Un environnement unifié pour le développement sur puce à cœurs asymétriques

Roy Jamil

▶ **To cite this version:**

Roy Jamil. Un environnement unifié pour le développement sur puce à cœurs asymétriques. Autre [cs.OH]. ISAE-ENSMA Ecole Nationale Supérieure de Mécanique et d'Aérotechique - Poitiers, 2022. Français. NNT : 2022ESMA0003 . tel-03662259

HAL Id: tel-03662259

https://theses.hal.science/tel-03662259

Submitted on 9 May 2022

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.

THÈSE

pour l'obtention du Grade de

Docteur de l'Ecole Nationale Supérieure de Mécanique et d'Aérotechnique

(Diplôme National - Arrêté du 25 mai 2016)

Ecole Doctorale : Sciences et Ingéniérie des Systèmes, Mathématiques, Informatique
Secteur de Recherche : Informatique et Applications

Présentée par :

Roy JAMIL

****

Un environnement unifié pour le développement sur puce à cœurs asymétriques

****

Directeur de thèse : **Emmanuel Grolleau**

****

Soutenue le 4 Mars 2022
devant la Commission d'Examen

****

JURY

Président :
Nicolas NAVET Professeur, Université de Luxembourg, Luxembourg

Rapporteurs :
Liliana CUCU-GROSJEAN Directrice de recherche, INRIA, Paris
Jean-Philippe BABAU Professeur, Université de Bretagne Occidentale, Brest

Membres du jury :
Emmanuel GROLLEAU Professeur, ISAE - ENSMA, Poitiers
Antoine BERTOUT Maître de Conférences, Université de Poitiers, Poitiers

Invité :
Bernard DAUTREVAUX Directeur général, Ac6, Courbevoie

Remerciements

Je tiens d'abord à exprimer ma profonde gratitude au Professeur Emmanuel Grolleau, mon directeur de thèse. Je tiens à le remercier pour son suivi, son soutien, son temps, ses conseils et ses encouragements qui m'ont permis de mener à bien ce travail.

Je voudrais adresser tous mes remerciements aux Professeurs Liliana CUCU-GROSJEAN et Jean-Philippe BABAU, les rapporteurs de cette thèse qui me font l'honneur d'accepter d'évaluer ce travail. Je remercie également le Professeur Nicolas NAVET ainsi que Monsieur Antoine BERTOUT pour l'intérêt qu'ils portent à ce travail doctoral en participant à ce jury.

Je remercie toute l'équipe chez AC6 Ayoub, Kevin et tous les alternants, qui m'ont apporté une aide précieuse et un encouragement tout au long de ce travail de longue haleine. Un grand merci à Bernard pour m'avoir fait confiance et si bien accompagné tout au long de ce projet. Je n'aurais pas imaginé de meilleures conditions de travail pour cette thèse.

Je souhaite également adresser mes remerciements à tous les membres du laboratoire LIAS et au personnel de l'ENSMA.

Merci à mes amis Marc, Mathieu, Wael, Mike, Mehdi, Georges et Hamza d'avoir trouvé le temps de relire mon travail et grâce à qui j'ai pu avancer dans les moments difficiles.

Je remercie également les personnes qui ont contribué de près ou de loin à la réalisation de ce travail de recherche.

Je souhaiterais accorder une place toute particulière dans ces remerciements à Joséphine pour son support inconditionnel et le temps passé à mes côtés. Je n'y serais pas là sans toi. Merci pour tout.

Enfin, je tiens à exprimer ma reconnaissance et mon affection à mes parents, mon frère, ma grand-mère et ma tante pour leurs encouragements. Merci pour m'avoir soutenu dans tous mes choix et mes projets.

Je dédie cette thèse à William, la personne qui a cru en moi dès le début et qui a accepté sans aucune hésitation de m'embaucher et de m'avoir accordé cette opportunité.

Table des matières

Remerciements **3**

Table des matières **3**

Liste des figures **9**

1 Les systèmes multicœurs asymétriques **15**

1.1 Introduction 17

1.2 Systèmes embarqués 18

1.3 Systèmes temps réel 19

1.3.1 Types de systèmes embarqués temps réel 20

1.3.2 Applications des systèmes embarqués temps réel 20

1.3.3 Systèmes asymétriques et le temps réel 21

1.4 Multiprocesseurs asymétriques 22

1.5 Les avantages des systèmes multiprocesseurs asymétriques 22

1.6 Les défis de l'environnement de développement AMP 23

1.7 Besoin industriel 24

1.8 Les apports de la thèse 25

1.9 Organisation de la thèse 25

1.10 Publications scientifiques 26

2 Architectures opérationnelles des systèmes embarqués temps réel **27**

2.1 Introduction 29

2.2 Architectures opérationnelles des systèmes embarqués 29

2.2.1 Exécutif cyclique 30

2.2.2 Système contrôlé par les interruptions 31

2.2.3 Systèmes d'exploitation temps réel 32

2.3 Stratégies d'ordonnancement 35

2.3.1 Ordonnanceurs 35
2.3.2 Algorithmes d'ordonnancement 36
2.3.3 Ordonnancement multiprocesseurs 39
2.4 Systèmes d'exploitation ciblés 39
2.4.1 FreeRTOS 40
2.4.2 Aperçu du noyau Linux 42
2.5 Conclusion 49

3 Architecture des processeurs hétérogènes 51
3.1 Introduction 53
3.2 Pipeline 54
3.3 Mémoire virtuelle 55
3.4 Mémoire cache 55
3.4.1 Cohérence de cache 56
3.4.2 Le protocole Snooping 56
3.5 Interruptions 57
3.6 Bus et périphériques sur une puce 58
3.6.1 Bus 58
3.6.2 Périphériques 58
3.7 Architecture ARM 60
3.7.1 Comparaison entre le Cortex-M et le Cortex-A 61
3.7.2 Système sur une puce 63
3.7.3 Système sur une puce à architecture hétérogène 63
3.8 ARM big.LITTLE 64
3.8.1 Gestion de l'énergie sur big.LITTLE 66
3.8.2 DynamIQ 67
3.9 Système asymétrique hétérogène 68
3.9.1 Exemples de puces asymétriques hétérogènes 68
3.10 Communication inter-processeurs 70
3.10.1 Contrôleur de communication inter processeur 70
3.10.2 Les enjeux de la communication inter processeurs 71
3.10.3 Sémaphore matériel 72
3.11 Développement sur des processeurs ARM 73
3.12 Débogage 74
3.12.1 Débogueur ARM et JTAG 75

3.12.2 Débogueur GNU 76
3.13 Firmware 77
3.13.1 CMSIS 78
3.13.2 Couche d'abstraction matérielle 79
3.13.3 Protocole de transmission de messages asymétriques 79
3.14 Conclusion 83

4 Mesure de durée d'exécution 85
4.1 Introduction 87
4.2 Pire durée d'exécution 87
4.3 Méthodes d'analyse de WCET 88
4.3.1 Analyse statique 89
4.3.2 Analyse dynamique 91
4.3.3 Analyse hybride basée sur les mesures 92
4.3.4 Analyse probabiliste basée sur les mesures 93
4.3.5 Problèmes d'analyse du temps d'exécution sur les architectures modernes 93
4.3.6 Apport de la mesure à l'estimation de WCET 96
4.3.7 Les marges de sécurité 96
4.4 Caractéristiques des techniques de mesure 97
4.5 Méthodes de mesure sur un cœur 98
4.5.1 Chronomètre 98
4.5.2 Les outils de mesure du temps sous Linux 99
4.5.3 Fonctions standard C 101
4.5.4 Compteur de cycles d'horloge 101
4.5.5 Timer/Compteur sur puce 102
4.5.6 Analyseur logique 103
4.6 Comparaison expérimentale 104
4.7 Mesure expérimentale de durée d'exécution sur un système multicœur SMP 106
4.8 Contribution industrielle 108
4.9 Conclusion 108

5 Mesure de durée d'exécution et migration sur les systèmes AMP 111
5.1 Introduction 113
5.2 Application AMP 113
5.3 Mesure de durée d'exécution sur les systèmes AMP 115

5.4 Méthodes de mesure . . . 116
5.4.1 Résolution du timer et débordement . . . 116
5.4.2 Remontée des mesures vers la station de développement . . . 117
5.4.3 Méthodes de mesure possibles . . . 118
5.4.4 Validation expérimentale . . . 122
5.5 Migration hétérogène . . . 124
5.5.1 Introduction . . . 124
5.5.2 Méthode de migration hétérogène proposée . . . 124
5.6 Étude de cas ROSACE . . . 132
5.6.1 Charge normale . . . 133
5.6.2 ROSACE stressé . . . 133
5.7 Discussion et perspective . . . 136
5.8 Conclusion . . . 137

