsi que pour les appliquer sur un ou plusieurs listes de données. Cependant, afin de permettre aux interfaces d'observation l'accès aux fonctions de filtrage et d'agrégation, ces dernières ont été implémentées à l'aide des macros définies par la chaîne d'outils Cecilia.

Composant de stockage Nous créons le composant de stockage en lui passant comme paramètres le type de stockage et la référence d'accès au support persistant où les données sont enregistrées (p. ex. le chemin vers un fichier de trace ou la chaîne de connexion à une base de données). Vu que chaque mécanismes d'enregistrement de données requiert une implémentation différente, nous en avons développée deux : l'un pour écrire sur des fichiers, et l'autre pour enregistrer les observations sur une base de données. Le `Default-Storage` écrit les données reçues dans un fichier, à l'aide de bibliothèques externes comme GTG [4], et peut gérer un tampon afin de limiter le nombre d'écritures dans le fichier.

4. Le générateur de traces génériques (GTG) est une bibliothèque pour la production de traces dans format Pajé et OTF [INR10a].

Le `SQLStorage` enregistre les messages reçus dans une base de données *sqlite*[5], pour permettre une exploitation simplifiée à l'aide du langage SQL. Chaque type de message différent est stocké dans une table différente de la base, qui fournit un champ estampille, et un champ data.

8.3 Application du modèle d'observation EMBera

Nous nous sommes intéressés à l'observation des performances du décodeur SVC. Pour cela, nous observons des aspects liés à la qualité des calculs effectués par le décodeur, ainsi qu'à l'utilisation des ressources de la plate-forme.

8.3.1 Définition des objectifs d'observation et identification des entités

Étant donné que le décodeur SVC est une application prototype qui n'a pas été testé auparavant sur des plates-formes avec plusieurs cœurs, nous avons voulu observer l'application afin d'évaluer principalement deux aspects :

- *Qualité de décodage* ***(Q)*** : il s'agit d'évaluer si l'application produit des résultats corrects dans un temps raisonnable sur la plate-forme multi-cœur de cette étude.
- *Utilisation de la machine* ***(U)*** : il s'agit de voir si l'application est optimale en termes d'exploitation de mémoire et de processeur.

Nous définissons un ensemble des métriques que nous avons organisé selon les niveaux d'observation utilisés :

- *Observation de la performance du décodeur SVC.*
  - *Nombre d'images par seconde* ***(Q)*** : pour estimer si le décodeur s'exécute dans un temps raisonnable, nous devons obtenir le nombre d'images décodées par seconde. Ce débit devrait être supérieur au nombre d'images par seconde demandé par le flux vidéo.
  - *Évolution dans les paramètres courants du décodage* ***(Q)*** : nous voulons vérifier si lors d'un changement dans les paramètres de décodage, le décodeur réagit correctement et fournit le nombre adéquat d'images par seconde.
  - *Taux de succès de décodage* ***(Q)*** : nous avons besoin d'obtenir le nombre de trames vidéo qui ont été correctement décodées et comparer au débit de calcul de l'application.
  - *Temps total d'exécution du décodeur* ***(U)*** : pour savoir si l'application termine le décodage de la vidéo dans un temps raisonnable, nous mesurons son temps d'exécution. Cette valeur devrait être inférieure à la durée de la vidéo décodée.

5. Bibliothèque qui implémente un moteur de base de données relationnelle, sans serveur, accessible à l'aide du langage SQL :`http://www.sqlite.org/`

- *Observation du déploiement et de la communication dans l'intergiciel.*
  - *Déploiement des composants* ***(U)*** : afin d'observer l'exploitation des unités de calcul de la plate-forme, nous avons besoin de connaître la liste des composants instanciés à un moment donné. Pour cela, nous obtenons le *Processing Element* assigné par Comete à chaque composant.
  - *Temps de communication entre deux composants* ***(U)*** : nous sommes persuadés que le calcul du temps de communication entre composants peut nous indiquer si le déploiement a été correctement effectué. Nous pensons cela pour deux raisons : d'une part, le temps indique si la communication bénéficie d'accès rapide à la mémoire (cache). D'autre part, le temps de communication dans des composants du décodeur SVC peut être indicatif du taux d'utilisation du PE.
- *Observation de l'utilisation du système d'exploitation.*
  - *Évolution de la mémoire et du nombre des threads* ***(U)*** : pour connaître la quantité de mémoire requise par le décodeur, nous collectons les valeurs de la mémoire utilisée par Comete. Nous avons également suivi l'évolution des threads pour comprendre le fonctionnement de Comete qui crée un thread par PE défini.

Nous identifions les entités à observer suivantes : au niveau de l'application, il s'agit des composants applicatifs (cf. figure 8.4) ; au niveau Comete, nous nous intéressons aux composants d'allocation, de gestion (`ComponentManager`) et de chargement (`ComponentLoader`) ; au niveau système, nous observons le processus exécutant le décodeur SVC.

8.3.2 Identification des composants EMBera

Les composants d'observation EMBera sont les suivants (cf. figure 8.8) :

Composants pour l'observation du décodeur SVC : Comme nous avons modifié Cecilia pour implémenter le modèle EMBera et vu que SVC est développée à l'aide des composants Cecilia, nous nous servons des interfaces d'EMBera ajoutées aux composants pour l'observation. Pour observer, nous activons l'observation sur les composants de SVC, à l'aide de l'interface de contrôle. Nous avons considéré les composants `SvcTest`, `SvcDecoder`, `Slice`, `Parser`, `MotionCompensation`, `IntraPrediction`, `InverseTransformation` et `LoopFilter` pour l'observation de l'application, car ils exécutent la partie la plus importante du décodage. Pour sélectionner les composants, nous avons effectué une première exécution afin d'observer sur quels composants le décodeur passe la plupart de temps à l'exécution.

Composants pour l'observation de Comete : Nous nous servons des interfaces d'observation d'EMBera présentes dans les composants Cecilia, et par conséquent de Comete. Les interfaces d'observation produisent des données brutes dans les composants `ComponentLoader` et `ComponentManager`. Dans ce cas, nous connectons des composants

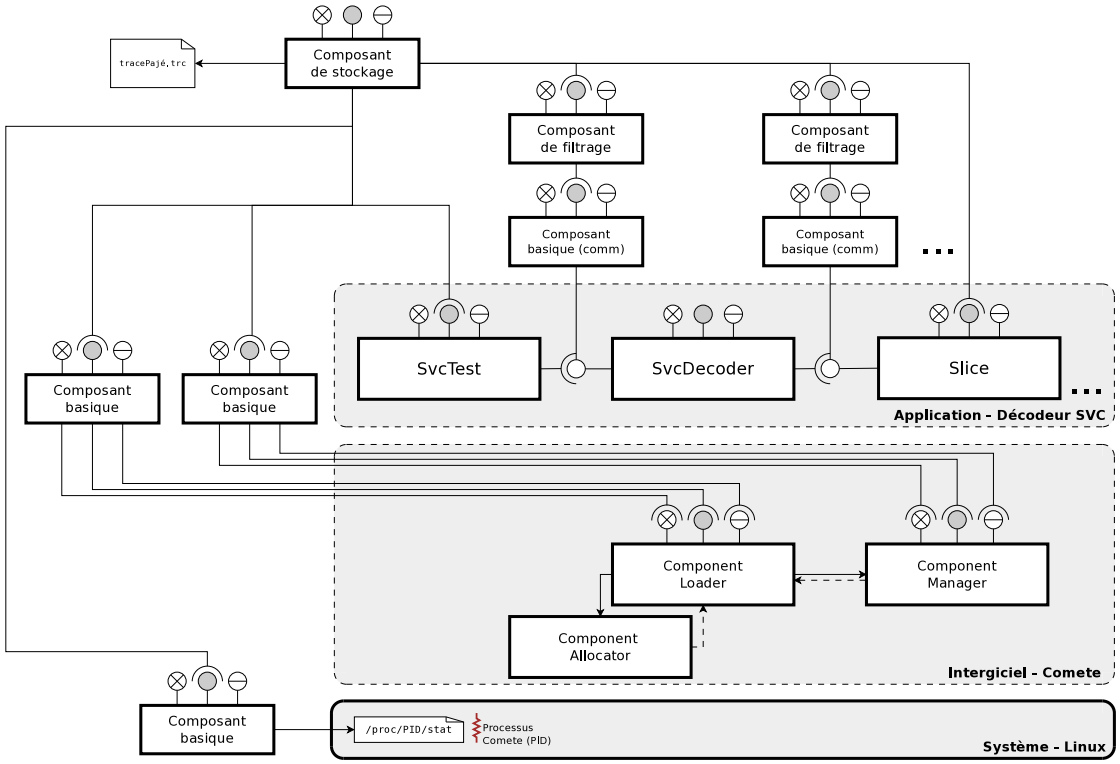

FIGURE 8.8 – Composants EMBera utilisés pour l'observation du décodeur SVC

basiques aux interfaces d'observation de ces deux composants, afin de régler la fréquence des observations.

Nous avons utilisé également des composants basiques pour l'observation de la communication entre composants de SVC. Des composants de filtrage ont été connectés à ces composants basiques afin de réduire la quantité des données à envoyer sur le composant de stockage.

Composants pour l'observation du système Linux : L'application SVC est exécutée à l'aide de Comete sur une machine à mémoire partagée, en créant un seul processus système et un ensemble de threads. Pour cette raison, nous utilisons un seul composant basique pour l'observation du processus crée, et pour le suivi de l'évolution des threads et de l'occupation mémoire.

8.3.3 Préparation du système pour l'observation

Vu que nous organisons l'observation selon trois niveaux logiques, nous avons besoin d'accéder aux observations de chacun des niveaux. C'est pourquoi nous devons effectuer des modifications sur l'application et l'intergiciel, ainsi qu'accéder aux informations du système d'exploitation. La manière dont nous avons modifié le logiciel et accédé aux observations est présentée ci-après.

Modifications effectuées à Cecilia

Dans la section 8.2, nous montrons que nous avons modifié le canevas Cecilia pour créer automatiquement des interfaces EMBera et contrôler l'observation. Vu que ce paramétrage offre déjà l'observation des composants du décodeur SVC, nous n'avons pas effectué des modifications additionnelles au canevas Cecilia pour l'intégration d'EMBera.

Modifications effectuées à Comete

Nous avons modifié l'intergiciel afin de récupérer des informations relatives au déploiement et à la communication des composants applicatifs.

Récupération des données : Nous pouvons obtenir des informations sur le déploiement des composants applicatifs disponibles en accédant aux interfaces d'observation du `ComponentManager`, qui communique avec le composant `ApplicationDB`. Nous ajoutons des méthodes à la définition de l'interface `ArchitectureDB` du dernier composant. Nous montrons dans la figure 8.9 un exemple de la définition de ces méthodes. Les méthodes ajoutées permettent d'obtenir la liste de composants instanciés (`getListOfInstances`) ainsi que leur déploiement (`getDeployment`).

```
package comete.generic.api;
import fractal.api.Component;
import fractal.api.Factory;

interface ArchitectureDB {
    ...
    int getListOfInstances(intptr_t * list);
    int getDeployment(intptr_t ** list);
    ...
}
```

FIGURE 8.9 – Méthodes ajoutées pour l'observation de Comete

Nous avons également besoin d'obtenir le temps entre l'appel d'une méthode, effectué par le composant appelant, et la fin de l'exécution de cette méthode. Cette information est collectée par le composant basique qui observe la communication. L'estampille de la fin de l'exécution de la méthode est obtenue à la sortie de celle-ci. Nous accédons à la pile d'appels de la méthode exécutée par le composant appelant, ceci nous permet d'identifier la méthode, l'interface et le composant qui ont fait l'appel. Puis nous relevons l'estampille au moment où l'évènement est posté dans la pile du composant `evtEngine` rattaché au PE où le composant appelé est exécuté. Nous soulignons que cette stratégie ne fonctionne que pour la communication basée sur la mémoire partagée.

Ce mécanisme a été mis en place, puisque la communication utilisée est asynchrone. Par conséquent, nous n'avons pas de retour de fonction dans le composant appelant. De plus, Comete ne fournissait pas à l'époque des moyens pour obtenir le temps d'exécution de méthodes sans contourner la logique à composants. Cependant, nous voyons dans la section 8.4 que l'intrusivité ajoutée est très importante.

Intégration des composants EMBera : Nous avons mis en place un mécanisme pour créer automatiquement les interfaces, les connexions, et le déploiement (pour l'observation de la communication) liés à l'observation.

En effet, nous définissons les composants d'observation dans un fichier XML, qui suit une syntaxe très proche à celui des fichiers ADL de Comete (cf. figure 8.10). Le fichier est pris en compte par Comete dans le composant `ComponentLoader`, après le déploiement et l'interconnexion des composants de l'application, mais avant leur lancement. Dans le fichier XML (cf. figure 8.10(a), nous définissons le type (`type`) et le nom (`name`) du composant, le PE (`pe`) sur lequel le composant est déployé, ainsi que le composant auquel il fournit les données observées (`out`).

```
    <component type={type} name={nom} out={nom du composant en sortie} pe={identifiant du PE}
(a)            observed={nom du composant observé}  <!-- Appliquent seulement aux basiques -->
               interface={nom de l'interface}/>     <!--     observant la communication    -->
-------------------------------------------------------------------------------------------
    <observedComponents>
(b)     <component observed={nom du composant de l'application}
                   observer={nom du composant basique}
                   out={nom du composant en sortie}/>
        ...
    </observedComponents>
```

FIGURE 8.10 – Définition des composants EMBera dans Comete

Afin de connecter et déployer les composants d'observation, les composants EMBera doivent être préalablement compilés. Pour les composants basiques qui observent la communication, nous les déployons dans le même PE que le composant (`observed`) qui possède l'interface fournie (`interface`) impliquée dans la communication observée.