6 Environnement de développement unifié 139
6.1 Introduction . . . 141
6.1.1 Les éléments d'une distribution Linux embarqué . . . 141
6.1.2 Processus de démarrage d'un système basé sur Linux . . . 143
6.1.3 Construction automatisée d'une distribution Linux . . . 145
6.2 SW4Linux . . . 148
6.2.1 Aperçu . . . 150
6.2.2 Sécurité et démarrage sécurisé . . . 151
6.2.3 Développement des pilotes Linux . . . 152
6.2.4 Plugin de mesure de temps d'exécution . . . 154
6.3 Outils de développement des microcontrôleurs . . . 157
6.4 Environnement de développement unifié . . . 158
6.4.1 Développement d'applications AMP . . . 159
6.4.2 Débogage des applications AMP . . . 160
6.5 Conclusion . . . 161

7 Conclusion et perspectives 163
7.1 Aperçu de la thèse . . . 165
7.2 Mesure de durée d'exécution . . . 165
7.2.1 Étude sur les méthodes de mesure sur un cœur . . . 165
7.2.2 Contribution industrielle . . . 167

7.3 Application AMP 167
7.3.1 Mesure de temps d'exécution d'un message AMP 167
7.3.2 Migration d'une tâche entre processeurs hétérogènes 168
7.3.3 Perspectives 168
7.4 Environnement de développement unifié 169
7.5 Fiabilisation des mesures et analyses de durées 169

Bibliographie **171**

Liste des figures

2.1 Flux d'exécution d'un exécutif cyclique 31
2.2 Flux d'exécution d'un système contrôlé par les interruptions 32
2.3 Systèmes d'exploitation temps réel 33
2.4 Changement de contexte 35
2.5 Ordonnancement des deux taches en utilisant un ordonnanceur préemptif . 36
2.6 Ordonnancement des deux taches en utilisant un ordonnanceur non préemptif 36
2.7 Noyau Linux 43
2.8 Arbre rouge-noir 45
2.9 Organisation du flux de données et du contrôle dans un système Linux à double noyau 46
2.10 Hiérarchie POSIX 49

3.1 Séquence d'exécution pour une architecture non pipelinée 54
3.2 Séquence d'exécution pour une architecture pipelinée 55
3.3 Bus et périphériques du SoC 59
3.4 Interconnexion de big.LITTLE 66
3.5 Cœurs virtuels big.LITTLE 67
3.6 Schéma bloc simplifié de la carte STM32MP1 68
3.7 Schéma bloc simplifié de la carte Zynq UltraScale+ MPSoC 69
3.8 Contrôleur de communication inter processeurs 71
3.9 Procédure de verrouillage du sémaphore matériel 73
3.10 Débogage croisé 77
3.11 Aperçu de la structure du firmware 78
3.12 Modèle de transmission de messages 79
3.13 Ensemble de configurations d'OpenAMP 81
3.14 Couches du protocole RPMsg 82

4.1 Distribution possible du temps d'exécution par rapport au BCET et au WCET . . . 88
4.2 Temps d'exécution mesurés par rapport au BCET et au WCET . . . 89
4.3 Graphe de flot de contrôle . . . 90
4.4 Distribution du temps d'exécution selon des différentes méthodes . . . 105
4.5 Distribution du temps d'exécution sous Linux . . . 106
4.6 Système multicœur symétrique . . . 107

5.1 Application AMP utilisant des cœurs hétérogènes . . . 114
5.2 Méthode de mesure : gauche-droite . . . 118
5.3 Méthode de mesure : gauche-droite-acknowledge . . . 119
5.4 Méthode de mesure : gauche-droite-hsem avec sémaphore matériel . . . 119
5.5 Méthode de mesure : zero-gauche-droite . . . 120
5.6 Méthode de mesure : gauche-droite-gauche . . . 120
5.7 Méthode de mesure : zero-gauche-droite-gauche . . . 121
5.8 Méthode de mesure : start-stop . . . 121
5.9 Méthode de mesure : zero-gauche-droite-stop . . . 122
5.10 Mesure du temps de communication d'une application AMP . . . 123
5.11 Exécution d'une tâche sur des cœurs hétérogènes . . . 125
5.12 Temps de migration intra-cluster . . . 128
5.13 Modèle de plateforme plate versus modèle de plateformes multicœurs non liées . . . 129
5.14 Distribution du temps d'exécution de différents programmes . . . 131
5.15 Temps d'exécution du ROSACE en charge normale . . . 134
5.16 Mesure du temps d'exécution de ROSACE en charge normale . . . 134
5.17 Temps d'exécution de ROSACE dans un environnement stressé . . . 135
5.18 Mesure du temps d'exécution de ROSACE dans un environnement stressé . . . 135
5.19 Système HMPSoC avec un ordonnanceur AMP . . . 137

6.1 Séquence de démarrage des systèmes embarqués . . . 143
6.2 Étapes de démarrage du système Linux . . . 144
6.3 Menu de configuration de Buildroot . . . 146
6.4 Définition du package busybox : SW4Linux (gauche) vs Yocto (droite) . . . 149
6.5 Construire une image Linux embarquée en utilisant SW4Linux . . . 150
6.6 SW4Linux après le build . . . 151
6.7 Les options du mode de démarrage dans SW4Linux . . . 152

6.8 Menu principal de création d'un nouveau pilote Linux 153
6.9 Sélection de la méthode de mesure 154
6.10 Complétion du code de mesure 155
6.11 Les valeurs d'entrée de la fonction à mesurer 156
6.12 Un extrait du fichier de résultats de mesure 156
6.13 Fonctionnalités de l'IDE unifié 158
6.14 Menu de configuration du noyau 160
6.15 Deux sessions de débogage simultanées sur des cœurs hétérogènes 161

Liste des acronymes

AMP *Asymmetric Multi-Processing*. 21, 22, 23, 24, 25, 40, 53, 63, 64, 79, 80, 113, 115, 116, 123, 124, 126, 132, 133, 135, 136, 137, 138, 158, 159, 160, 165, 167, 168

API *Application Programming Interface*. 39, 41, 45, 46, 81, 125

CFG *Control Flow Graph*. 89, 90

CFS *Completely Fair Scheduler*. 42, 44, 45

CMSIS *Common Microcontroller Software Interface Standard*. 77, 78

DVFS *Dynamic Voltage and Frequency Scaling*. 62, 63, 66, 106

EDF *Earliest deadline first*. 37, 38

FIFO *First In First Out*. 44

FPGA *Field Programmable Gate Arrays*. 53, 63, 64, 68, 69, 154

GPU *Graphics Processing Unit*. 53, 63, 95

HAL *Hardware abstraction layer*. 77, 79

HMPSoC *Heterogeneous Multiprocessor System on a Chip*. 17, 21, 25, 53, 68, 70, 93, 96, 113, 115, 118, 122, 124, 125, 128, 129, 130, 137, 138, 141, 152, 154, 157, 158, 161, 166, 167, 168, 169

IDE *Integrated Development Environment*. 141, 148, 149, 157, 158, 160, 161, 165, 169

IoT *Internet of Things*. 21, 68

IPC *Inter-process communication*. 33, 47, 79

IPCC *Inter-Processor Communication Controller*. 69, 70

ISA *Instruction Set Architecture*. 60, 64, 65, 68, 113, 124, 125, 136, 138, 168

ISR *Interrupt Service Routine*. 32, 34

JTAG *Portable Joint Test Action Group*. 75

MMU *Memory Management Unit*. 33, 55, 60, 167

OpenAMP *Open Asymmetric Multi Processing*. 79, 80, 81, 114, 124, 167

OS *Operating system*. 32, 34

PLL *Phase-Locked Loop*. 80

POSIX *Portable Operating System Interface*. 48

RTOS *Real Time Operating System*. 24, 32, 33, 37, 39, 40, 80

SMP *Symmetric Multi-Processing*. 19, 23, 53, 106, 107, 113, 124, 132, 133, 138, 168

SoC *System on a Chip*. 17, 23, 24, 53, 58, 59, 63, 64, 66, 68, 80, 83, 113, 138, 141, 147, 148, 157, 165, 166, 167

SW4Linux *System Workbench for Linux*. 108, 109, 141, 148, 149, 150, 151, 152, 153, 154, 158, 159, 160, 161, 165, 167, 169

WCET *Worst Case Execution Time*. 17, 38, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 107, 108, 115, 124, 129, 166, 169

Chapitre 1

Les systèmes multicœurs asymétriques

Sommaire

1.1 Introduction 17
1.2 Systèmes embarqués 18
1.3 Systèmes temps réel 19
1.3.1 Types de systèmes embarqués temps réel 20
1.3.2 Applications des systèmes embarqués temps réel 20
1.3.3 Systèmes asymétriques et le temps réel 21
1.4 Multiprocesseurs asymétriques 22
1.5 Les avantages des systèmes multiprocesseurs asymétriques 22
1.6 Les défis de l'environnement de développement AMP 23
1.7 Besoin industriel 24
1.8 Les apports de la thèse 25
1.9 Organisation de la thèse 25
1.10 Publications scientifiques 26

1.1 Introduction

Le domaine de l'architecture des processeurs a évolué au cours de la dernière décennie pour englober les conceptions à plusieurs cœurs (les puces multicœurs) et plus que jamais, l'hétérogénéité des ressources en termes de diversité de cœurs de calcul dans les systèmes sur une puce. Ce type de puces est devenue tellement utilisé qu'on le trouve quasiment partout, des serveurs, aux ordinateurs de bureau et ordinateurs portables, des appareils mobiles jusqu'à l'internet des objets. Chacun de ces appareils peut être utilisé pour exécuter différentes charges de travail dans des circonstances spécifiques, et dans le respect de certaines contraintes. Les technologies des transistors permettent de combiner encore plus de types de processeur, des processeurs graphique, d'accélérateurs et de mémoire cache sur le même système sur une puce ou *System on a Chip* (SoC).

Pour les systèmes embarqués, les puces hétérogènes récentes, *Heterogeneous Multiprocessor System on a Chip* (HMPSoC), ajoutent généralement un processeur à basse consommation et optimisé pour les applications temps réel. Cela ouvre la possibilité de créer de nouvelles applications qui n'étaient pas possibles auparavant ou de faire la même application de manière plus efficace. Mais cela entraîne de nombreux défis à plusieurs niveaux, en particulier pour les développeurs, car il n'existe pas d'outil de développement capable de gérer le développement de l'ensemble du système. Un autre défi concerne les applications temps réel, car ces plateformes sont très complexes ce qui rend plus difficile l'estimation du temps d'exécution du code.

Le problème de la mesure de durée, sans parler de la pire durée, d'exécution des traitements pris en isolation est aussi un problème plus complexe sur les HMPSoCs que sur les processeurs classiques, ne serait-ce que d'un point de vue de l'outillage et du déploiement de l'application à mesurer dans son environnement d'exécution (notamment son système d'exploitation). L'objectif principal de cette thèse est de fournir et outiller un environnement de développement unifié pour les HMPSoCs offrant des possibilités simples d'emploi de mesures logicielles de durées d'exécutions. Le choix d'offrir des mesures logicielles, à l'opposé de matérielles, requérant du matériel spécialisé, se justifie par la volonté de gratuité et de disponibilité de ces moyens dès la sortie de nouveaux SoCs. Les moyens de mesure fournis permettent de mener de façon unifiée des campagnes de mesures de programmes pris en isolation sur différents types de coeurs, que ceux-ci soient stressés par ailleurs ou non. De plus, des moyens permettant d'évaluer les durées de communication entre coeurs ou entre ensemble (nous parlerons de clusters) de coeurs différents permettront d'évaluer la durée de migration d'un traitement entre deux types de coeurs différents. La thèse apporte ainsi une étude de faisabilité de migrations entre clusters différents et les moyens de mesurer expérimentalement le surcoût que cela engendre. Cette thèse n'est cependant pas une thèse qui propose des analyses de *Worst Case Execution Time* (WCET) (pire durée d'exécution), mais ses contributions pourront être utilisées pour aider à mener les campagnes de mesures nécessaires à différentes familles d'analyse de WCET.

1.2 Systèmes embarqués

Un système embarqué est un système informatique spécialisé qui est généralement chargé de procéder un procédé, et qui s'exécute sur une plateforme pouvant être alimentée sur batterie. Un système embarqué consiste en une combinaison de composants matériels et logiciels capable d'exécuter une fonction spécifique. Contrairement aux ordinateurs de bureau qui sont conçus pour être à usage général, les systèmes embarqués sont limités dans leur application. Les systèmes anti-blocages des roues (ABS), les drones, les appareils photo numériques, les smartphones sont quelques exemples de systèmes embarqués [119].

Les systèmes embarqués fonctionnent souvent dans des environnements réactifs soumis à des contraintes temporelles, cela est particulièrement vrai lorsqu'ils sont dédiés au contrôle de procédé dynamique, comme des systèmes avioniques, spatiaux ou automobiles. Un système embarqué peut être divisé en deux parties : le matériel, qui fournit les performances nécessaires à l'application (et d'autres propriétés du système comme la sécurité), et le logiciel, qui fournit la majorité des fonctionnalités et de la flexibilité du système.

Les principaux composants matériels d'un système embarqué sont :

- **Processeur** : le ou les cœurs de calcul sont les éléments fondamentaux du système embarqué. Cela peut aller d'un simple microcontrôleur 8 bits à un microprocesseur multicœur de 32 ou 64 bits. Le concepteur de systèmes embarqués doit choisir le module le plus efficace pour l'application qui soit capable de répondre à toutes les exigences fonctionnelles et non fonctionnelles [1] (comme les contraintes temps réels).
- **Capteurs et actionneurs** : les capteurs sont utilisés pour capter des informations sur l'environnement. Les actionneurs sont utilisés pour agir sur l'environnement d'une manière ou d'une autre.
- **La mémoire** : est une partie importante d'un système embarqué et les applications embarquées peuvent l'épuiser en fonction de l'application. Il existe de nombreux types de mémoire, classifiées en volatile et non volatile, qui sont utilisés par les systèmes embarqués.
- **Entrées et sorties** : Il s'agit d'un connecteur physique qui peut être utilisé comme entrée pour les communications extérieur vers processeur, ou sortie pour les communications processeur vers extérieur.
- **Connecteur de débogage** : généralement appelé JTAG (Joint Test Action Group), il est utilisé pour un accès de bas niveau aux ressources du système embarqué telles que la lecture et l'écriture des registres, la lecture et l'écriture de la mémoire et le contrôle complet de l'exécution de la cible avec des opérations telles que exécuter, continuer ou faire un pas.

Les systèmes embarqués sont souvent définis comme des systèmes réactifs. Un système réactif doit utiliser une combinaison de matériel et de logiciel pour répondre aux

1. Les exigences sont habituellement qualifiées de fonctionnelles si elles concernent des fonctionnalités que le système doit posséder, et non fonctionnelles si elles concernent des propriétés que doivent posséder des fonctionnalités (contraintes de temps, d'énergie dissipée, de sécurité, etc.)

événements de l'environnement dans des contraintes bien déterminées. Le fait que ces événements externes puissent être périodiques et prévisibles ou apériodiques et difficiles à prévoir complique la situation.

Les systèmes embarqués présentent plusieurs caractéristiques essentielles :

- **Surveillance et réaction à l'environnement** : les systèmes embarqués reçoivent généralement des informations en lisant les données de capteurs d'entrée. Il existe de nombreux types de capteurs qui surveillent divers signaux analogiques dans l'environnement, par exemple la température, la pression et les vibrations. Ces données sont traitées par des fonctions faisant partie de chaînes fonctionnelles. Les résultats peuvent être affichés sous une forme ou une autre à un utilisateur ou simplement utilisés pour commander des actionneurs (comme le déploiement des airbags après un accident de voiture).
- **Contrôle de l'environnement** : les systèmes embarqués peuvent générer et transmettre des commandes qui contrôlent des actionneurs tels que des moteurs, les systèmes drive-by-wire [26], etc.
- **Traitement de l'information** : le traitement des données collectées par les capteurs, comme par exemple la pression sur la pédale de frein, le contrôleur en boucle fermée pour le régulateur de vitesse, etc.
- **Ressources limitées** : les systèmes embarqués sont optimisés en fonction de l'application, ce qui signifie qu'un grand nombre des ressources sont limitées, comme les cycles du processeur, la mémoire, l'énergie, etc.
- **Temps réel** : les systèmes embarqués doivent répondre au caractère temps réel de l'environnement dans lequel ils fonctionnent afin de respecter les contraintes de temps inhérentes à la dynamique du procédé ou des dynamiques internes aux éléments matériels ou logiciels, tout retard dans le traitement pourrait entraîner une défaillance du système.

Les processeurs multicœurs ont été initialement utilisés sous la forme *Symmetric Multi-Processing* (SMP) où plusieurs cœurs de même type et de même architecture sont regroupés et interconnectés, ils partagent la même mémoire vive et ils peuvent accéder aux mêmes périphériques et éléments de la même puce. Les systèmes SMP représentent la forme la plus utilisée des multicœurs et ce groupe de cœurs est appelé cluster. Il n'y a pas de définition officielle d'un processeur ou d'un CPU, dans cette thèse ils pourraient être un seul cœur ou un cluster.

1.3 Systèmes temps réel

Les systèmes temps réel sont des logiciels soumis à des contraintes de temps inhérentes à la dynamique d'un objet ou procédé contrôlé. De nombreux systèmes temps réel sont embarqués, et de type contrôle-commande.

Une plateforme d'exécution est dite déterministe si l'exécution d'un même traitement

prend le même ordre de grandeur de temps pour exécuter un même traitement, quel que soit l'état interne de la plateforme. Un système temps réel sera d'autant plus simple à valider qu'il s'exécute sur une plateforme déterministe car il est synchronisé avec le monde extérieur. S'il n'y est plus synchronisé, cela signifie qu'il a échoué. Il n'y a aucune implication de vitesse ou de temps de réponse dans la notion de temps réel. Un système en temps réel peut être lent, mais il doit avoir la garantie de répondre dans un délai borné et connu. Certains processeurs sont plus déterministes que d'autres, ils sont généralement beaucoup plus simples et tournent à une fréquence plus basse, ce point sera détaillé dans le chapitre 3.