Pour connecter les composants EMBera aux composants de l'application, les composants du décodeur doivent étendre la définition d'observation EMBera, puis définir la connexion. Cette définition est faite dans la section `<observedComponent>` du même fichier de configuration XML d'EMBera (cf. figure 8.10(b)).

Accès aux observations du système Linux

Nous n'introduisons pas de modifications au système pour son observation. Au lieu de modifier le code, nous utilisons les données sur le processus fournies par le répertoire virtuel de fichiers `/proc/PID`, dans les fichiers `stat` et `mem`. Nous n'avons pas obtenu des informations par thread depuis le répertoire `/proc/PID/task/TID` car d'une part, un seul thread peut être le support d'exécution de plusieurs composants, par conséquent, cela n'indique pas les ressources utilisées par composant. D'autre part, l'information donnée par `/proc/PID/task/TID` n'est pas très précise.

8.3.4 Spécialisation du modèle d'observation pour la plate-forme

Notre modification au niveau Cecilia nous permet d'observer le décodeur SVC sans modifier son code source. La spécialisation consiste à configurer les composants afin

d'observer les métriques que nous avons choisies.

Composant basique : L'observation des composants de l'application est faite en activant l'observation avec EMBera. L'activation est faite à l'aide de l'interface de contrôle. Vu que cette activation permet la production de données, nous connectons directement les interfaces spécialisées aux composants de filtrage et de stockage.

L'intergiciel est observé, en connectant des composants basiques indépendants aux interfaces EMBera du `ComponentManager`. Pour observer la communication, nous connectons un composant basique au composant `evtEngine` de chaque connexion que nous observons. La taille du tampon de ce dernier n'a pas été borné pour deux raisons, d'une part cette observation produit une quantité très importante d'évènements et la remise à zéro périodique du tampon augmente l'intrusivité. D'autre part, nous avons estimé l'espace mémoire requis pour l'enregistrement temporaire des toutes les évènements lors d'une exécution du décodeur, puis constaté que la machine possède suffisamment de mémoire vive pour cet enregistrement.

Composant de filtrage : Nous avons configuré les composants de filtrage, connectés aux basiques qui observent la communication, pour filtrer les appels d'une durée inférieure à 15 ms. Cette valeur a été choisie après plusieurs observations. Nous avons vu que les appels qui durent moins que 15 ms correspondent la plupart à des messages de configuration qui ne sont pas d'intérêt pour nos observations.

Composant de stockage : Nous avons configuré le composant de stockage pour l'enregistrement des données sur un fichier de trace, en utilisant le support pour GTG. Nous avons fait ce choix principalement parce que nous avons besoin d'obtenir les observations dans un fichier de trace prêt pour la visualisation dans des outils comme `Vite`[6] ou Pajé [dKdOS00].

8.3.5 Instanciation et déploiement des composants EMBera

Pour instancier les composants EMBera, nous devons d'abord instancier les composants du décodeur. Les composants du décodeur doivent être définis dans Cecilia en étendant l'observation minimaliste fournie EMBera. Puis nous définissons le déploiement de l'application avec un fichier de configuration de Comete.

Pour instancier les composants d'observation, nous utilisons un fichier XML, qui suit la structure présentée dans la figure 8.10. Nous montrons dans la figure 8.11 un extrait de ce XML pour l'instanciation et le déploiement des composants EMBera : les composants `Basic_System` et `Basic_Comete` sont respectivement connectés à des composants de stockage et de filtrage. De même, nous voyons la définition du composant `Basic_Communication_Slice`, qui observe a communication entre les composants `Slice` et `MotionCompensation`.

6. Outil d'exploration et de visualisation des traces sous formats Pajé et OTF [INR10b]

```
<component type="observation_Basic_System" name="Basic_System" out="StorageGTG" pe="0" />
<component type="observation_Basic_Comete" name="Basic_Comete" out="Filter" pe="0" />
<component type="observation_obsFilterComponent" name="FilterComponent0"
           out="StorageGTG" pe="5" />
...
<component type="observation_Basic_Communication" name="Basic_Communication_Slice"
           out="StorageGTG" observed="Slice" interface="manageSlCmpFb" />
```

FIGURE 8.11 – Fichier XML de configuration des composants d'observation

Suite à la définition, nous lançons l'exécution de SVC par le biais d'une procédure de démarrage de Comete. Cette procédure crée un processus Linux et autant des threads que de *Processing Elements*. Nous présentons dans la figure 8.12, les composants que nous avons utilisé pour l'application SVC et son observation, déployés sur les PEs correspondants. Nous rappelons que nous avons utilisé le processeur #0 de la machine, c'est-à-dire que les PEs de la figure représentent un cœur indépendant chacun.

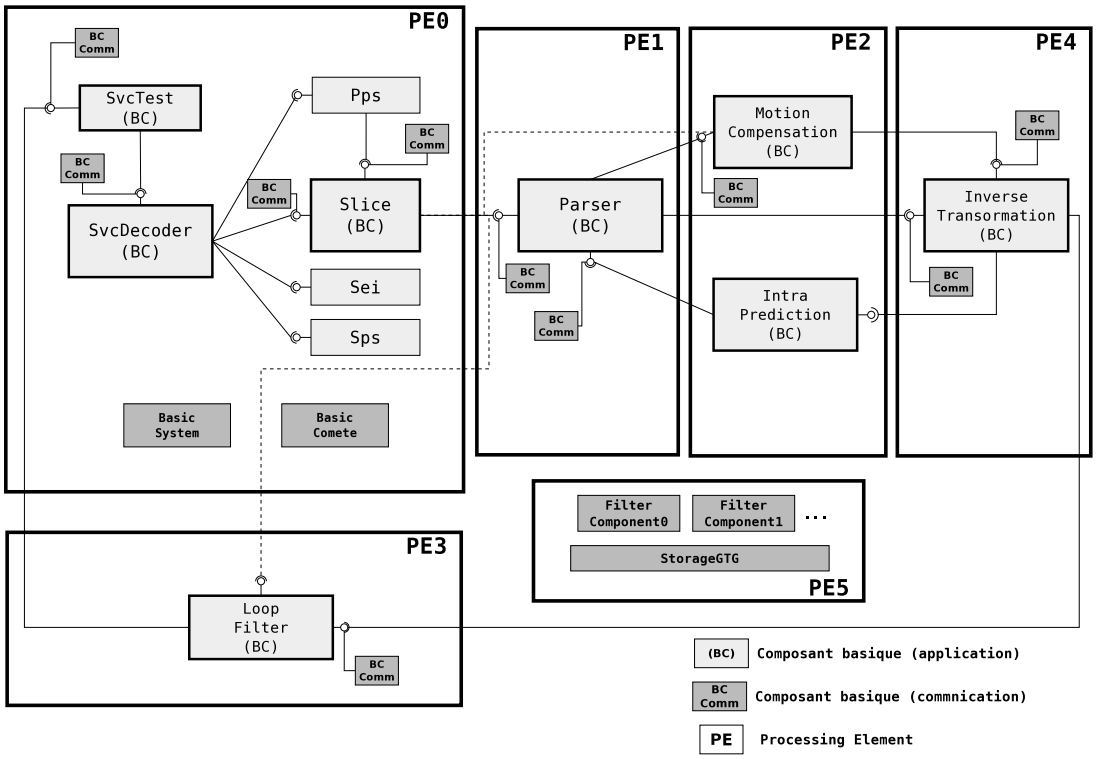

FIGURE 8.12 – Déploiement des composants SVC et EMBera

8.4 Observations effectuées et problèmes détectés

Compte tenu que cette version du décodeur SVC n'avait pas été testé auparavant sur une machine à plus de deux cœurs, nous nous servons des observations pour évaluer la performance de SVC. Les observations nous permettent aussi d'évaluer la pertinence du

déploiement choisi pour l'application sur la plate-forme multi-cœur.

Les observations sont obtenues et présentées selon trois niveaux logiques d'abstraction, notamment : l'application, l'intergiciel et le système. À la fin de la section, nous effectuons une corrélation entre les observations obtenues.

Nous avons expérimenté avec deux configurations du modèle d'observation. La première est moins intrusive car elle n'observe pas la communication. Elle rend possible l'observation du décodage des trames et le temps d'exécution. Du point de vue système, elle permet d'observer les ressources utilisées par le support d'exécution utilisé. La deuxième configuration, plus intrusive à cause des composants basiques observant les interactions dans Comete, sert à observer la communication entre les composants de l'application et à obtenir leur temps d'exécution.

Nos expérimentations ont été effectuées sur une séquence vidéo d'une durée de 362 s. Elle est composée de 9 050 *images*, chacune d'une taille de 896×512 *pixels*. La vidéo, contenue dans un fichier, contient information de deux changements des paramètres de décodage.

Les observations ont été effectuées en décodant cette vidéo dix fois pour chacune des configurations de l'observation. Nous considérons que ce nombre d'exécutions suffit, d'une part, parce que l'écart du temps total entre les exécutions est de moins de 1%, pour les observations moins intrusives et d'environ 2% les autres. D'autre part, toutes ces mesures ont été prises disposant d'un accès exclusif à la machine pendant les expérimentations. De plus, une fois sélectionné le déploiement, nous avons utilisé toujours les mêmes cœurs de la machine pour l'exécution des composants.

Enfin, nous avons utilisé le logiciel `Vite`, pour la visualisation des traces dans format Pajé.

8.4.1 Observations au niveau de l'application

Nous avons utilisé les composants EMBera pour observer des paramètres liés à la qualité du rendu de l'application. Dans ce cas, nous avons observé le nombre d'images décodées, ainsi que le temps de décodage de chacune des images.

Nous avons calculé le temps total de décodage en utilisant la première configuration pour l'observation (sans composants basiques observant la communication). Vu que l'application ne garantit pas le temps réel, le temps total d'exécution est rallongé par les traitements effectués par EMBera. La table 8.1 présente le temps de référence du décodage, ainsi que les temps d'exécution de SVC obtenus sur la plate-forme. Nous clarifions que ces temps correspondent à une exécution qui utilise le déploiement montré dans la figure 8.12. Nous avons sélectionné ce déploiement parce qu'il a donné le temps d'exécution le plus court du décodeur, parmi d'autres configurations essayées.

Dans la table, le temps de *référence* est le temps requis pour décoder et afficher 9 050 *images* à une vitesse de 25 *images*/s. Le temps *non-observé* correspond à une exécution sans composants EMBera. Les temps *sans* et *avec observation* s'obtiennent

Exécution	Temps (s)
Temps de référence	362,00
Non-observée	210,26
Sans observation de la communication	221,78
Avec observation de la communication	2 874,00

TABLE 8.1 – Temps d'exécution du décodeur SVC

des observations sur l'exécution avec les deux configurations des composants EMBera.

Nous avons mesuré un temps de $221,78\ s$, si nous observons l'exécution sans regarder la communication. Nous calculons que le temps de décodage par image est de 25 ms, c'est-à-dire, que l'application peut décoder environ 40 *images* par seconde lors d'une exécution observée avec EMBera.

L'observation du décodage des images et les valeurs de la table nous permettent d'évaluer si, sur la plate-forme multi-cœur, le décodeur SVC peut atteindre des performances qui donnent la qualité requise des images et qui respectent des contraintes temporelles. Grâce aux données observées et enregistrées dans la trace, nous voyons que toutes les 9 050 *images* ont été correctement décodées. À partir de l'analyse des données de la table 8.1, nous constatons que le temps de décodage par image est inférieur aux 40 ms requises pour respecter le temps réel. Nous concluons que l'application SVC peut attendre ses performances, sur cette plate-forme, même si nous observons le décodeur avec EMBera.

8.4.2 Observations au niveau de l'intergiciel

Nous utilisons la deuxième configuration des composants EMBera, qui fournit des observations détaillées sur les appels de méthodes entre composants. Nous soulignons que nous observons les temps de communication en reconstruisant le comportement de l'application. Pour cela, nous observons la chaîne de traitement des messages dans Comete, depuis l'envoi jusqu'à l'exécution de la méthode appelée.

Ci-dessus, nous montrons que l'observation de la communication sert à guider le déploiement du décodeur SVC. Cette observation permet d'obtenir les temps d'exécution effectifs des composants, ainsi que la fréquence des appels entre eux. Enfin, nous introduisons la discussion sur la perturbation, induite par l'observation sur la qualité du décodage, qui est approfondie dans la section 9.2.3.3.

Sélection du déploiement basé sur le temps effectif d'utilisation des PEs

Nous présentons dans la figure 8.13, une vue de l'exécution des composants déployés par Comete, par le biais du logiciel `Vite`. Les flèches qui montrent les appels entre les composants ont été supprimées de la figure afin de permettre la visualisation de l'exécution des méthodes des composants.