1.3.1 Types de systèmes embarqués temps réel

Il existe essentiellement deux types de systèmes embarqués temps réel, le temps réel dur et le temps réel mou :

- **Temps réel dur** : Les systèmes embarqués temps réel dur doivent absolument respecter toutes leurs contraintes de temps. Cela implique que tous les retards du système sont strictement prohibés. Ces systèmes temps réel sont utilisés dans différents domaines tels que les avions et les missiles, etc. Cette définition très académique tranche avec la vision de criticité mixte présente dans l'industrie [97] ([DO-178C]) pour laquelle on pourra donc parler de temps réel dur pour des sous-systèmes (comme les sous-systèmes DAL A ou B par exemple).
- **Temps réel mou** : Les systèmes embarqués temps réel mous sont moins restrictifs que les systèmes temps réel durs. Toutefois, le principe de base est le même, par exemple, une fonctionnalité critique doit être exécutée dans le délai prévu. Toutefois, ce délai peut être un peu flexible. Ces systèmes temps réel sont utilisés dans différents domaines comme les projets scientifiques, le multimédia, les drones de loisir, etc. Nous souhaitons un système qui réponde dans des délais acceptables, sans qu'un dépassement soit très pénalisant.

1.3.2 Applications des systèmes embarqués temps réel

Il existe différentes applications des systèmes embarqués temps réel. Voici quelques-unes de ces applications : [49]

- Systèmes de contrôle des véhicules pour les chemins de fer, les navires, les avions, les automobiles, etc.
- Opérations spatiales telles que le contrôle des stations spatiales, le lancement et la surveillance des vaisseaux spatiaux, etc.
- Les opérations militaires comme les bases de contrôle militaires, les tirs de missiles, etc.
- Les systèmes de contrôle des bâtiments qui gèrent les ascenseurs, les portes, le chauffage, etc.

- Les systèmes multimédias qui fournissent, vidéo, audio, graphique, texte, etc.
- Les systèmes robotiques et d'intelligence artificielle.
- La radio, les communications par satellite le téléphone.
- Systèmes médicaux pour les traitements cardiaques, la surveillance des patients, la radiothérapie, etc.

1.3.3 Systèmes asymétriques et le temps réel

De nombreux systèmes asymétriques contiennent au moins un processeur optimisé pour les systèmes temps réel. Ces systèmes permettent de créer de nouvelles applications qui n'étaient pas possibles auparavant, ou de créer des systèmes beaucoup plus optimisés qui exécutent les applications de manière beaucoup plus déterministe en utilisant moins d'énergie que s'ils étaient exécutés sur de puissants processeurs d'application.

Voici quelques exemples d'applications qui pourraient être réalisées efficacement avec les systèmes *Asymmetric Multi-Processing* (AMP) qui ont par exemple un cluster de haute performance à usage général et un microcontrôleur plus déterministe à faible consommation :

- Dans un drone, la partie vol et contrôle est exécutée sur le microcontrôleur, cette partie doit être en exécution continue et doit être très déterministe. La deuxième partie est celle des applications de plus haut niveau telles que la communication et le traitement de l'image, cette partie restera en mode veille (en arrêtant les cœurs) la plupart du temps et ne s'exécutera que si nécessaire. Ceci permettra de n'avoir qu'une seule puce capable de faire à la fois les calculs de haute performance et la partie temps réel avec une faible consommation d'énergie. Un second type de déploiement pourrait être un autopilote avancé et très gourmand en termes de puissance de calcul exécuté sur un cluster haute performance, alors qu'un second autopilote de secours peut s'exécuter sur le microcontrôleur.
- Dans les robots, la partie temps réel, comme le contrôle des mouvements du bras robotique, est effectuée par un processeur optimisé pour le temps réel, tandis que la reconnaissance vocale et le traitement d'images sont exécutées par des cœurs à haute performance.
- Dans l'internet des objets, *Internet of Things* (IoT), où il existe des applications avec des besoins intermittents de performance, il est important d'obtenir une faible consommation d'énergie. Ces systèmes passent la plupart de leur temps à attendre un événement (qui nécessite du traitement) afin de maximiser la durée de vie, surtout si l'énergie est captée dans l'environnement pour recharger la batterie. Avec un HMPSoC, les cœurs à faible consommation peuvent être utilisés à cette fin, puis, lorsque l'événement arrive, les cœurs à haute performance peuvent être utilisés pour les besoins de traitement intensif comme le traitement de vidéo ou de l'intelligence artificielle.

1.4 Multiprocesseurs asymétriques

Dans le cas d'un système AMP, la puce contient des cœurs d'architecture différente et interconnectés, qui partagent certains périphériques mais pas tous. Il y a au moins deux côtés formés d'un cœur ou d'un cluster où chacun a ses propres caractéristiques. AMP est un style de calcul hétérogène conçu pour répondre aux exigences et aux besoins en puissance de calcul qui peuvent être variables en fonction de la charge et adaptés à des applications spécifiques. Les systèmes hétérogènes utilisent un ou plusieurs types de processeurs ensemble afin de fournir de meilleurs rendements dans un certain nombre de situations différentes [62]. En incorporant plusieurs processeurs spécialisés dans le traitement de différentes tâches, le système hétérogène est, dans l'ensemble, plus efficace en termes de consommation d'énergie et de performances. L'architecture hétérogène offre de nombreux avantages par rapport à ses homologues homogènes, bien qu'elle soit de plus en plus complexe et qu'elle présente de nouveaux défis.

Les systèmes AMP peuvent se servir de la configuration multiprocesseur du processeur soit au niveau du matériel, soit au niveau du système d'exploitation. Par exemple, le système peut dédier des fonctions à chaque cœur. Un processeur peut gérer uniquement le code système tandis qu'un autre peut gérer les fonctions d'entrées et sorties. D'autres types de systèmes AMP peuvent utiliser n'importe quel cœur pour n'importe quelle fonction en les traitant de manière symétrique en ce qui concerne les rôles des processeurs. Cependant, ils peuvent être traités de manière asymétrique en ce qui concerne les périphériques.

Les systèmes AMP ont une architecture multicœur qui suit normalement un concept, en général, maître-esclave pour l'exécution des processus. Chaque processeur peut exécuter un système d'exploitation, des applications ou des logiciels différents, ce qui permet d'obtenir de meilleures performances. Par exemple, il est possible d'avoir un système AMP où certains processeurs ont un usage général et d'autres sont dédiés à l'exécution d'une application avec des contraintes temps réel.

1.5 Les avantages des systèmes multiprocesseurs asymétriques

Les systèmes multiprocesseurs asymétriques existent depuis les années 1960 et constituent une solution efficace et économique pour augmenter la puissance et les performances des systèmes informatiques. De plus, ils offrent de nombreux avantages et bénéfices [29], y compris :

- **Économie d'énergie :** L'architecture hétérogène est utilisée pour offrir un certain nombre d'avantages, notamment une faible consommation d'énergie et une puissance de traitement importante. Le processeur à faible consommation d'énergie peut être utilisé pour exécuter les tâches de routine et le processeur à haute consommation d'énergie peut être réveillé et utilisé uniquement en cas de besoin.
- **Peut exécuter plusieurs systèmes d'exploitation et applications :** L'un des avan-

tages les plus significatifs de l'utilisation d'un système AMP est la possibilité d'exécuter plusieurs systèmes d'exploitation et applications sur son architecture hétérogène. Par exemple, Linux peut être installé sur un processeur de haute performance tandis que FreeRTOS peut être exécuté sur l'autre processeur moins performant. Ceci permet par exemple de programmer un système avec mode dégradé en cas d'erreur : on peut imaginer un autopilote de drone avec fonctionnalités évoluées s'exécutant sous Linux, profitant ainsi de bibliothèques logicielles riches, incluant traitement vidéo et intelligence artificielle, et un coeur de repli avec un autopilote basique s'exécutant sous FreeRTOS sur le microcontôleur.

- **Robustesse :** Dans le traitement symétrique, si un ou plusieurs processeurs tombent en panne, la charge est répartie uniformément entre les processeurs restants, ce qui réduit les performances. Cependant, les systèmes AMP ont une solution de secours lorsque leurs processeurs maîtres tombent en panne. Tout processeur esclave peut reprendre son rôle et poursuivre l'exécution des tâches normalement. Ils sont donc utiles pour les systèmes à mission critique qui ont une exigence de haute fiabilité.

Les systèmes AMP offrent donc principalement une versatilité permettant de ne consommer que l'énergie dont on a besoin, et ainsi maximiser la durée de vie d'un système sur batterie et/ou l'énergie ambiante. De plus, ils peuvent offrir des solutions hybrides pour des systèmes embarqués offrant deux systèmes sur un même SoC.