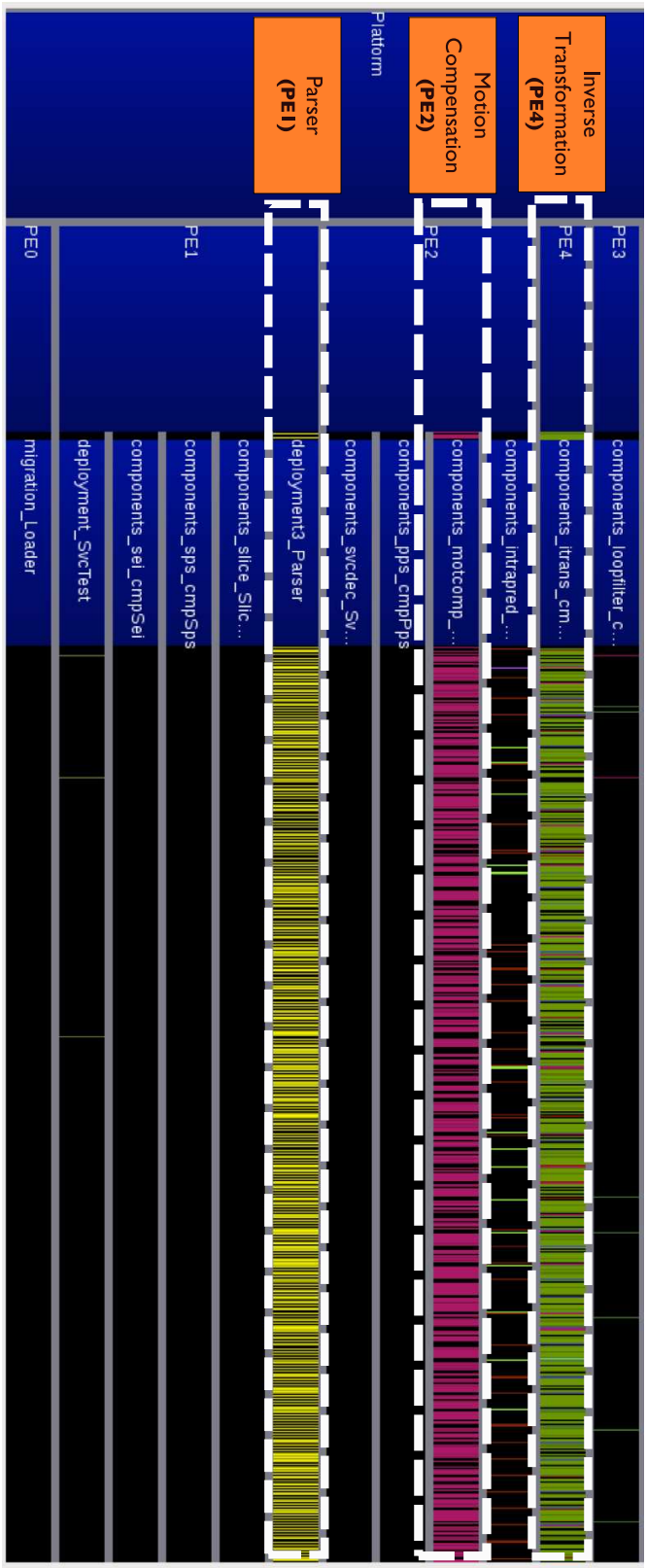

FIGURE 8.13 – Exécution des composants de SVC déployés sur les PEs définis dans Comete

Dans la figure nous voyons l'exécution des composants du décodeur à l'aide de Vite. Les lignes horizontales montrent les instants, quand les composants s'exécutent suite à un appel depuis un autre composant. Nous constatons visuellement que le décodeur utilise principalement les composants `InverseTransformation`, `MotionCompensation` et `Parser`. Afin de connaitre l'usage effective composants, nous calculons leurs pourcentages d'utilisation des PEs que les composants utilisent. Nous présentons ces pourcentages, dans la table 8.2.

Composant	PE	% d'utilisation
`SVCTest`	0	0,7
`SVCDecoder`	0	0,1
`Slice`	0	0,2
`Parser`	1	58,1
`Motion Compensation`	2	65,2
`Intra Prediction`	2	11,0
`Loop Filter`	3	1,2
`Inverse Transformation`	4	80,6

TABLE 8.2 – Pourcentages d'utilisation des PEs par les composants SVC

Ces pourcentages nous donnent des pistes pour un déploiement plus approprié des composants sur la machine. Nous avons mentionné précédemment que les observations ont été obtenues sur le meilleur déploiement essayé, suite aux observations du décodeur avec EMBera. Nous avons effectué un premier déploiement où les composants `Inverse-Transformation` et `MotionCompensation` ont été exécutes sur le même PE. Le temps d'exécution de l'application non observée a été de $270,8\ s$, environ 28,8% plus lente que avec la configuration sélectionnée, après une observation détaillée.

Sélection du déploiement basé sur la fréquence d'appels

Vu que le décodeur réutilise des données pour le décodage des images du même groupe, deux composants échangeant une quantité importante de messages doivent être déployés dans des cœurs qui partagent la mémoire vive ou la mémoire cache. De même, si après l'observation, nous constatons que les traitements effectués par un composant de l'application ne bloquent pas ceux d'autre composant, ils doivent être exécutés en parallèle par des PEs indépendants.

Pour montrer l'utilité d'observer la communication, nous avons modifié le déploiement des composants. Ce déploiement vise à « éloigner »les composants qui communiquent le plus pour observer l'impact sur l'exécution. Nous rappelons que la plate-forme d'exécution est composée de 4×6 cœurs. Le déploiement est : `IntraPrediction`, processeur `#0` ; `InverseTransformation`, processeur `#1` ; `Parser` et `LoopFilter`, processeur `#2`. Le temps d'exécution du décodeur non-observé est de $428,86\ s$ (environ 48 ms pour décoder une image). Nous constatons que le nouveau déploiement a augmenté le temps d'exécution d'environ 104%. Cette exécution dépasse le temps de référence, même sur cette plate-

forme, qui dispose d'une puissance de calcul assez importante. Nous concluons qu'EMBera a permis de prouver qu'un déploiement qui implique plusieurs processeurs augmente le temps de communication, ainsi que de trouver un déploiement plus adéquat pour les composants du décodeur, en fonction des caractéristiques de la plate-forme.

Perturbation

Malheureusement, l'observation de la communication des composants du décodeur, même si elle ne perturbe pas la qualité individuelle des images rendues, réduit sensiblement le nombre d'images décodées à 3 $images/s$. Cette dégradation du nombre d'images est due au grand nombre de données collectées. Dans ce cas, l'exécution de SVC produit des millions d'appels d'une durée très courte et l'observation d'un appel prend plus de temps que l'appel. Si nous évaluons cette intrusivité seulement en termes de ralentissement du temps de décodage, elle est inacceptable. Par contre, la valeur ajoutée des informations obtenues est incontestable. Toutefois, nous envisageons que ce type d'observation est à utiliser uniquement pendant l'étape de sélection du déploiement des composants de SVC sur la plate-forme, mais pas pour des exécutions courantes du décodeur.

8.4.3 Observations au niveau du système

Les mesures au niveau système portent sur l'évolution de l'utilisation des threads et de la mémoire. Pour ces observations, nous avons utilisé la première configuration des composants EMBera, qui n'observe pas la communication.

Nous avons constaté que le nombre de threads est fixé au début d'exécution de Comete et correspond à deux threads par PE, un pour `ComponentManager` et un pour `ComponentLoader`.

Par rapport à la mémoire, nous constatons une évolution (figure 8.14). Nous observons deux augmentations dans la quantité de mémoire utilisée : l'une vers 35 s (1) et l'autre vers 63 s (2). Elles correspondent aux reconfigurations de la vidéo en entrée.

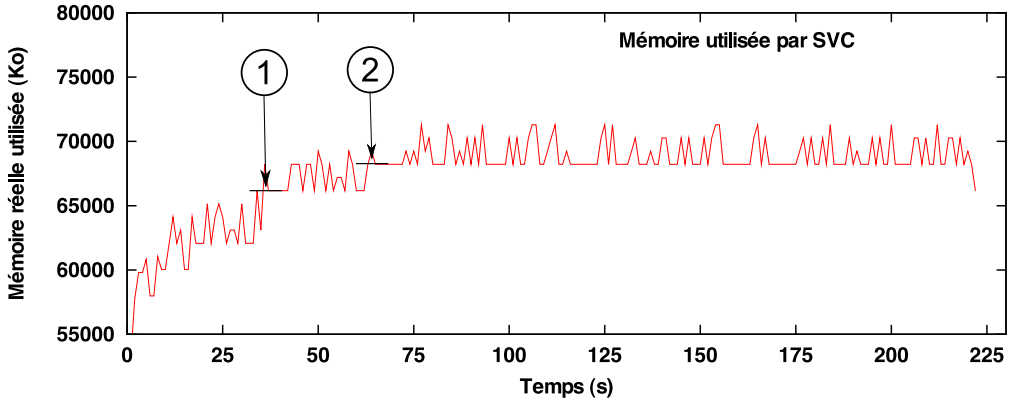

FIGURE 8.14 – Évolution de l'utilisation mémoire de l'application SVC

À partir des données obtenues, nous n'observons pas des fuites de mémoire lors de l'exécution, mais des variations de la mémoire utilisée, si les paramètres de la vidéo changent. Ces observations sont très importantes pour le dimensionnement de la mémoire des plates-formes embarquées ainsi que pour la gestion des blocs de mémoire, si la plate-forme le permet. Des propositions sur ce dernier ont été proposées dans [DLLKK06] et [DGMB07].

8.4.4 Vue d'ensemble

Nous corrélons les observations en nous basant sur les estampilles. La figure 8.15 présente la corrélation entre le décodage d'une image de la vidéo, l'exécution des composants et l'évolution de la mémoire utilisée.

Nous voyons dans la figure qu'au début du décodage de l'image, le composant `Parser` exécute une méthode qui effectue des appels répétitifs vers `IntraPrediction` (1), qui en conséquence démarre une communication continue avec `InverseTransformation` (2). Ces deux derniers exécutent des appels au composant `LoopFilter` (3), qui prépare les données pour les composants chargés de la sortie des données. Dans ce cas, nous voyons que l'observation détaillée nous présente une opportunité d'optimisation : si les composants `IntraPrediction` et `InverseTransformation` s'exécutent sur des cœurs qui partagent leur niveau `L2` de cache le temps de communication peut être réduit.

La suite de l'exécution est reprise par le composant `Parser`, qui communique avec les autres composants de l'application (`SVCDecoder`, `Pps`, `MotionCompensation`, etc.). Enfin, les résultats sont envoyés vers `LoopFilter` (4), qui les achemine vers la sortie.

La mémoire de l'application reste constante, ce qui nous indique que les paramètres d'affichage de la vidéo restent inchangés, pendant la période montrée.

L'analyse simultanée des trois niveaux d'observation du décodeur SVC permet, d'une part, de comprendre le fonctionnement de l'application. D'autre part, l'observation donne des indications sur les possibilités d'exploitation plus performante de la plate-forme.

8.5 Synthèse

Dans cette étude de cas, les observations ont été surtout focalisées sur des aspects applicatifs et de l'intergiciel. En effet l'application présente des comportements complexes, qui ne peuvent pas être analysés uniquement du point de vue système.

Les observations que nous avons effectuées sur le décodeur SVC, nous ont permis de comprendre des détails du fonctionnement, difficilement observables autrement, même en ayant accès au code source de l'application. Par exemple, à partir du temps d'exécution des composants, nous avons observé des composants de SVC qui ont besoin du processeur en permanence. En outre, nous avons observé qu'un accès plus rapide à la mémoire réduit le temps d'exécution des composants qui communiquent très fréquemment. En conséquence, nous avons effectué des modifications sur le déploiement des composants de

SVC pour améliorer la performance du décodeur, par exemple, en mettant les composants plus gourmands, en termes de processeur, sur des cœurs indépendants.

Nous avons constaté que la plate-forme a la performance requise pour exécuter le décodeur et l'observer (application et système), dans un temps d'exécution plus court que le temps de référence calculé pour le décodage. Les observations de l'intergiciel, basées sur les observations des messages échangés, ajoutent une intrusion élevée, mais donnent des informations très riches qui contribuent à l'amélioration de la performance de l'application.