1.6 Les défis de l'environnement de développement AMP

L'AMP présente plusieurs défis [55] en raison de la nouveauté et de la complexité du système. Ces défis n'existaient pas lors du développement des systèmes SMP, où les développeurs peuvent programmer, tester et optimiser leur application dans les limites de leur système.

Aujourd'hui, avec les architectures hétérogènes, plusieurs défis se posent aux développeurs embarqués. Parmi ceux-ci, on peut citer :

- **Démarrage** : Chaque processeur a sa propre séquence de démarrage qui dépend également de son système d'exploitation spécifique. Dans le cas d'un multicœur hétérogène, cette séquence de démarrage doit prendre en considération les différents systèmes d'exploitation installés sur les différents processeurs [102] et mettre en place chaque partie du système de manière coordonnée en fonction des exigences de l'application. De plus, elle doit prendre en compte le matériel partagé sur la puce comme la mémoire principale ou les mémoires dédiées, différentes fréquences de bus principaux, entrées/sorties, etc.
- **Débogage** : En combinant les divers systèmes, les développeurs ont besoin d'un moyen de comprendre comment chaque système d'exploitation et chaque application fonctionne. Ils doivent comprendre où ils sont confrontés à des conflits de ressources partagées ou à la saturation du processeur, du bus ou du périphérique. Ils ont besoin

d'un moyen d'analyser le comportement des applications de chaque côté du système, ce qui permet d'optimiser les performances globales du système.

- **Séparation des préoccupations** : Les systèmes AMP offrent une excellente opportunité d'avoir un moyen peu onéreux de réaliser la séparation matériel/logiciel des préoccupations au sens de la norme ISO 26262 [45]. En effet, l'utilisation de plusieurs *Real Time Operating System* (RTOS) différents sur les différents clusters de cœurs asymétriques peut être utilisée pour garantir le fonctionnement du système, en cas de défaillance d'un côté ou de l'autre due à une mauvaise programmation ou un problème matériel. Même dans ce cas, l'autre partie du système mixte ne sera pas affectée et protégera l'ensemble du système contre les défaillances. Cela pourrait prouver son utilité, par exemple dans le cas des systèmes de drones où les développeurs de systèmes cherchent des moyens abordables pour assurer la séparation des préoccupations.
- **Sécurité** : La nature partagée des dispositifs sur le SoC ajoute des moyens d'isolation supplémentaires pour protéger le système contre les attaquants. Les ressources partagées critiques doivent être isolées, en utilisant les technologies matérielles dédiés à cette fin comme la TrustZone pour les processeurs ARM, et ne doivent être accessibles que par un processeur spécifique.
- **Communication et synchronisation entre processeurs** : Lorsque nous avons commencé à travailler sur un réel SoC AMP, nous avons constaté que la communication entre les cœurs hétérogènes, utilisant différents RTOS, était nécessaire pour répondre aux besoins de développement ; en outre, elle devait offrir des garanties temps réel si le système devait être utilisé dans un contexte temps réel. Un framework de communication inter-processeurs temps réel qui fonctionne sur plusieurs environnements logiciels est nécessaire pour gérer les restrictions dues au système asymétrique.

1.7 Besoin industriel

Les systèmes asymétriques, en particulier ceux composés d'un processeur à usage général et d'un microcontrôleur, sont devenus très populaires depuis quelques années seulement. Il n'existe actuellement aucun outil de développement qui permette de développer toutes les applications pour tous les processeurs. Il faudrait généralement utiliser plusieurs outils, au moins un pour le développement sur le microcontrôleur, un pour générer la distribution et le système d'exploitation qui vont être déployés sur le côté haute performance et un pour développer ses applications. Ac6 a financé cette thèse dans le cadre d'un contrat CIFRE, son objectif est de soutenir la recherche et le développement nécessaires pour créer un environnement de développement capable de traiter tous les défis de développement AMP cités dans la section précédente auxquels les développeurs d'aujourd'hui sont confrontés, en particulier lorsqu'ils travaillent avec des distributions Linux embarquées. AC6 a déjà développé deux outils basés sur Eclipse, le premier est utilisé par les développeurs de microcontrôleurs, le second est un outil de construction d'une dis-

tribution et de développement Linux. L'objectif est de combiner les deux outils dans un environnement unifié pour le développement sur puces hétérogènes ; en outre, il doit être capable de mesurer le temps d'exécution du code, car cet outil sera utilisé par les développeurs de systèmes embarqués qui ont parfois besoin de créer des applications respectant des contraintes de temps ou avoir une idée des performances.

1.8 Les apports de la thèse

Nous allons voir la structure d'une application AMP qui utilise tous les processeurs disponibles dans les HMPSoCs, ainsi que le mécanisme de communication AMP et les ressources partagées. Plusieurs méthodes de mesure de temps d'exécution d'un message AMP ont été proposées et comparées en mettant en avant les principaux choix et critères. Nous avons également proposé et mis en œuvre une méthode de migration d'une tâche entre processeurs hétérogènes qui permet à une tâche de migrer entre des cœurs AMP. Le temps de latence de communication a été mesuré à l'aide de la méthode de mesure proposée. Enfin, une étude de cas a été réalisée, dans laquelle nous avons comparé le temps d'exécution entre une même application avec et sans la migration d'une tâche entre processeurs hétérogènes.

1.9 Organisation de la thèse

Cette thèse se compose de sept chapitres et s'organise comme suit :

- **Chapitre 2** : présente les architectures logicielles pour les systèmes embarqués et les algorithmes d'ordonnancement d'une application temps réel dans un système multitâche.
- **Chapitre 3** : décrit l'architecture matérielle des différents types de processeurs, ainsi que les différents composants physiques qui peuvent être trouvés dans une puce. Il aborde plus particulièrement les systèmes asymétriques hétérogènes et leurs périphériques spécifiques.
- **Chapitre 4** : présente l'état de l'art des principales méthodes de mesure du temps d'exécution, puis une comparaison expérimentale entre les différentes méthodes.
- **Chapitre 5** : propose une méthode de migration des tâches entre processeurs hétérogènes et une méthode pour mesurer la durée de cette migration.
- **Chapitre 6** : présente l'environnement de développement unifié pour les puces asymétriques, qui a été développé, ainsi que la fonctionnalité qui permet de mesurer le temps d'exécution sur une cible.
- **Chapitre 7** : présente une conclusion générale en résumant toutes le contributions et donne les perspectives de recherche.

1.10 Publications scientifiques

Cette thèse a donné lieu à deux publications scientifiques :

- Jamil R, Grolleau E, Dautrevaux B, Bertout A. Measurement-based timing analysis on heterogeneous mpsocs : A practical approach. InEuropean Conference on Software Architecture 2020 Sep 14 (pp. 279-293). Springer, Cham.
- Bertout A, Goossens J, Grolleau E, Jamil R, Poczekajlo X. Workload assignment for global real-time scheduling on unrelated clustered platforms. Real-Time Systems. 2021 May 15 :1-32.

Chapitre 2

Architectures opérationnelles des systèmes embarqués temps réel

Sommaire

2.1 Introduction 29
2.2 Architectures opérationnelles des systèmes embarqués 29
2.2.1 Exécutif cyclique 30
2.2.2 Système contrôlé par les interruptions 31
2.2.3 Systèmes d'exploitation temps réel 32
2.3 Stratégies d'ordonnancement 35
2.3.1 Ordonnanceurs 35
2.3.2 Algorithmes d'ordonnancement 36
2.3.3 Ordonnancement multiprocesseurs 39
2.4 Systèmes d'exploitation ciblés 39
2.4.1 FreeRTOS 40
2.4.2 Aperçu du noyau Linux 42
2.5 Conclusion 49

2.1 Introduction

Les systèmes embarqués, comme expliqué dans la section 1.2, sont une combinaison de matériel et de logiciels informatiques, et peut-être d'autres pièces mécaniques, conçus pour exécuter une fonction spécifique. Dans certains cas, les systèmes embarqués font partie d'un système ou d'un produit plus grand, comme dans le cas d'un système de freinage antiblocage dans une voiture. Ils sont constitués de plusieurs composants dont le plus important est le processeur. Ces ressources sont limitées, généralement non réentrantes [1] et ne peuvent, de notre point de vue, effectuer qu'une seule chose à la fois, c'est-à-dire exécuter seulement une seule instruction à la fois sur un cœur du processeur.

Les processeurs ont des contraintes de taille, de coût, de consommation d'énergie et de dissipation de chaleur. Les processeurs haut de gamme sont chers et consomment beaucoup d'énergie avec une importante dissipation de chaleur, ils ne sont donc pas adaptés pour être utilisés partout. Comme un processeur ne peut exécuter qu'une fonction à la fois et les ressources sont limitées, une solution logicielle est nécessaire, dont le rôle est de trouver le rythme d'exécution le plus adapté à l'application. Cette solution permet de profiter des ressources de la manière la plus efficace possible et répond à plusieurs besoins, comme l'exécution entrelacée de plusieurs fonctions ou tâches ou encore l'utilisation d'un algorithme d'exécution complexe. L'architecture logicielle est devenue un domaine de recherche important au cours des dernières décennies [88].

Par exemple, pour contrôler un drone, le processeur doit traiter les données reçues de nombreux capteurs comme les accéléromètres et les gyromètres, il doit également contrôler les hélices via la vitesse du moteur. Dans ce cas, le processeur n'a pas besoin d'être très rapide, il doit avoir une vitesse suffisante pour tout faire et consommer moins d'énergie afin de prolonger la durée de vie de la batterie (typiquement de l'ordre de la centaine de MHz). En outre, nous devons utiliser une architecture logicielle qui permet de traiter les différentes tâches en parallèle et de les ordonnancer de façon à assurer des temps de réponse (délai s'écoulant entre activation et terminaison d'un travail) suffisamment court pour respecter les échéances.

Ce chapitre a pour but d'examiner les divers aspects de développement d'un système embarqué, notamment les architectures opérationnelles et logicielles des systèmes embarqués, les stratégies d'ordonnancement et les systèmes d'exploitation utilisés sur des cibles embarquées. Pour discuter de ces aspects, une revue de littérature est menée, pour présenter les travaux réalisés.

2.2 Architectures opérationnelles des systèmes embarqués

Il existe plusieurs architectures opérationnelles, chacune ayant ses avantages et ses inconvénients. Les critères des algorithmes utilisés sont divers, comme la gestion des in-

1. Les ressources non réentrantes peuvent avoir seulement un seul utilisateur.

terruptions, ou encore l'exécution des tâches de manière séquentielle ou parallèle, tout en tenant compte du temps de transition d'une tâche à l'autre. En outre, il faut savoir quand et comment mettre le processeur en mode économie d'énergie.

Parmi ces algorithmes, nous pouvons citer l'exécutif cyclique qui peut être utilisé sur les applications les plus basiques et simples qui ne nécessitent pas d'exécuter des tâches en parallèle, ou d'autres qui sont basés sur des interruptions, ou encore le plus complexe qui est le système d'exploitation dans lequel nous pouvons exécuter des systèmes multitâches.

2.2.1 Exécutif cyclique

Dans cette configuration, l'exécutif cyclique est un système monolithique [2] formé essentiellement une boucle infinie. Cette boucle appelle des fonctions aussi appelées tâches, chaque tâche gère une partie du logiciel et du matériel. Elle est également connue sous le nom "polling" ou "superloop" [19].

Voici un exemple d'utilisation de l'exécutif cyclique dans des applications embarquées :

```
int main() {
   /* Initialisation du systeme (interruptions, timers...) */
   fonction_d_initialisation();

   /* La boucle infinie*/
   while(1) {
      /* Les taches de l'application */
      ...;
      if(tache_A_est_prete) {
         /* Code de la tache A */
      }
      if(tache_B_est_prete) {
         /* Code de la tache B */
      }
      ...;
   }
   /* Il ne sortira jamais de la boucle */
   /* L'execution du programme n'arrivera jamais ici */

   return 0;
}
```

Listing 2.1 – Implémentation d'une superloop

L'utilisateur exécute les routines d'initialisation avant d'entrer dans la boucle infinie, parce que l'utilisateur n'a besoin que d'initialiser le système une fois. Lorsque la boucle

2. Un système monolithique est composé d'un seul programme.

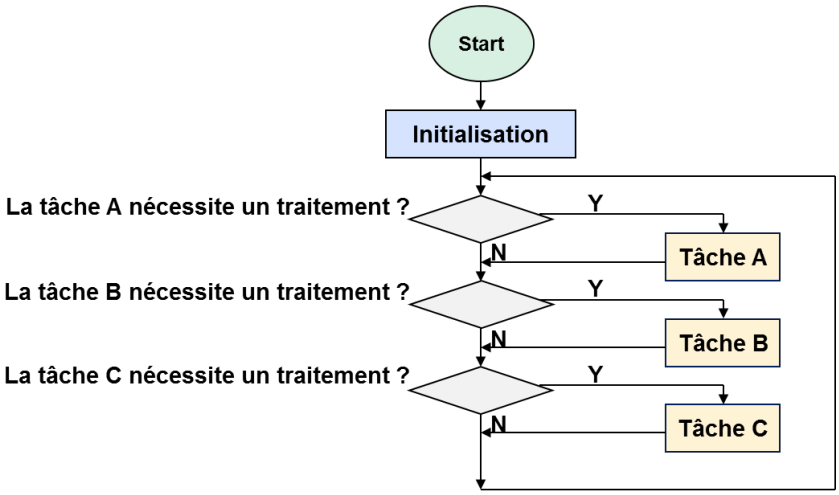

FIGURE 2.1 – Flux d'exécution d'un exécutif cyclique

infinie démarre, l'utilisateur ne réinitialise pas ses valeurs, car nous devons maintenir une vue persistante du système. La boucle est une sorte de flux de contrôle classique (traitement par lots) comme illustré sur la figure 2.1. Elle lit les entrées, effectue des calculs et met à jour ses sorties, on retrouve ce type d'implémentation dans les autopilotes sur étagère de drones [77]. De plus, les logiciels des systèmes embarqués ne sont pas les seuls à utiliser ce type d'architecture. Par exemple, les jeux vidéos utilisent fréquemment une boucle similaire.

L'exécutif cyclique est utilisé car quand on programme des systèmes embarqués, il est souvent très important de respecter les délais du système, et d'accomplir toutes les tâches clés du système dans un temps raisonnable, et dans le bon ordre. L'architecture de la "superloop" est une architecture de programme classique qui est très utile pour répondre à ces exigences. D'une part, il présente de nombreux avantages car il n'y a pas de système d'exploitation, ce qui signifie implicitement une faible empreinte mémoire et une mise en œuvre beaucoup plus simple. D'autre part, il n'est pas préemptif et ne prend en charge que des traitements relativement courts dirigés par le temps ce qui limite son champs d'action aux systèmes exécutables sur un seul processeur, et n'exécutant que des fonctions relativement courtes. La plupart des systèmes utilisant des entrées-sorties (E/S) relativement lentes par rapport au processeur, de longues fonctions d'E/S nuisent fortement à ce modèle. C'est pour cela que la plupart du temps, le modèle de programmation monolithique s'accompagne d'utilisation d'interruptions palliant à la lenteur des E/S.

2.2.2 Système contrôlé par les interruptions

Certains systèmes embarqués sont contrôlés principalement par des interruptions (voir section 3.5). Cela implique que les tâches effectuées par le système sont déclenchées par différents types d'événements ; une interruption peut être générée, par exemple, par un contrôleur de port série recevant un octet ou par un timer à une fréquence prédéfinie. Ces types de systèmes sont généralement utilisés si les gestionnaires d'événements exigent

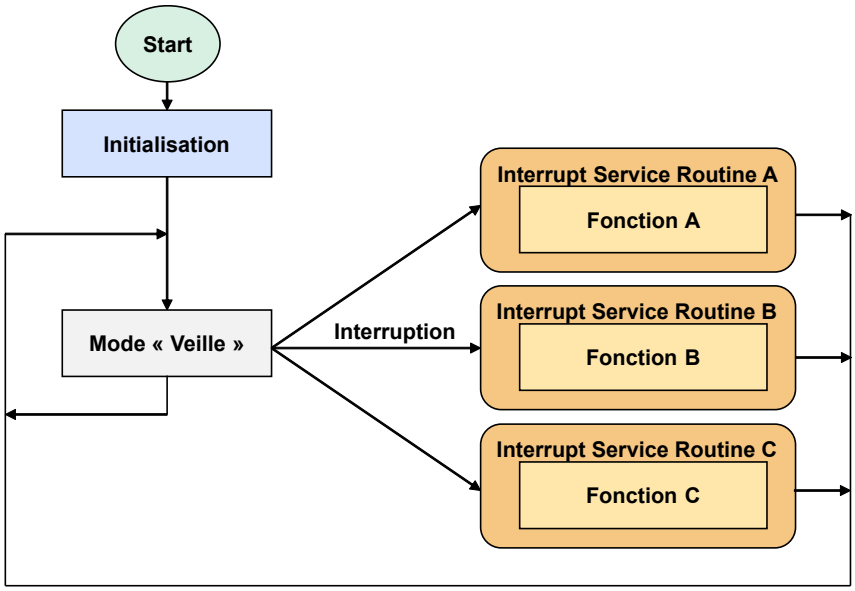

FIGURE 2.2 – Flux d'exécution d'un système contrôlé par les interruptions

une faible latence, et si leur traitement est simple et court. En général, ces types de systèmes exécutent également une tâche simple dans la boucle principale qui peut être une implémentation d'exécutif cyclique. Parfois, le gestionnaire d'interruption diffère l'exécution du code. Une fois que le gestionnaire d'interruption a terminé, ce code sera exécuté par la boucle principale. Cette technique permet au processeur de recevoir et traiter de nouvelles interruptions et diminue le risque de perdre des événements. Il est possible également, comme illustré sur la figure 2.2, de mettre le processeur en mode "veille" dans le but d'économiser l'énergie quand on sort de la routine d'interruption. Il est très fréquent de combiner superloop et utilisation d'*Interrupt Service Routine* (ISR) : les ISR vont stocker les événements reçus dans des variables globales (souvent des files), et des fonctions de la superloop vont scruter l'état de ces variables à chacune de leurs exécutions. Cependant, si l'utilisation d'ISR permet de ne pas ralentir les traitements à cause des E/S, il y a toujours l'inconvénient d'être limité à un seul cœur, mais aussi l'inconvénient d'être limité à des fonctions courtes.