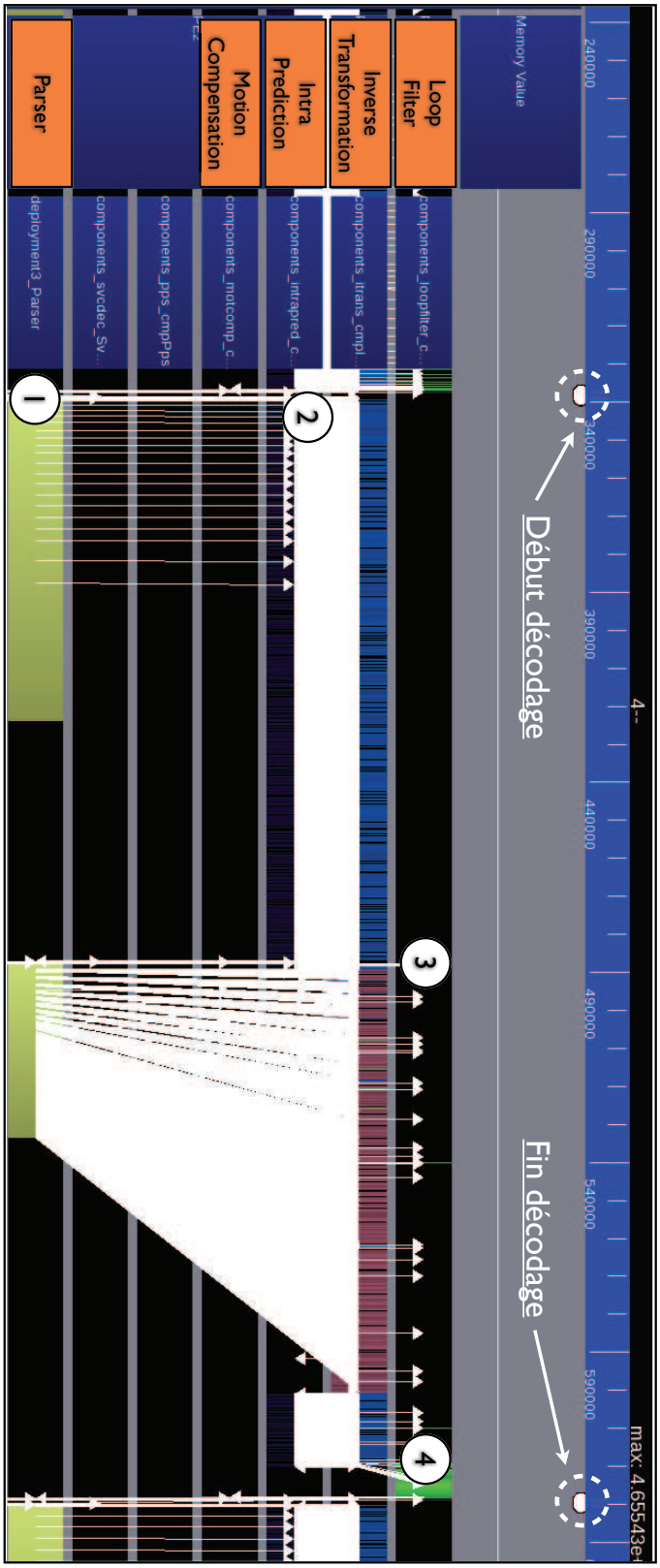

FIGURE 8.15 – Décodage d'une image dans SVC

Chapitre 9

Évaluation générale des résultats

Dans ce chapitre, nous introduisons les critères d'évaluation d'EMBera et ensuite les appliquons aux études présentées dans les chapitres précédentes.

9.1 Critères d'évaluation

Nous avons défini trois critères d'évaluation qui nous permettent de montrer la pertinence du modèle EMBera pour l'observation des plates-formes embarquées. Nous discutons sa valeur ajoutée, sa facilité d'intégration au système sous, observation ainsi que son intrusivité.

9.1.1 Valeur ajoutée des observations

Nous voulons montrer que, comparé aux outils d'observation existants, EMBera permet des observations qui sont plus complètes et plus facilement configurables. Les observations contribuent principalement à la correction d'erreurs (p. ex. détection des fuites de mémoire) et l'amélioration de la performance (p. ex. réduction des temps d'exécution ou augmentation du nombre d'images décodées).

Pour démontrer cette valeur ajoutée, nous discutons les besoins en observations et la manière dont ils peuvent être satisfaits sans EMBera. Puis nous comparons les approches et discutons les avantages de l'observation avec EMBera.

9.1.2 Simplicité d'intégration

Nous évaluons la démarche suivie pour intégrer notre modèle d'observation à la plate-forme sous observation. L'intégration est considérée comme simple si elle n'induit pas des modifications importantes sur la plate-forme. Nous discutons de la minimisation de l'instrumentation du code source et de l'utilisation d'informations déjà disponibles au sein du système ou à travers des outils de la chaîne de développement.

Regardant les techniques, nous parlons soit de la manière dont la plate-forme ciblée a dû être modifiée (instrumentation des sources), soit de l'utilisation des mécanismes existants (utilisation des informations fournies par le système d'exploitation). La technique peut comprendre aussi des options de compilation qui activent la production des informations de débogage et de profilage.

9.1.3 Intrusivité

Vu que l'observation des systèmes informatiques peut modifier leur comportement, avec une incidence majeure sur les plates-formes embarquées, nous considérons essentiel l'évaluation de l'intrusion ajoutée. L'intrusion peut changer le comportement du système principalement de deux manières : elle peut induire le système à transgresser les spécifications fonctionnelles ou elle peut perturber les performances du système.

Dans notre évaluation nous ciblons principalement les performances temporelles du système, en d'autres termes, son ralentissement dû à l'observation. Toutefois, dans le cas de systèmes temps réel, le ralentissement se traduit par une qualité de fonctionnement dégradée.

D'autre part, vu que l'intrusion est due en partie à la manque de ressources, nous considérons que des ressources telles que la mémoire requise pour supporter l'observation doivent être prises en compte comme partie de l'intrusion. L'estimation de la taille de cette mémoire peut servir au dimensionnement d'un support pour l'observation, à ajouter sur les plates-formes embarquées. Cette mémoire est d'une part la mémoire vive pour le stockage des tampons et, d'autre part, l'espace requis dans le support persistant pour le stockage de l'historique.

9.2 Évaluation des études de cas

Dans la suite du chapitre, nous présentons l'évaluation de notre modèle EMBera, sur les trois études de cas, effectuée par le biais des critères définis. Nous cherchons à positionner notre travail par rapport à d'autres techniques disponibles pour observer les plates-formes des études de cas. Nous discutons dans quels cas le modèle EMBera offre une alternative plus avantageuse pour l'observation mais aussi quand EMBera présente les mêmes prestations que les autres approches. Nous présentons d'abord un court rappel de chaque étude de cas :

- *Décodeur MJPEG* : l'application, exécutée sur un MPSoC à 5 processeurs, a été programmée à l'aide de composants EMBera pour être observée. Le décodeur comporte trois fonctionnalités majeures : (1) la lecture du flux et le traitement initial des images (`Fetch`) ; (2) le calcul de la fonction principale du décodage (`IDCT`) ; (3) la reconstitution d'images et l'envoi vers une sortie (`Reorder`).
  Nous avons voulu observer ce décodeur pour constater que les choix de déploiement ont été adéquats, ainsi que l'utilisation de la mémoire est optimale.

- *Application et intergiciel multimédia* : l'application, exécutée sur un MPSoC à 3 processeurs, utilise un intergiciel qui lui fournit l'accès aux fonctions de décodage et aux processeurs de la plate-forme. La plate-forme utilise les systèmes d'exploitation STLinux [STM09b], pour le processeur maître, et un RTOS, pour les accélérateurs. La communication entre processeurs et l'accès au RTOS sont abstraits par l'intergiciel.
  Nous nous sommes intéressés à observer si le décodeur donne la qualité attendue pour le traitement des trames vidéo. De même, nous avons voulu observer l'utilisation de la mémoire et l'évolution des threads de l'application.
- *Décodeur SVC* : la particularité de cette étude de cas est que l'application et l'intergiciel (Comete) sont organisés en composants logiciels (Cecilia). L'application, exécutée sur une plate-forme multi-cœur SMP, effectue une suite de traitements comportant la lecture de fichiers, le découpage, le décodage et la restitution d'images. Comete fournit une abstraction de la plate-forme matérielle et facilite le déploiement des composants applicatifs.
  Nous avons principalement observé le décodeur SVC afin de constater qu'elle effectue correctement le décodage vidéo et aider à la compréhension du fonctionnement du décodeur.

9.2.1 Valeur ajoutée des observations

Nous discutons, ci-après, la valeur ajoutée des observations effectuées dans chacune des études de cas. Nous discutons la méthode qu'il aurait fallu utiliser pour observer sans EMBera ainsi que l'intérêt des observations.

9.2.1.1 Décodeur MJPEG

Pour obtenir les mêmes informations, si nous n'utilisons pas EMBera pour l'observation du décodeur, nous aurions dû utiliser des outils permettant de :

- *Collecter les observations du processeur maître* : il aurait fallu instrumenter explicitement le code pour invoquer les fonctions fournies par le RTOS, qui donnent des informations sur le temps d'exécution et l'utilisation de la mémoire.
- *Collecter les observations des accélérateurs* : comme dans le précédent, mais il aurait fallu programmer le renvoi des informations vers le processeur principal. Pour renvoyer les données, il aurait fallu utiliser directement les primitives de communication de la plate-forme embarquée dans le code de l'application.
- *Observer la communication* : il aurait fallu instrumenter les appels aux fonctions de communication. L'instrumentation aurait été faite soit dans le code de l'application lors de l'appel, soit dans la bibliothèque de communication. Cette dernière demande une connaissance profonde des fonctions de communication et de la plate-forme.
- *Acheminer les données collectées vers un support persistant unique* : il aurait fallu envoyer les données vers une machine de développent. Pour cela, nous aurions

utilisé des outils de débogage à l'aide d'un port JTAG. Les appels aux fonctions du débogueur et d'accès au port auraient dû être ajoutés au code source du décodeur.

Nous constatons qu'il est possible d'obtenir les mêmes informations par d'autres moyens. Cependant, cette alternative d'observation exige l'instrumentation du logiciel à chaque fois qu'une nouvelle application est observée. De plus, si nous n'utilisons pas EMBera, il aurait fallu utiliser plusieurs outils différents.

L'observation a permis l'identification de problèmes et des opportunités d'amélioration. En observant le décodeur MJPEG, nous avons constaté des fuites de mémoire, car l'application effectue de plus en plus des allocations qui ne sont jamais libérées (cf. figure 6.9(b)). En tant qu'opportunités d'amélioration, nous avons identifié que les fonctionnalités fournies par le composant `Fetch-Reorder` doivent être traitées par des composants indépendants. En effet, nous avons programmé initialement ces fonctionnalités dans un seul composant pour limiter la création de tâches, mais nous avons observé que le composant prend beaucoup de temps à s'exécuter. En outre, nous constatons l'avantage, en temps total d'exécution, de communiquer plus fréquemment vers les processeurs accélérateurs en envoyant des messages de petite taille (cf. figure 6.10).

Par ailleurs, cette étude de cas nous a servi à l'identification des niveaux d'observation ainsi qu'à une définition plus précise des interfaces spécialises.

9.2.1.2 Intergiciel multimédia STMicroelectronics

Pour observer l'intergiciel multimédia STMicroelectronics sans EMBera, il aurait fallu disposer d'outils pour :

- *Collecter des données de l'intergiciel et de l'application* : il aurait fallu instrumenter les fonctions de l'intergiciel, effectuant la dernière étape du décodage audio et vidéo ainsi que de l'application, recevant les commandes utilisateur. Les données observées auraient été produites soit en instrumentant le code source des fonctions, soit en activant des options de profilage à la compilation. Puis les données auraient été écrites dans un historique d'exécution.
- *Obtenir des échantillons de l'utilisation de la mémoire et les threads* : il aurait fallu récupérer l'identifiant du processus de l'application et implémenter un programme qui interrogerait le système d'exploitation pour récupérer les valeurs de la mémoire et les threads, entre le début et la fin de l'exécution du processus. Les données obtenues auraient dû être écrites dans un historique sur le système de fichiers de la plate-forme.
- *Identifier les défauts de décodage* : il aurait fallu implémenter des algorithmes pour corréler le succès du décodage de l'audio et de la vidéo, qui traitent les données observées sur les fonctions de décodage.

Pour obtenir les mêmes observations fournies par EMBera, il aurait fallu implémenter tout le mécanisme à la main pour chaque application observée. D'ailleurs, nous voyons

que l'alternative pour remplacer EMBera est basée sur l'instrumentation du code source. Cette pratique n'est pas largement acceptée pour l'observation d'applications embarquées, entre autres raisons, parce que les optimisations effectuées par le compilateur peuvent être perturbées en introduisant des nouvelles instructions.

EMBera nous a permis de détecter des fuites de mémoire entre deux exécutions du décodeur, pendant une utilisation prolongée de l'application (utilisation quotidienne de la plate-forme). De plus, nous avons constaté que si le flux vidéo contient des trames corrompues successives, le décodeur alloue de plus en plus de mémoire. Ces allocations successives ne sont pas bornées et peuvent produire le remplissage total de la mémoire de la plate-forme.

D'autre part, les observations faites à l'aide d'EMBera ont servi à mieux comprendre le fonctionnement de l'application. En effet, nous avons déduit la chaîne des traitements de la vidéo dans l'intergiciel dont le code est très volumineux pour l'analyser manuellement (environ 2 millions de lignes de code).

9.2.1.3 Décodeur SVC

Ci-après, nous listons les mécanismes nécessaires qu'il aurait fallu mettre en place afin d'observer le décodeur SVC.

- *Génération des données des composants* : il aurait fallu définir des interfaces pour fournir des observations et implémenter leurs méthodes pour produire les données. Ces modifications auraient été ajoutées manuellement à chaque composant de l'application ou à Cecilia pour qu'il ajoute automatiquement les interfaces et leurs méthodes requises pour une observation. La modification de Cecilia exige une bonne connaissance de la chaîne de transformations effectuée par ce dernier.
- *Génération des données de déploiement et de communication* : il aurait fallu instrumenter l'intergiciel Comete. Pour l'observation du déploiement, nous aurions affecté le composant `ComponentLoader`. Pour la communication, nous aurions instrumenté tous les composants de la chaîne de communication du protocole *SHMCBST* (cf. figure 3.2), ou utilisé un mécanisme d'observation de la pile d'appels, sans instrumenter le code de Comete.
- *Collection d'information sur l'évolution de la mémoire* : il aurait fallu implémenter un programme qui intercepte l'identifiant du processus lancé par Comete et interroge le mécanisme système, fournissant l'information de l'évolution de la mémoire.

Si nous n'utilisons pas EMBera, nous pouvons observer le décodeur SVC, mais la solution implémentée reste *ad-hoc*. Dans notre implémentation d'EMBera, nous avons modifié Cecilia pour générer automatiquement des composants observables. Cela permet aux composants d'être observés en leur donnant un paramètre lors de leur définition, mais sans avoir besoin d'instrumentations supplémentaires. De plus, l'utilisateur peut définir des observations différentes de celles fournies par défaut.