2.2.3 Systèmes d'exploitation temps réel

Un système d'exploitation (*Operating system* (OS)) est une couche logicielle entre matériel et logiciel permettant d'arbitrer l'accès au matériel par plusieurs entités exécutées en parallèle et donc en concurrence. Dans la problématique de la thèse, la principale ressource arbitrée considérée est le processeur. L'OS est donc en charge d'assurer le partage du processeur en utilisant un arbitrage, s'exécutant sous la forme d'une stratégie d'ordonnancement. La plupart des stratégies d'ordonnancement existant sur les RTOS sont basées sur les priorités. Pour les systèmes temps réel, une métrique d'importance fondamentale est la latence noyau, le pire temps durant lequel l'OS peut utiliser lui-même le processeur sans pouvoir être interrompu, même par un traitement de la plus haute

priorité. Les systèmes d'exploitation généraliste, afin de maximiser les performances en moyenne, peuvent avoir de grandes latences noyau. Ce surcoût, ou overhead, peut entraîner des délais dans les traitements qui sont très pénalisant pour le respect d'échéances. Les RTOSs sont conçus de sorte à réduire cette latence noyau, afin de rapprocher le plus possible l'overhead dû au système d'exploitation d'une quantité négligeable au regard des délais imposés sur les applications hébergées. Les RTOS sont donc fortement utilisés dans de nombreux domaines tels que, par exemple, pour le contrôle des navettes spatiales, le pilotage des missiles, le contrôle des avions, etc [24]. Les systèmes à mission critique dépendent du temps. La génération actuelle de développeurs de systèmes embarqués utilise le RTOS comme système d'exploitation pour concevoir des applications embarquées [3]. Le diagramme 2.3 ci-dessous montre une abstraction RTOS.

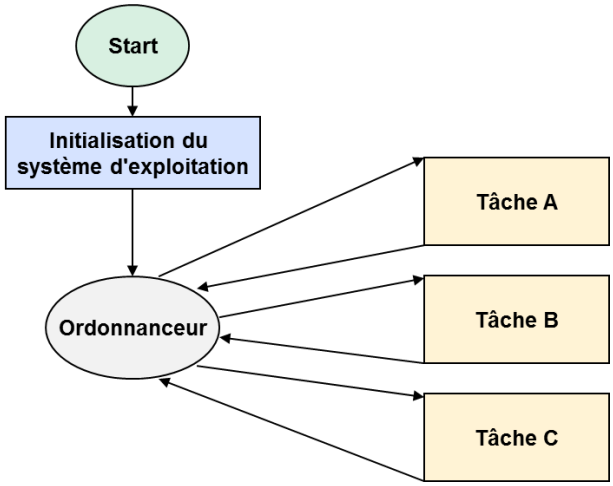

FIGURE 2.3 – Systèmes d'exploitation temps réel

2.2.3.1 Processus

Un processus est une instance d'exécution dans le système, qui possède un espace d'adressage qui lui est propre et communique avec d'autres processus via le système d'exploitation. Un processus comprend un code exécutable, un contexte de sécurité et un identifiant de processus unique. Un programme ou une application peut être divisé en différents processus lors de son exécution. Chaque processus possède son propre espace d'adressage virtuel et ne communique pas avec les autres, sauf par des mécanismes de gestion du noyau tels que la communication interprocessus *Inter-process communication* (IPC). De ce fait, si un processus plante, cela n'affectera pas les autres processus. Chaque processus est composé d'une ou plusieurs tâches.Afin d'assurer la protection mémoire, qui se fait généralement en utilisant le mécanisme de mémoire virtuelle, un processeur doit être assisté d'un circuit spécialisé dans la conversion entre adresse virtuelle et adresse physique : la *Memory Management Unit* (MMU).

2.2.3.2 Tâche

Une tâche, aussi appelée `thread` ou `fil d'exécution`, est un programme exécuté séquentiellement et contrôlé par le système d'exploitation à l'intérieur de la mémoire d'un processus. Puisque plusieurs tâches s'exécutent dans le même espace mémoire d'un processus, ces tâches peuvent partager directement des variables globales, ce qui facilite et allège les mécanismes de communication entre les tâches. Cependant, leur cohabitation dans le même espace mémoire fait que le plantage d'une tâche peut impacter tout le processus. L'information sur l'état d'une tâche est représentée par la structure de la tâche, ses données et le bloc de contrôle. La conception pour une application temps réel implique de diviser le travail en tâches responsables d'une partie du système. Chaque tâche a sa propre priorité, sa propre zone de pile et son propre bloc de registre et elle peut être vue comme une fonctionnalité ayant son propre rythme d'exécution.

2.2.3.3 États de la tâche

Chaque tâche est généralement une boucle infinie qui peut se trouver dans un des états suivants [22] :

1. Prête (ready) : Cette tâche a été créée et peut être exécutée. Cependant, la tâche prête n'est pas en cours d'exécution, car le ou les cœurs de calcul sont utilisés par une tâche de priorité plus élevée.
2. Dormante : est une tâche qui a été créée et la mémoire est allouée à sa structure mais le noyau ne peut pas la sélectionner pour s'exécuter.
3. En cours d'exécution (running) : Un CPU exécute la tâche "running" dans cet état jusqu'à ce qu'elle soit préemptée ou se bloque en attente d'un événement ou se termine.
4. En attente (ou tâche bloquée) : La tâche est en attente d'une ressource qui peut être une donnée traitée par une entrée externe du monde réel ou une autre tâche.
5. Interrompu par ISR : une interruption a eu lieu et le CPU est en train d'exécuter l'interruption, en interrompant la tâche qui était en état "running" sur le cœur.

2.2.3.4 Préemption et changement de contexte

Le changement de contexte ("context switch") est l'un des principaux apports des OS par rapport aux autres modèles de programmation. En effet, grâce à celui-ci, un traitement long peut être préempté pour laisser la place, par exemple, à un traitement plus urgent ou prioritaire. Il passe par différentes étapes pour accomplir la tâche requise. Le noyau va d'abord sauvegarder le contexte (valeurs de registre) de la tâche en exécution en fonction de sa politique d'ordonnancement, ensuite élire la tâche suivante à exécuter, puis restaurer le contexte de la tâche élue. Le changement de contexte ne peut être effectué que par l'ordonnanceur. De plus, cela se produit soit à la suite de l'appel au noyau, soit

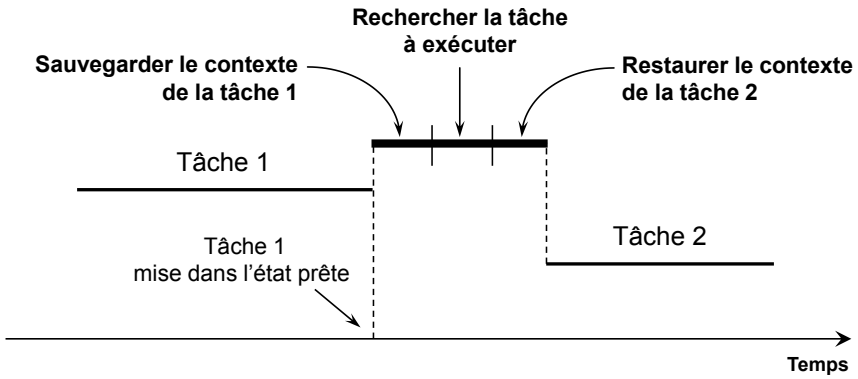

FIGURE 2.4 – Changement de contexte

explicitement par l'application en raison d'une interruption. La figure 2.4 montre les étapes d'un changement de contexte.

2.2.3.5 Types de tâches

Dans les applications temps réel, la plupart des traitements sont récurrents. Une tâche donne donc naissance, à chacune de ses activations, à un travail, aussi appelé job, que le processeur doit exécuter. On peut distinguer différents rythmes d'activation de tâches :

- Périodique

  Travail effectué de manière répétitive à intervalles réguliers. Une tâche périodique est activée suite à interruption horloge du système. On peut prévoir précisément à quels moments une tâche périodique sera activée dans le futur.

- Apériodique

  Ces tâches sont déclenchées par des événements généralement externe, qu'on ne peut prédire à l'avance. On ne peut pas non plus prédire une durée minimale séparant deux activations successives.