Nous avons observé la communication de Comete en accédant à la pile d'appels des méthodes. Ainsi, nous observons mais en préservant la logique de composants, sans établir artificiellement d'autres canaux de communication. Nous considérons que les observations obtenues sur l'interaction et le temps effectif d'exécution des composants est très utile et elle est difficilement disponible autrement. Cette information sert principalement à identifier quels composants doivent s'exécuter en parallèle pour améliorer le temps total de décodage.

L'observation de Comete a contribué à effectuer un déploiement plus adéquat du décodeur SVC. En effet, l'observation de la communication nous a suggéré que les composants, qui échangent beaucoup des données, doivent être confinés dans des cœurs d'un même processeur, de préférence en partageant le cache.

L'utilisation des composants de filtrage a réduit substantiellement le nombre d'observations à traiter, à acheminer et à enregistrer. Dans notre cas, nous avons instancié des composants de filtrage, configurés pour filtrer des données observées de communication ayant un temps inférieur à 15 ms (temps observé pour les appels de synchronisation entre composants). Grâce à ce filtre, nous avons réduit d'un 60% les données à enregistrer.

Enfin, la corrélation a permis de dévoiler les interactions de composants entre le début et la fin du traitement d'une image. L'observation donne un moyen pour comprendre le décodeur SVC, un décodeur qui a des interactions complexes et un nombre important des lignes de code (environ 50 000).

Discussion

En ce qui concerne les observations, EMBera nous a permis d'obtenir, d'une manière simple, des informations depuis trois niveaux logiques et de les traiter en les filtrant ou les agrégeant. Nous avons observé, entre autres, la qualité de décodage au niveau applicatif, la pertinence du déploiement, au niveau intergiciel, et les fuites de mémoire au niveau système. Dans chacun des cas, les observations ont dévoilé des problèmes et des opportunités de amélioration.

Nous avons proposé une alternative pour observer chacune des études de cas, sans utiliser EMBera. Nous constatons que, sauf dans l'observation du décodeur SVC, les alternatives ont permis d'obtenir les mêmes données brutes. Cependant, nous voyons que pour obtenir ces observations, nous devons effectuer une implémentation considérable à chaque fois. De plus, ces implémentations impliquent l'instrumentation du code du logiciel observé qui, comme nous avons discuté, est une pratique à éviter dans l'observation des MPSoC.

Nous considérons qu'EMBera est une approche intéressante pour l'observation, car elle apporte des propriétés à l'observation comme la *généricité*, qui permet d'utiliser les mêmes interfaces d'observation pour observer différents types d'entités et de données dans les différentes études de cas, ainsi que le *passage à l'échelle*, qui permet l'observation et l'analyse d'une sous-partie des éléments du MPSoC.

Par rapport à la portabilité, EMBera fournit des composants de traitement pour leur

utilisation sur différentes plates-formes. Cependant, la production des données, offert par les composants basiques, reste spécifique et doit être reimplémentée quand nous avons besoin d'observer une nouvelle plate-forme.

9.2.2 Simplicité d'intégration

Nous évaluons, ci-après, la simplicité d'intégration d'EMBera à la plate-forme observée, dans chacune des études de cas.

9.2.2.1 Décodeur MJPEG

Vu qu'aucune version embarquée du décodeur n'était disponible, nous avons implémenté une version de ce décodeur conçue pour l'observation avec EMBera et adaptée pour le MPSoC de l'étude de cas. Malgré le fait que nous n'avons pas modifié fonctionnellement le décodeur, nous considérons que dans ce cas l'observation n'est pas simple à intégrer. En effet, nous avons analysé le code du décodeur pour le programmer dans des composants EMBera afin de pouvoir l'observer, ce qui a introduit des modifications importantes dans le code. Par exemple, nous avons remplacé tous les transferts de données entre modules par des envois de messages entre composants. Suite aux modifications, le décodeur a été recompilé afin de fournir des observations. Cependant, les observations du RTOS, faites par les composants EMBera, sont simples à intégrer car le système offre des fonctions qui donnent accès à l'utilisation de la mémoire et les tâches.

Il est important à noter que dans ce cas, la simplicité d'intégration n'était pas l'un de nos objectifs.

9.2.2.2 Intergiciel multimédia STMicroelectronics

Nous avons observé la pile logicielle selon trois niveaux d'observation : l'application, l'intergiciel et le système d'exploitation. Nous considérons qu'en général, EMBera est simple à intégrer aux trois niveaux considérés. En effet, l'intégration a été faite sans modifier le code source du logiciel. En ce qui concerne l'application et le niveau intergiciel, nous avons utilisé les caractéristiques de `gcc` [gcc] et avons profité des options de profilage et de débogage, ajoutées à la compilation, afin de rajouter les traitements d'observation. En ce qui concerne le système d'exploitation, nous avons utilisé des interfaces déjà existantes et avons récupéré les données d'observation nécessaires, sans devoir instrumenter davantage.

9.2.2.3 Décodeur SVC

Vu que le décodeur SVC est organisé en composants, nous ajoutons des interfaces spécialisées d'EMBera aux composants, afin de permettre l'intégration de l'observation. Les interfaces ajoutées permettent la connexion des composants EMBera à l'application et au intergiciel.

Les interfaces d'EMBera sont ajoutées en modifiant le canevas à composants Cecilia, dans lequel le décodeur et l'intergiciel Comete sont programmés. Les modifications sont effectuées uniquement une fois, puis sont réutilisables pour toute application programmée avec Cécilia et qui utilise Comète. Les modifications effectuées permettent d'activer de manière simple l'observation d'un composant, en ajoutant un paramètre à la définition de celui-ci. Pour cette raison, nous considérons que le modèle EMBera est simple à intégrer. Nous précisons que si nous devons rajouter des nouvelles observations au canevas ou à l'intergiciel, au delà de celles qui sont définies, il est nécessaire d'effectuer des modifications dans le code et de le recompiler.

Regardant en détail les modifications, pour l'observation de Cecilia, nous ajoutons la définition des interfaces dans des fichiers IDL, ainsi que leurs implémentations, contenue dans des fichiers C indépendants à ceux de Cecilia. Une fois ajoutée cette définition, nous devons recompiler Cecilia. L'observation des composants applicatifs est possible en étendant la définition de ces composants dans le fichier ADL. Pour l'observation de Comete, les changement impliquent rajouter en dur les interfaces d'EMBera aux composants de l'intergiciel, puis le recompiler. Par rapport au système, il n'a pas subi des modifications pour son observation.

Discussion

Nous avons constaté que la simplicité d'intégration d'EMBera a été différente dans chacun des études de cas. Cette simplicité dépend de la technique utilisée pour collecter les données, ainsi que de l'information dont nous avons besoin.

En fonction du type de système à observer, nous définissons ce que nous observons (les entités), puis nous instancions EMBera pour ce système. L'instanciation consiste à implémenter une couche spécifique, plus ou moins intrusive, impliquant des modifications au logiciel observé. Puis l'observation peut s'effectuer à priori d'une manière simple.

9.2.3 Intrusivité

Nous présentons ci-après l'évaluation de l'intrusivité effectuée sur chacune des études de cas.

9.2.3.1 Décodeur MJPEG

Nous avons mesuré le surcoût, en temps d'exécution, des composants du décodeur. Les temps des composants sont montrés dans la table 9.1.

Nous constatons que l'intrusion due à l'observation 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 accélérateurs exécutent moins d'instructions pour les traitements liés à l'observation. Ce coût est lié à la performance des processeurs accélérateurs, en ce qui concerne les changements de contexte impliqués dans l'ordonnancement

Composant	Exéc. non observée (s)	Exéc. observée (s)	Surcoût (%)
`Fetch-Reorder`	83.30	87.14	4.6
`IDCT0`	6.42	7.36	14.7
`IDCT1`	7.19	8.47	17.8

TABLE 9.1 – Temps d'exécution supplémentaire du décodeur MJPEG lors de l'observation

des tâches des composants. Grâce à cette observation, nous avons décidé de déployer les composants de filtrage et de stockage sur le processeur maître, afin d'alléger les traitements d'observation sur les processeurs accélérateurs.

Le surcoût lié à la mémoire (cf. table 9.2) est négligeable pour les composants s'exécutant sur le processeur maître (au maximum il est de 2%) et important sur les accélérateurs (au moins 7%). Ceci nous mène aux mêmes conclusions que celles que nous avons eues pour le temps d'exécution : notamment qu'il est nécessaire d'alléger les traitements liés à l'observation sur les processeurs accélérateurs.

Composant	Taille au lancement (Ko)	Taille en fin d'exécution (Ko)	Taille structures d'observation (Ko)	Surcoût (%)
Fetch-Reorder (`CB0`)	2 460	14 870	24	$1 \rightarrow 0$
IDCT0 (`CB1`) ou IDCT1 (`CB2`)	132	332	24	$18 \rightarrow 7$
Filtre (`CF0`)	2 460	2 460	24	$1 \rightarrow 0$
Stockage (`CS0`)	2 460	28 312	2 048	$2 \rightarrow 0$

TABLE 9.2 – Mémoire utilisée par les composants et les structures d'observation dans l'application MJPEG

Enfin, nous mesurons la taille des historiques obtenus pour chaque processeur (composant). La mesure est d'environ 350 Ko pour le processeur maître et d'environ 160 Ko pour chaque processeur accélérateur. Vu que la plate-forme embarquée ne possède pas du support persistant de stockage, les données pour la génération de l'historique sont acheminées vers la machine de développement, à travers le port JTAG, pour leur enregistrement. Pour cette raison, nous considérons que le stockage n'induit pas d'intrusion, car il est effectué en dehors de la plate-forme embarquée.

9.2.3.2 Intergiciel multimédia STMicroelectronics

Nous avons comparé la durée d'exécution de l'application de décodage par rapport à une exécution non observée. Nous avons constaté, après plusieurs exécutions utilisant des vidéos de différentes tailles et différentes configurations pour l'observation (p. ex. différentes fréquences d'échantillonnage et de stockage), que les différences de temps n'ont pas dépassé 1%. Ce comportement peut être expliqué par le fait que l'application de décodage respecte des contraintes de temps réel. En d'autres termes, l'application doit respecter des délais, comme le montre la figure 9.1.

Vu que le temps d'exécution t_x est plus court que le temps entre deux échéances t_s, le système peut exécuter d'instructions supplémentaires sans induire une variation dans le temps d'exécution, si leur exécution n'est pas plus longue que $t_s - t_x$ (cf. figure 9.1(a)).

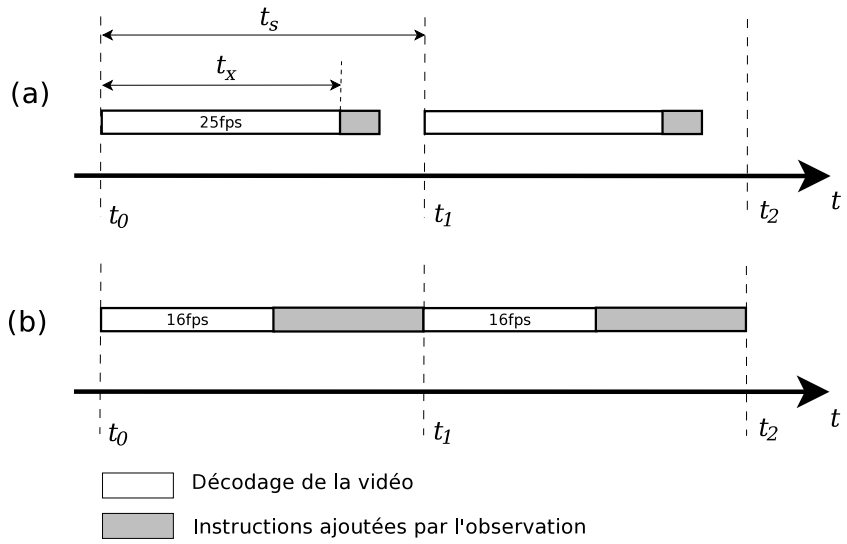

FIGURE 9.1 – Échéances du décodage

Autrement, l'application dégrade la qualité de décodage en réduisant soit le nombre d'images décodées, soit la qualité individuelles des images affichées (cf. figure 9.1 (b)) [1].

Afin d'obtenir une dégradation mesurable de l'application due à l'observation, nous avons augmenté la périodicité d'échantillonnage, initialement fixée à 1 $observation/s$, à 500 $observations/s$. Nous avons obtenu une dégradation de 4%, qui correspond au pourcentage du nombre d'images non décodés.

En ce qui concerne la mémoire, nous avons mesuré la taille utilisée par les trois instances du modèle EMBera, observant le système, l'intergiciel et l'application. La table 9.3 présente les valeurs maximales de mémoire observées dans chacun des cas, ainsi que le pourcentage de la mémoire ajoutée par rapport à une exécution non observée. La valeur maximale mesurée de la mémoire, dans une exécution non-observée du décodeur, a été d'environ 50 Mo. Nous utilisons cette valeur comme base pour la comparaison.

Niveau d'observation	**Mémoire utilisée par EMBera (Ko)**	**% de mémoire ajoutée**
Système	378	0,74
Intergiciel	1 449	2,83
Application	124	0,24

TABLE 9.3 – Utilisation de la mémoire par l'instance du modèle EMBera dans l'intergiciel multimédia

Nous observons que l'utilisation de la mémoire est directement liée au nombre de composants instanciés et à la quantité des données que le composant doit gérer. En effet, l'instance d'EMBera qui observe l'intergiciel utilise le plus de mémoire. Dans ce cas, les

1. *Frame skipping* et *quality reduction* sont deux techniques utilisées pour garantir des contraintes de temps de décodage dans MPEG-2 [IFS03].

composants basiques collectent des observations du décodage d'environ 24 $images/s$ [2], en ajoutant 2,9% de la mémoire utilisée par l'observation. Cet impact est considéré comme acceptable, parce que la plate-forme embarquée a suffisamment de mémoire vive disponible (plus de 60 Mo pendant le décodage d'un flux MPEG-2), par rapport à la quantité de mémoire nécessaire pour l'instance utilisée d'EMBera .

Enfin, la taille de l'historique enregistré a été d'environ 1 Mo. Cet historique, stocké comme un fichier de trace, à été enregistré sur une mémoire flash connecté à l'un des ports USB de la plate-forme.

9.2.3.3 Décodeur SVC

Vu que l'application ne respecte pas des contraintes temporelles, le temps nécessaire pour exécuter les fonctions d'observation est égal au temps d'exécution additionnel de toute l'application. Nous présentons ces temps d'exécution dans la figure 8.1. Si nous n'observons pas la communication, nous ajoutons $5,5\%$ du temps. Si nous observons la communication, ce temps s'incrémente à plus de 1 000%. Les valeurs obtenues peuvent être considérées comme acceptables pour la première observation, mais pas pour la deuxième.

Cependant, l'observation de la communication fournit des informations très riches, assez difficiles d'obtenir autrement. C'est pourquoi, cette intrusivité peut être considérée comme acceptable. De plus, l'intrusivité, même si elle réduit la quantité d'images décodées en 88%, n'affecte pas la qualité individuelle des images décodées. Pour cette raison, nous considérons que dans ces conditions d'exécution, l'observation ne perturbe pas le bon fonctionnement de l'application et les améliorations envisagées sont également valables pour une exécution non-observé de SVC.

Enfin, regardant le volume de données collectées, EMBera a générée un fichier de trace contenant plus de 200 millions d'entrées, dont la taille totale est de $8,5$ Go.

Discussion

L'intrusion varie principalement par rapport à la quantité et à la fréquence de données obtenues ainsi qu'au type d'application (temps réel ou non). Nous constatons de manière pratique, que l'intrusivité n'est pas simplement une valeur numérique mais qu'elle représente un compromis entre la précision des observations et la perturbation admissible par la plate-forme observée.

Dans l'observation de MJPEG, nous constatons que si nous appliquons les mêmes traitements d'EMBera sur deux architectures de processeur différentes, ils ajoutent des intrusions différentes. Nous observons que pour avoir une intrusion plus faible, les traitements d'EMBera doivent être exécutés, autant que possible, sur le processeur maître. Pour la mémoire, la quantité des données et la taille des traces, permettent leur envoi sur le port de débogage sans introduire une perturbation importante. Cependant, un plus gros volume doit induire plus de perturbations dues à l'acheminement de ces données.

2. Fréquence d'affichage de la vidéo utilisée par l'application multimédia du chapitre 7.

L'observation de l'intergiciel multimédia permet de constater que l'intrusion n'est pas toujours mesurable en tant qu'un pourcentage du temps total d'exécution. En effet, nous observons qu'EMBera n'affecte pas ce temps total. Cependant, si la collecte de données aurait été plus fréquente, nous aurons aperçu une dégradation de la qualité de la vidéo décodée. En ce qui concerne la mémoire, les données produites rentrent encore dans la mémoire disponible. Toutefois, lors d'une observation plus détaillée, il est nécessaire d'envisager l'utilisation des dispositifs de support matériel pour la gestion des traces d'exécution.

Enfin, nous constatons dans l'observation du décodeur SVC, que l'intrusion augmente en fonction de la quantité et de la fréquence des observations. Lors d'une première observation, le système a été faiblement perturbé, mais les informations obtenues n'ont pas été suffisantes pour supporter le choix du déploiement. La deuxième observation, plus intrusive, car elle observe la communication, fournit des informations qui permettent de définir un déploiement adéquat. En ce qui concerne à la mémoire, nous n'aurons pas pu gérer les tampons des composants EMBera, ni la trace dans un système embarqué classique. Ce qui veut dire que pour ce type d'observation, le système doit être muni d'un port de traçage ou d'un espace considérable de mémoire consacrée au support de l'observation.

Synthèse

Dans ce chapitre, nous avons évalué la pertinence du modèle EMBera pour l'observation. Nous avons utilisé trois critères d'évaluation, notamment la valeur ajoutée d'EMBera pour l'observation, la simplicité d'intégration et l'intrusion. Nous avons considéré les trois études de cas portant sur trois applications multimédia : un décodeur simple MJPEG, une application industrielle de STMicroelectronics et un décodeur du format SVC, implémenté à base de composants.

Dans l'observation des décodeurs MJPEG et SVC, l'observation nous a permis de réfléchir à des améliorations et à déceler des problèmes sur des applications, qui n'ont pas été testées auparavant sur les plates-formes utilisées. Cela permet de prouver qu'EMBera sert comme une solution pour trouver des problèmes pendant l'évolution et le portage du logiciel entre plates-formes matérielles. De même, l'utilisation d'EMBera a dévoilée des problèmes d'utilisation de ressources sur une plate-forme exécutant du logiciel déjà optimisé.

D'autre part, nous avons constaté que l'intégration d'EMBera dépend directement du type d'éléments qui ont besoin d'être observés. Dans nos expérimentations la collecte de données d'EMBera, sauf pour le décodeur SVC, a reposée sur des mécanismes d'interception ou des interfaces du système relativement simples à utiliser. Toutefois, si nous avons besoin de récupérer des informations, qui ne sont pas disponibles par le biais de ces mécanismes, nous devons faire des modifications et des implémentations importantes, au logiciel observé et à EMBera.

Enfin, nous avons constaté que l'intrusion ajoutée par les implémentations d'EM-

Bera doit être évaluée en fonction de la perturbation de l'exécution et de la richesse des informations fournies. Par exemple, nous avons vu qu'une observation détaillée de la communication introduit un surcoût considéré comme inacceptable, considérant sa valeur, mais qui permet de reconstruire le comportement de toute l'application.

Chapitre 10

Conclusion et perspectives

Pendant cette thèse, nous nous sommes intéressés à la définition d'une nouvelle approche d'observation pour les systèmes embarqués. Notre proposition consiste en un modèle d'observation, appelé **EMBera**, qui permet de structurer l'observation avec des composants logiciels. Nous avons appliqué cette approche à plusieurs études de cas.

Synthèse sur les choix de conception

Dans l'introduction de cette thèse, nous avons énoncé quelques uns des défis auxquels nous souhaitions répondre avec la nouvelle génération des outils pour la mise au point des MPSoC. Ci-après, nous discutons comment nous avons répondu à ces défis avec notre modèle d'observation EMBera :

- *Généricité :* La propriété de généricité de notre environnement d'observation repose principalement sur le choix d'un modèle à composants. Les composants vont d'abord encapsuler l'observation des éléments à analyser. Les interfaces des composants vont constituer une interface unique pour l'observation. Ces interfaces vont également permettre de contrôler l'observation. Toutefois, le modèle proposé ne définit actuellement qu'un nombre limité de fonctionnalités d'observation et de types de données. Nous devons étendre ces définitions afin de fournir l'observation d'éléments complexes qui n'ont pas été considérés dans les études de cas.
- *Portabilité :* Dans l'environnement d'observation construit avec le modèle EMBera, des composants spécifiques (de bas-niveau) collectent les données directement du dispositif matériel. Ces composants sont fortement dépendants de l'architecture et donc non portables. EMBera fournit un ensemble de codes à trous où l'implémentation spécifique peut-être complétée. L'environnement d'observation basé sur EMBera comprend un ensemble de composants qui vont traiter les données observées et ensuite les stocker sur un support de mémoire persistante. Cette dernière partie est indépendante de la plate-forme et donc portable.
- *Passage à l'échelle :* L'observation d'un grand nombre d'éléments est abordée par

EMBera en permettant une observation partielle du MPSoC. L'utilisateur ne définira des composants que sur la sous-partie du système sur laquelle il souhaite concentrer son analyse. Dans les trois études de cas, nous avons construit un dispositif d'observation avec un sous-ensemble restreint d'éléments (threads, fonctions, composants).

- *Intrusivité :* Nous avons traité le contrôle de l'intrusion au travers de deux aspects ;
  1. Nous avons laissé la possibilité à l'utilisateur de fixer des paramètres des composants comme la fréquence d'observation. Cela permet de fournir un outil pour ajuster le compromis entre la finesse des observations et la perturbation. Dans l'état actuel de la plate-forme, les mécanismes de réglage sont assez basiques et restent à être étendus ;
  2. Nous avons porté une attention toute particulière à éviter de modifier le code source des applications et la logique applicative. Pour cela, nous avons utilisé certaines informations externes à l'application comme les données de profilage fournies par le compilateur ou par le système d'exploitation. Cependant, même si nous avons ajouté un faible nombre de lignes de code pour l'instrumentation, nous n'avons pas cherché à optimiser le code d'EMBera. D'ailleurs, nous n'avons pas exploré d'autres possibilités offertes pour l'instrumentation, comme l'instrumentation du code binaire.

Nous considérons que notre approche a abordé partiellement les défis définis dans la thèse. En effet, nous sommes conscients que la solution fournie par EMBera a besoin d'être étendue, par exemple, en supportant plus de types d'éléments et des données à observer, ou en définissant des fonctions de haut niveau, qui abstraient plus facilement la plate-forme observée. Malgré cela, nous pensons qu'EMBera est un bon point de départ pour la définition d'une solution complète d'observation pour les MPSoC, car il fournit une couche générique qui permet la définition des nouveaux traitements et des observations.

Synthèse sur l'expérimentation

Nous avons montré, dans les études de cas, que le modèle EMBera a fonctionné comme une solution pour l'observation sur trois plates-formes avec des configurations différentes. Nous considérons que l'utilisation d'EMBera a été relativement simple, et qu'il a permis d'obtenir les informations souhaitées. Ci-dessus, une brève synthèse des trois études de cas.

- *Décodeur MJPEG* : nous avons montré que nous pouvons utiliser le modèle à composants EMBera pour encapsuler des fonctionnalités d'une application en les structurant à l'aide des composants. Nous constatons que les observations obtenues ont permis de détecter des problèmes liés à la gestion mémoire et au déploiement du décodeur, ainsi que d'optimiser le décodeur en réduisant son temps d'exécution.
- *Application et intergiciel multimédia* : nous avons étudié un logiciel optimisé pour une plate-forme embarquée, utilisée dans des produits à STMicroelectronics. Les

observations ont été analysées pour détecter des problèmes dans la gestion mémoire en présence des flux vidéo avec des données corrompues. De plus, l'observation a permis de comprendre le déroulement des étapes du décodage vidéo, dans un logiciel qui est composé de plus de 2 millions lignes de code.

– *Décodeur SVC* : nous avons étudié un décodeur du format SVC, programmé dans des composants logiciels du canevas Cecilia et qui utilise l'intergiciel Comete pour son déploiement. Le décodeur a été exécuté sur une plate-forme SMP à 24 cœurs. Cette plate-forme est munie de processeurs généralistes, mais nous pensons que son architecture représente les systèmes embarqués de demain. Les observations obtenues ont facilité la compréhension du fonctionnement du décodeur, principalement en ce qui concerne la communication entre les différents traitements. Ces observations ont permis de sélectionner le déploiement le plus approprié des composants sur les cœurs de la plate-forme SMP. Les observations ont été supportées par une implémentation d'EMBera pour l'observation de Cecilia et Comete. Nous pensons que cette implémentation peut être proposée pour être intégrée aux distributions du canevas et de l'intergiciel. En effet, l'implantation va automatiquement ajouter l'observation à toute application développée avec Cecilia et déployée avec Comete.

Les observations ont été effectuées à l'aide des prototypes de recherche d'EMBera, qui n'ont pas encore la maturité pour être utilisés industriellement. En effet, la mise en œuvre a des limitations en termes du nombre d'éléments observables et de robustesse. D'autre part, nous n'avons pas exploré d'autres mécanismes de collecte d'information comme l'instrumentation dynamique, effectuée par Pin [LCM+05], ou l'utilisation de langages pour effectuer des requêtes pour obtenir des observations, faite par `DTrace` [CSL04].

Perspectives

Suite à notre travail de thèse, nous visons quelques extensions possibles en termes de modèle et d'implémentation. Nous pensons que pour le modèle, il est possible d'envisager les améliorations suivantes :

– *Composition* : Nous pensons que le modèle d'observation doit tirer profit du concept de composition de composants. Grâce à l'exploitation de la composition, nous pourrions créer des groupes de composants, représentant des ensembles d'éléments à observer. Sur ces regroupements, nous pourrions appliquer des configurations ou des traitements. De plus, la composition permettrait d'établir des niveaux de détail pour l'observation. Cela veut dire qu'un composant composite, représentant tout le système observé, pourrait fournir par défaut des observations très succinctes du système. Si un observateur aurait besoin de plus de détail, le composant permettrait, à travers ses interfaces, d'explorer les observations qui ont générée les observations succinctes, données initialement.

– *Intrusivité* : Nous pensons que le modèle doit permettre d'exprimer l'intrusivité, en termes d'utilisation des ressources ciblant la dégradation de la qualité du calcul effectué. Pour cela, nous pensons que le modèle ne doit pas être absolu, c'est-à-

dire, qu'il doit pouvoir se spécialiser en fonction du type d'application embarquée (multimédia, réseau, sécurité, etc.).

- *Observation causale multi-niveau* : Nous considérons que le modèle doit tenir compte de la corrélation entre les observations obtenues aux différents niveaux observés. Par exemple, dans l'observation d'un décodeur, qu'il soit possible de déterminer que la cause de la réduction de la qualité de service est due au grand nombre des défauts de cache.

Concernant l'implémentation, nous considérons que les extensions sont dirigées par les besoins du modèle d'observation :

- Pour la généricité, l'implémentation doit supporter plus des types d'éléments et de données en considérant, par exemple, des nouveaux modèles de programmation pour l'embarqué ;
- Pour la portabilité d'EMBera, la mise en œuvre du modèle doit être spécialisée par domaines d'application (communications, systèmes de navigation, etc). Cette spécialisation peut comporter des ensembles des composants de traitement préconfigurés par domaine d'application, permettant, par exemple, d'obtenir la qualité d'une transmission dans un système de communication, ou le temps total de freinage dans une voiture ;
- Pour le passage à l'échelle, EMBera doit supporter la gestion d'un grand nombre de composants d'observation, principalement concernant le déploiement et l'acheminement des observations ;
- Enfin, pour l'intrusivité, l'implémentation doit rajouter des mécanismes pour un contrôle plus précis de la collecte de données et du stockage. De même, le code des composants EMBera doit être optimisé, surtout celui qui s'exécute sur la plateforme sous observation.

Table des figures

2.1 Étapes suivies pour l'observation d'un système informatique 18

2.2 Visualisation d'une trace d'exécution d'un système d'exploitation. Les lignes horizontales représentent les historiques d'exécution de différentes entités (cœurs, processus et threads) 22

2.3 Plate-forme multi-cœur ARM avec des composants de traçage ETM/PTM 24

2.4 Collecte de traces du processeur CELL 26

2.5 Gestion des traces dans TAU 30

2.6 Collecte et traitement de données dans HPCToolkit 31

2.7 Vues de performance basées sur de profilage 33

2.8 Hiérarchie de surveillance dans Ganglia [Kra09] 36

2.9 Génération de l'historique des interactions dans BlackBoxes [AMW+03] . 37

3.1 Interfaces, connexions et composition des composants 42

3.2 Schéma de communication dans Comete 49

3.3 Niveaux d'abstraction gérés par Koala 51

3.4 Observation d'une application CCA avec PMM 54

3.5 Hiérarchies dans Composite Probes 55

3.6 Le canevas de Rainbow 56

3.7 Administration des éléments d'un Serveur Web 57

5.1 Le modèle à composants EMBera 68

5.2 Interfaces d'observation d'un composant EMBera 70

5.3 Exemple de composant basique : observation de la mémoire utilisée par un processus 72

5.4 Exemple de composant de filtrage 73

5.5 Exemple de composant d'agrégation 74
5.6 Exemple de composant de stockage 75
5.7 Exemple de déploiement des composants EMBera 77
5.8 Implémentation des composants d'observation EMBera 78
5.9 Extrait de code d'un composant basique 78
5.10 Extrait de code d'un composant de filtrage 79
5.11 Extrait de code d'un composant de stockage 80
5.12 Extrait de code d'une fonction de profilage 81
5.13 Description d'une instance d'EMBera en format XML 82

6.1 Plate-forme STi7200 88
6.2 Application de décodage MJPEG avec des composants EMBera 89
6.3 Déploiement de l'application de MJPEG sur la plate-forme *STi7200* . . . 91
6.4 Extrait du code exécuté par le composant IDCT0 92
6.5 Extrait du code `main` exécuté sur le processeur ST231 92
6.6 Composants EMBera utilisés pour l'observation de l'application MJPEG . 95
6.7 Code de déploiement des composants d'observation sur le processeur *ST40* 96
6.8 Déploiement des composants d'observation EMBera 97
6.9 Évolution de la mémoire des composants 98
6.10 Temps de communication en utilisant la primitive `send` 100

7.1 Plate-forme *IDTV* 104
7.2 Exemple de déploiement des composants EMBera 106
7.3 Composants EMBera utilisés pour observer l'application de décodage . . . 108
7.4 Composants EMBera utilisés pour observer l'intergiciel multimédia 109
7.5 Composants EMBera utilisés pour observer le processus de l'application . 109
7.6 Extrait de code d'une fonction de profilage 110
7.7 Flots d'exécution créés pour l'observation de l'application 112
7.8 Différences des temps d'exécution des fonctions de décodage 113
7.9 Évolution de l'utilisation de la mémoire et des threads utilisées par l'application sur trois exécutions 114
7.10 Corrélation des évènements applicatifs avec l'intergiciel et le système . . . 115

8.1 Organisation des cœurs et de la mémoire cache dans le processeur Intel x7460 118
8.2 Chaîne d'outils Cecilia 119
8.3 Communication et exécution dans Cecilia 120
8.4 Architecture de l'application de décodage SVC implémentée avec des composants Cecilia. 121
8.5 Modèle d'exécution de Comete 123
8.6 Modifications effectuées à Cecilia pour l'ajout d'EMBera 124
8.7 Composant basique observant la communication dans Comete 125
8.8 Composants EMBera utilisés pour l'observation du décodeur SVC 128
8.9 Méthodes ajoutées pour l'observation de Comete 129
8.10 Définition des composants EMBera dans Comete 130
8.11 Fichier XML de configuration des composants d'observation 132
8.12 Déploiement des composants SVC et EMBera 132
8.13 Exécution des composants de SVC déployés sur les PEs définis dans Comete 135
8.14 Évolution de l'utilisation mémoire de l'application SVC 137
8.15 Décodage d'une image dans SVC 140

9.1 Échéances du décodage 150

Liste des tableaux

3.1 Travaux orientés composants considérés 44

5.1 Exemples des entités et des évènements par niveau d'observation 69

6.1 Temps d'exécution des composants du décodeur MJPEG 97

7.1 Objectifs d'observation organisés par niveau logiciel 107

7.2 Nombre d'échecs de décodage des trames 112

8.1 Temps d'exécution du décodeur SVC 134

8.2 Pourcentages d'utilisation des PEs par les composants SVC 136

9.1 Temps d'exécution supplémentaire du décodeur MJPEG lors de l'observation 149

9.2 Mémoire utilisée par les composants et les structures d'observation dans l'application MJPEG 149

9.3 Utilisation de la mémoire par l'instance du modèle EMBera dans l'intergiciel multimédia 150

Bibliographie

[93801] IEEE Standard Test Access Port and Boundary-Scan Architecture. *IEEE Std 1149.1-2001*, pages i –200, 2001.

[AAW+02] B.A. Allan, R.C. Armstrong, A.P. Wolfe, J. Ray, D.E. Bernholdt, and J.A. Kohl. The CCA core specification in a distributed memory SPMD framework. *Concurrency and Computation : Practice and Experience*, 14(5) :323–345, 2002.

[ABF+10] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and NR Tallent. HPCToolkit : Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation : Practice and Experience*, 22(6) :685–701, 2010.

[AMW+03] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Blackboxes. In *SOSP '03 : Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 74–89, New York, NY, USA, 2003. ACM.

[APcDG05] 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.

[ARM10] ARM Limited. ARM CoreSight. Website, 2010. `http://www.arm.com/products/system-ip/coresight.php/`.

[AS08] AWOX and STMicroelectronics. Connected TVModule. Website, 2008. `http://www.awox.com/pdf/us/ConnectedTVM.pdf`.

[BCL+06] 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".

[BCM04] Frédéric Briclet, Christophe Contreras, and Philippe Merle. OpenCCM : une infrastructure à composants pour le déploiement d'applications à base de composants CORBA. In IMAG/LSR, editor, *DECOR04*, ISBN : 2-7261-1276-5, pages 101–112, 2004.

[BDG+00] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. High Perform. Comput. Appl.*, 14(3) :189–204, 2000.

[BDH+98] Manfred Broy, Anton Deimel, Juergen Henn, Kai Koskimies, Frantisek Plasil, Gustav Pomberger, Wolfgang Pree, Michael Stal, and Clemens A. Szyperski. What characterizes a (software) component ? *Software - Concepts and Tools*, 19(1) :49–56, 1998.

[BLM+07] Koen Bosschere, Wayne Luk, Xavier Martorell, Nacho Navarro, Mike O'Boyle, Dionisios Pnevmatikatos, Alex Ramirez, Pascal Sainrat, André Seznec, Per Stenström, and Olivier Temam. High-Performance Embedded Architecture and Compilation Roadmap. *Transactions on High-Performance Embedded Architectures and Compilers I*, pages 5–29, 2007.

[Box97] Don Box. *Essential COM.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1997.

[BST+08] M. Biberstein, U. Shvadron, J. Turek, B. Mendelson, and M.S. Chang. Trace-based Performance Analysis on Cell BE. In *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, pages 213 –222, 20-22 2008.

[CBG+08] Geoff Coulson, Gordon Blair, Paul Grace, Francois Taiani, Ackbar Joolia, Kevin Lee, Jo Ueyama, and Thirunavukkarasu Sivaharan. A Generic Component Model for Building Systems Software. *ACM Trans. Comput. Syst.*, 26 :1 :1–1 :42, March 2008.

[CCM05] Paolo Costa, Geoff Coulson, and Cecilia Mascolo. The RUNES Middleware : A Reconfigurable Component-based Approach to Networked Embedded Systems. In *Proceedings of 16th International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC05)*, pages 11–14. IEEE Press, 2005.

[CFB05] A.C. Chow, G.C. Fossum, and D.A. Brokenshire. A Programming Example : Large FFT on the Cell Broadband Engine. *Global Signal Processing Expo (GSPx)*, 2005.

[CMdA+08] Gustavo Rau de Almeida Callou, Paulo Romero Martins Maciel, Ermeson Carneiro de Andrade, Bruno Costa e Silva Nogueira, and Eduardo Tavares. A Coloured Petri Net Based Approach for Estimating Execution Time and Energy Consumption in Embedded Systems. In *SBCCI '08 : Proceedings of the 21st annual symposium on Integrated circuits and system design*, pages 134–139, New York, NY, USA, 2008. ACM.

[Com10] The Symbian Foundation Community. Symbian operating system, 2010. `http://www.symbian.org/`.

[Con] ObjectWeb Consortium. JOnAS Java Open Application Server. Webpage. `http://wiki.jonas.ow2.org/`.

[Cor07a] Jonathan Corbet. Linux Kernel Markers. Website, August 2007. `http://lwn.net/Articles/245671/`.

[Cor07b] IBM Corp. CELL BE SDK 3.0. Website, 2007. `http://www.ibm.com/developerworks/power/cell/pkgdownloads.html`.

[Cor10] Microsoft Corporation. Windows Embedded Homepage. Website, 2010. `http://www.microsoft.com/windows/embedded/default.mspx`.

[CRDI05] Thomas Chen, Ram Raghavan, Jason Dale, and Eiji Iwata. Cell Broadband Engine Architecture and its First Implementation. Website, November 2005. `http://www.ibm.com/developerworks/power/library/pa-cellperf/`.

[Crn04] Ivica Crnkovic. Component-Based Approach for Embedded Systems. In *Proceedings of Ninth International Workshop on Component-Oriented Programming*, 2004.

[CSL04] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems. In *ATEC '04 : Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[DD06] M. Desnoyers and M.R. Dagenais. The LTTng tracer : A Low Impact Performance and Behavior Monitor for GNU/Linux. In *Proceedings : Ottawa Linux Symposium*, pages 209–224, 2006.

[DD09] M. Desnoyers and M.R. Dagenais. Deploying LTTng on Exotic Embedded Architectures. In *Embedded Linux Conference 2009*, 2009.

[Def81] Defense Advanced Research Projects Agency. Rfc 791 - internet protocol, September 1981. http ://tools.ietf.org/html/rfc791.

[Des09] Mathieu Desnoyers. *Low-Impact Operating System Tracing*. PhD in Computer Science, Université de Montreal, 2009.

[DGMB07] Bruno Diniz, Dorgival Guedes, Wagner Meira, Jr., and Ricardo Bianchini. Limiting the Power Consumption of Main Memory. *SIGARCH Comput. Archit. News*, 35 :290–301, June 2007.

[Dia07] Ada Diaconescu. Composite Probes, a Monitoring Framework for Organizing Data into Configurable Hierarchies. Technical report, Orange Labs, February 2007.

[Dil09] Bruno Dillenseger. CLIF, a framework based on Fractal for flexible, distributed load testing. *Annals of Telecommunications*, 64 :101–120, 2009. 10.1007/s12243-008-0067-9.

[dKdOS00] Jacques Chassin de Kergommeaux and Benhur de Oliveira Stein. Pajé : An Extensible Environment for Visualizing Multi-threaded Programs Executions. In *Euro-Par '00 : Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, London, UK, 2000. Springer-Verlag.

[DLLKK06] V. De La Luz, M. Kandemir, and I. Kolcu. Reducing Memory Energy Consumption of Embedded Applications that Process Dynamically Allocated Data. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(9) :1855 –1860, 2006.

[DM98] L. Dagum and R. Menon. Open MP : An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1) :46–55, 1998.

[dPBH+08] Noël de Palma, Sara Bouchenak, Daniel Hagimont, Sylvain Sicard, and Christophe Taton. Jade : Un Environnement d'Administration Autonome. *Techniques et Sciences Informatiques*, 2008.

[ea09] P. Lichtner et al. PFLOTRAN Project Website. Website, 2009. `https://software.lanl.gov/pflotran`.

[EH06] F.C. Eigler and R. Hat. Problem solving with SystemTAP. In *Proceedings of the Ottawa Linux Symposium*, 2006.

[Eva05] UPC Group Performance Tool Evaluation. Practical Experiences with Modern Parallel Performance Analysis Tools : An Evaluation. Whitepaper, July 2005. `http://www.hcs.ufl.edu/upc/archive/toolevals/-WhitepaperEvalSummary.pdf`.

[Fag97] Alain Fagot. *Réexécution déterministe pour un modèle procédural parallèle basé sur les processus légers*. PhD thesis, Institut National Polytechnique de Grenoble - INPG, 1997.

[Fou10] The Apache Software Foundation. Apache Maven Project. Website, 2010. `http://maven.apache.org/`.

[Gar97] V.K. Garg. Methods for Observing Global Properties in Distributed Systems. *Concurrency, IEEE*, 5(4) :69 –77, oct-dec 1997.

[gcc] The GNU Compiler Collection. `http://gcc.gnu.org/`.

[GCH+04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow : Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37 :46–54, 2004.

[Ger08] Germain Haugou. Comete User Manual, 2008. `http://fractal.ow2.org/minus-site/current/comete/Comete_tutorial.pdf`.

[GKM04] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof : A Call Graph Execution Profiler. *SIGPLAN Not.*, 39(4) :49–57, 2004.

[Gro10] Object Management Group. Catalog of OMG Modeling and Metadata Specifications. Website, May 2010. `http://www.omg.org/technology/documents/modeling_spec_catalog.htm`.

[GS99] A. Gokhale and D.C. Schmidt. Techniques for Optimizing CORBA Middleware for Distributed Embedded Systems. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 513 –521 vol.2, mar 1999.

[GWW+10] M. Geimer, F. Wolf, B.J.N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca Performance Toolset Architecture. *Concurrency and Computation : Practice and Experience*, 22(9999) :702–719, 2010.

[GWWM09] Markus Geimer, Felix Wolf, Brian J.N. Wylie, and Bernd Mohr. A Scalable Tool Architecture for Diagnosing Wait States in Massively Parallel Applications. *Parallel Computing*, 35(7) :375 – 388, 2009.

[Hru01] Cathy Hrustich. CORBA for Real-Time, High Performance and Embedded Systems. In *Object-Oriented Real-Time Distributed Computing, 2001. ISORC - 2001. Proceedings. Fourth IEEE International Symposium on*, pages 345 –349, 2001.

[Huf52] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9) :1098 –1101, sep. 1952.

[IBM09] IBM Corp. Visualize Function Calls with Graphviz, 2009. http ://www.ibm.com/developerworks/library/l-graphvis/.

[iCM10] iCMG : Software for CORBA Components. Webpage, 2010. `http://www.icmgworld.com`.

[IFS03] Damir Isovic, Gerhard Fohler, and Liesbeth Steffens. Timing Constraints of MPEG-2 Decoding for High Quality Video : Misconceptions and Realistic Assumptions. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, 2003.

[Inc] Sun Microsystems Inc. Java Remote Method Invocation. Website. `http://www.oracle.com/technetwork/java/javase/tech/-rmi-141156.html`.

[Inc02] Sun Microsystems Inc. Java Message Service Specification Final Release 1.1. Website, April 2002. `http://www.oracle.com/technetwork/java/overview-137943.html`.

[Inc10] Free Software Foundation Inc. GDB : The GNU Project Debugger. Website, 2010. http ://www.gnu.org/software/gdb/.

[INR10a] INRIA. Generic Trace Generator (GTG). Website, October 2010. `https://gforge.inria.fr/projects/gtg/`.

[INR10b] INRIA. Visual Trace Explorer. Website, December 2010. `http://vite.gforge.inria.fr/`.

[Int08] Intel Corporation. Intel xeon processor x7460. Website, 2008. `http://ark.intel.com/Product.aspx?id=36947`.

[iso07] ISO/IEC 15444-3, "Information Technology–JPEG2000 Image Coding System–Part 3 : Motion JPEG2000", 2007.

[iso08] ISO/IEC 29341-1, "Information technology – UPnP Device Architecture – Part 1 : UPnP Device Architecture Version 1.0", 2008.

[Jai91a] Raj Jain. *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modeling*, chapter 8, pages 112–114. John Wiley & Sons, 1st edition, April 1991.

[Jai91b] Raj Jain. *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modeling*, chapter 7, pages 93–110. John Wiley & Sons, 1st edition, April 1991.

[jbo10] JBoss Community. Website, 2010. `http://www.jboss.org`.

[KBB+06] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. Nagel. Introducing the Open Trace Format (OTF). In *In International Conference on Computational Science*, pages 526–533. Springer, 2006.

[Kra08] Paul Kranenburg. `strace` : System call tracer. Website, August 2008. `http://sourceforge.net/projects/strace/`.

[Kra09] Sacha Krakowiak. *Middleware Architecture with Patterns and Frameworks*, chapter 10, pages 10–1–10–51. Creative Commons, October 2009. `http://sardes.inrialpes.fr/ krakowia/MW-Book/Chapters/-Admin/admin-body.html`.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin : Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI '05 : Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[Lev04] John Levon. *OProfile Manual*. Victoria University of Manchester, 2004. `http://oprofile.sourceforge.net`.

[Lil00] David Lilja. *Measuring Computer Performance : A practitioner's guide*, chapter 6, pages 82–110. Cambridge University Press, New York, NY, USA, August 2000.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20 :46–61, January 1973.

[LV95] D.C. Luckham and J. Vera. An Event-based Architecture Definition Language. *Software Engineering, IEEE Transactions on*, 21(9) :717 –734, sep 1995.

[Mai96] Eric Maillet. *Le traçage logiciel d'applications parallèles : conception et ajustement de qualité*. PhD thesis, Institut National Politechnique de Grenoble, 1996.

[Mar03] Vania Marangozova. *Duplication et cohérence configurables dans les applications réparties à base de composants*. PhD thesis, Université Joseph Fourier — Grenoble 1, June 2003.

[Mar06] Grant Martin. Overview of the MPSoC Design Challenge. In *Proceedings of the 43rd annual Design Automation Conference*, DAC '06, pages 274–279, New York, NY, USA, 2006. ACM.

[MCC04] M.L. Massie, B.N. Chun, and D.E. Culler. The Ganglia Distributed Monitoring System : Design, Implementation, and Experience. *Parallel Computing*, 30(7) :817–840, 2004.

[MMSW02] 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.

[MSMS08] Alan Morris, Wyatt Spear, Allen D. Malony, and Sameer Shende. Observing Performance Dynamics Using Parallel Profile Snapshots. In *Euro-Par '08 : Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pages 162–171, Berlin, Heidelberg, 2008. Springer-Verlag.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind : A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.*, 42(6) :89–100, 2007.

[Obj06] Object Management Group. CORBA Component Model, v4.0. Website, 2006. `http://www.omg.org/technology/documents/formal/components.htm`.

[Obj09a] ObjectWeb Consortium. Cecilia Framework. Website, 2009. `http://fractal.ow2.org/cecilia-site/current/`.

[Obj09b] ObjectWeb Consortium. ThinkMC Programming Language. Website, 2009. `http://fractal.ow2.org/cecilia-site/2.1.0/toolchain-parent/-cecilia-adl/thinkmc.html`.

[POS93] *Portable Operating System Interface (POSIX)—Part 2 : Shell and Utilities (Volume 1)*. Information technology—Portable Operating System Interface (POSIX). 1993.

[Ray96] Eric S. Raymond. *The New Hacker's Dictionary*, page 149. MIT Press, Cambridge, MA, USA, 3rd edition, October 1996.

[RCD+00] R. Rosner, A. Calder, J. Dursi, B. Fryxell, D.Q. Lamb, J.C. Niemeyer, K. Olson, P. Ricker, F.X. Timmes, J.W. Truran, H. Tueo, Yuan-Nan Young, M. Zingale, E. Lusk, and R. Stevens. Flash Code : Studying Astrophysical Thermonuclear Flashes. *Computing in Science Engineering*, 2(2) :33 –41, march-april 2000.

[ROC08] Erven Rohou, Andrea Ornstein, and Marco Cornero. Compiling C to CLI for Heterogeneous Multicore SoCs. 5th HiPEAC Industrial Workshop, June 2008.

[RTA+04] J. Ray, N. Trebon, R.C. Armstrong, S. Shende, and A. Malony. Performance Measurement And Modeling of Component Applications in a High Performance Computing Environment : A Case Study. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 95, 2004.

[Sch06] Douglas Schmidt. Real-time CORBA with TAO. Website, October 2006. `http://www.cs.wustl.edu/~schmidt/TAO.html`.

[SLJD08] E. Senn, J. Laurent, E. Juin, and J.-P. Diguet. Refining Power Consumption Estimations in the Component-based AADL Design Flow. In *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, pages 173 –178, 23-25 2008.

[SM06] Sameer Shende and Allen Malony. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2) :287–311, 2006.

[SOW+95] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI : The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.

[STM06] STMicroelectronics. STi7200 Data Brief, September 2006. http://www.st.com/stonline/products/literature/bd/12876.pdf.

[STM07a] STMicroelectronics. Component Based System Technology for xSTream, January 2007. http://fractal.objectweb.org/minus-site/current/-comete/CBSTxST-0_7.pdf.

[STM07b] STMicroelectronics. *OS21 User Manual*. STMicroelectronics, 2007.

[STM07c] STMicroelectronics. STMicroelectronics to Demonstrate Wide Array of Cutting-Edge Solutions at CEATEC Japan 2007, Septembre 2007. http://www.st.com/stonline/stappl/cms/press/news/year-2007/t2216.htm.

[STM08] STMicroelectronics. Minus Overview. Website, December 2008. http://fractal.ow2.org/minus-site/current/root/index.html.

[STM09a] STMicroelectronics. Dynamic Kernel Tracing with KPTrace. Website, 2009. http://www.stlinux.com/devel/traceprofile/kptrace.

[STM09b] STMicroelectronics. STLinux Distribution. Website, 2009. http://www.stlinux.com/.

[STM10] STMicroelectronics. STWorkbench 4.01. Website, 2010. http://www.stlinux.com/node/737.

[Sun98] Sun Microsystems Inc. Enterprise JavaBeans Technology. Website, 1998. http://java.sun.com/products/ejb/.

[Sun10] Sun Microsystems Inc. Java Pet Store Demo. Website, 2010. http://developer.java.sun.com/developer/releases/petstore/.

[SW08] H. Schwarz and M. Wien. The Scalable Video Coding Extension of the H.264/AVC Standard [Standards in a Nutshell]. *Signal Processing Magazine, IEEE*, 25(2) :135 –141, mar. 2008.

[sys] Systemtap. Website. http://sourceware.org/systemtap/.

[TAMC10] N.R. Tallent, L. Adhianto, and J.M. Mellor-Crummey. Scalable Identification of Load Imbalance in Parallel Executions Using Call Path Profiles. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1 –11, 2010.

[TG06] MIPI Test and Debug Working Group. MIPI Test and Debug Interface Framework v3.2. Whitepaper, April 2006. http://www.mipi.org/docs/mipi_tdwg_whitepaper_v3_2.pdf.

[TGG+05] Tobin Titus, Syed Gilani, Mike Gillespie, James Hart, Benny Mathew, Andy Olsen, David Curran, Jon Pinnock, Robin Pars, Fabio Ferracchiati, Sandra Gopikrishna, Tejaswi Redkar, and Srinivasa Sivakumar. The .net component model. In *Pro .NET 1.1 Remoting, Reflection, and Threading*, pages 343–395. Apress, 2005. 10.1007/978-1-4302-0025-3_12.

[TVK08] Jelena Trajkovic, Alexander V. Veidenbaum, and Arun Kejariwal. Improving SDRAM Access Energy Efficiency for Low-Power Embedded Systems. *ACM Trans. Embed. Comput. Syst.*, 7(3) :1–21, 2008.

[VdWW02] R.F. Van der Wijngaart and P. Wong. NAS Parallel Benchmarks, Version 2.4. Technical Report NAS-02-007, National Aeronautics and Space Administration, 2002.

[VKR+08] Bart Vermeulen, Rolf Kuhnis, Jeff Rearick, Neal Stollon, and Gary Swoboda. Overview of Debug Standardization Activities. *IEEE Design and Test of Computers*, 25 :258–267, 2008.

[vOvdLKM00] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3) :78–85, 2000.

[VSN06] Visual studio 2005. In Laurence Moroney and Matthew MacDonald, editors, *Pro ASP.NET 2.0 in VB 2005*, pages 23–62. Apress, 2006.

[VWS11] A. Vetro, T. Wiegand, and G. J. Sullivan. Overview of the Stereo and Multiview Video Coding Extensions of the H.264/MPEG-4 AVC Standard. *Proceedings of the IEEE*, PP(99) :1 –17, 2011.

[WFM+06] Felix Wolf, Felix Freitag, Bernd Mohr, Shirley Moore, and Brian Wylie. Large Event Traces in Parallel Performance Analysis. In *8th Workshop Parallel Systems and Algorithms, Lecture Notes in Informatics, Frankfurt/Main*, pages 264–273, 2006.

[WM04] Felix Wolf and Bernd Mohr. EPILOG Binary Trace-Data Format. Technical Report FZJ-ZAM-IB-2004-06, Forschungszentrum Jülich, May 2004.

[Wol04] 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.