- Sporadique

  Comme pour les tâches apériodiques, les tâches sporadiques sont généralement déclenchées suite à interruption matérielle déclenchée par un événement externe. Cependant, contrairement aux tâches apériodiques, il existe un délai minimal entre deux activations successives.

2.3 Stratégies d'ordonnancement

2.3.1 Ordonnanceurs

L'ordonnanceur est le logiciel qui détermine la tâche à exécuter par la suite. La logique du mécanisme et de l'ordonnanceur qui détermine le moment où elle doit être exécutée est appelée l'algorithme d'ordonnancement.

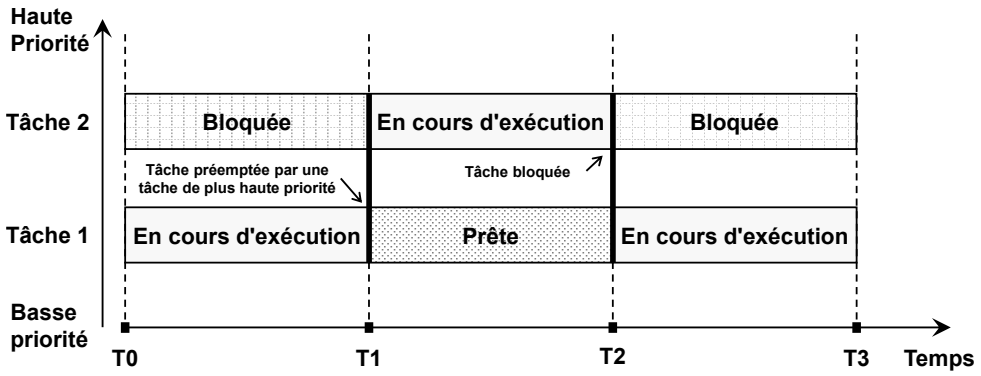

FIGURE 2.5 – Ordonnancement des deux taches en utilisant un ordonnanceur préemptif

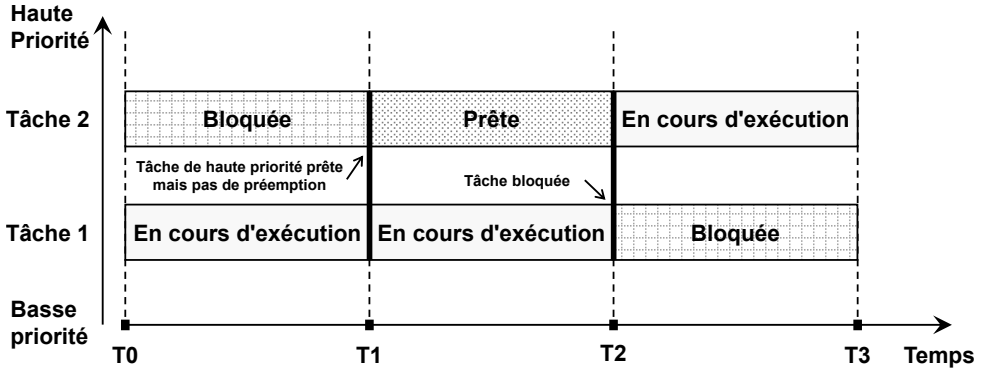

FIGURE 2.6 – Ordonnancement des deux taches en utilisant un ordonnanceur non préemptif

2.3.1.1 Ordonnancement préemptif

Un ordonnanceur préemptif est capable de préempter une tâche en cours d'exécution. Des ressources (cycles CPU) sont allouées à la tâche pour une durée déterminée, puis retirées, et la tâche est à nouveau placée dans la file d'attente de l'état prêt. Cette tâche reste dans la file d'attente "prêt" jusqu'à la prochaine possibilité de s'exécuter. Comme on peut le voir sur la figure 2.5, la tâche 1 a été préemptée dès que la tâche 2 est prête, qui va continuer à s'exécuter jusqu'à ce qu'elle soit bloquée.

2.3.1.2 Ordonnancement non préemptif

L'ordonnancement non préemptif, aussi appelé coopératif, alloue le processeur à une tâche jusqu'à ce qu'elle se bloque et passe en état d'attente. Dans ce cas, l'ordonnanceur n'interrompt pas la tâche et il attend plutôt que la tâche termine son exécution ensuite il peut allouer le CPU à une autre tâche. Sur la figure 2.6, la tâche 1, qui a la priorité la plus basse, continuera à s'exécuter même si une autre tâche devient prête.

2.3.2 Algorithmes d'ordonnancement

L'ordonnanceur a un ou plusieurs objectifs, par exemple : maximiser la capacité de traitement (quantité totale de travail achevé par le CPU) ; minimiser le temps d'attente

(temps entre le moment où la tâche est prête et celui où elle commence à être exécutée) ; minimiser le temps de réponse ; ou maximiser l'équité (temps CPU égal à chaque procédure, ou temps plus approprié en fonction de la charge de travail et de la priorité de chaque procédure). Cependant, ces objectifs sont souvent contradictoires (par exemple, latence par rapport à la capacité de traitement). La préférence est estimée en fonction des différentes exigences mentionnées non seulement ci-dessus, mais aussi en fonction des objectifs et des besoins du système. Dans les environnements temps réel, par exemple les systèmes embarqués pour le contrôle automatique dans l'industrie, l'ordonnanceur est utilisé pour s'assurer que les tâche impliquées peuvent respecter leurs échéances ; il est également important pour maintenir la stabilité du système. Les algorithmes d'ordonnancement à priorités peuvent être classés en plusieurs catégories.

- Ordonnancement à priorité fixe au niveau des tâches

  Aussi appelé ordonnancement statique, car les priorités des tâches sont attribuées une fois pour toutes. Ici, l'allocation des priorités est effectuée au moment de la conception. Ce type ordonnancement a généralement un temps de latence limité. L'ordonnancement "Rate Monotonic Scheduling" est un exemple courant d'ordonnancement à priorité fixe au niveau des tâches qui attribut une priorité de façon inversement proportionnelle à la période.

- Ordonnancement à priorité fixe au niveau des travaux

  Aussi appelé ordonnancement dynamique, dans le sens où la priorité d'une tâche peut varier, alors que celle d'un travail reste constante. L'ordonnanceur calcule les priorités au moment de l'activation d'un travail. Contrairement à l'ordonnancement à priorité fixe au niveau des tâches, l'ordonnancement dynamique a un temps de latence plus élevé, mais il permet une plus grande capacité d'utilisation du processeur. *Earliest deadline first* (EDF), qui attribut une priorité d'autant plus grande qu'un travail est urgent, est un exemple typique d'algorithme d'ordonnancement à priorité fixe au niveau des travaux [73].

- Ordonnancement à priorités dynamiques

  L'ordonnanceur recalcule régulièrement les priorités des travaux. Ce type d'ordonnancement n'est pas utilisé par les RTOS et ne sera donc pas abordé dans la thèse.

2.3.2.1 Ordonnancement Round Robin

Le Round Robin est utilisé pour ordonnancer les tâche de manière équitable, Round Robin utilise généralement le partage du temps, en donnant à chaque job un temps CPU autorisé [114], ainsi que l'interruption du job s'il n'est pas terminé. Le job est poursuivi la prochaine fois qu'un slot de temps est attribué à cette tâche. Si la tâche Round-robin a cessé d'être en attente pendant la période qui lui a été attribuée, l'ordonnanceur choisit la première tâche de la file d'attente prête à être exécuté. L'algorithme Round Robin est un algorithme préemptif car l'ordonnanceur force la tâche d'ordonnancement à s'arrêter de s'exécuter lorsque le quota de temps expire.

2.3.2.2 Rate Monotonic Scheduling (RMS)

L'algorithme `Rate Monotonic scheduling` a une règle simple permettant de donner la priorité à différentes tâches en fonction de leur période sur un monocœur. Les tâches se voient attribuer, par le concepteur, une priorité inversement proportionnelle à leur période. Cette affectation de priorités traduit le fait que plus la fenêtre d'exécution d'une tâche est petite, plus elle doit être prioritaire.

Afin de déterminer si le système est ordonnançable, il est nécessaire de réaliser une analyse d'ordonnaçabilité, comme la Rate Monotonic Analysis (RMA). La RMA est l'un des principaux représentants qui permet de déterminer le pire temps de réponse des tâches [50] [59]. Les tâches de la période la plus courte obtiennent ici la priorité la plus élevée, tandis que la priorité est inversement proportionnelle à la période. En outre, il a été prouvé que pour "n" tâches périodiques avec des périodes uniques, il existe un ordonnancement réalisable qui respectera toujours les délais si l'utilisation du CPU est inférieure à une limite spécifique (déterminée en fonction du nombre de tâches). L'équation du test est donnée ci-dessous [71] :

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1)$$

C_i est le WCET de la tâche i et T_i sa période

2.3.2.3 Earliest deadline first

L'algorithme EDF est un algorithme d'ordonnancement à priorités fixes aux travaux qui sélectionne les tâches en fonction de leur échéance absolue. Plus précisément, les tâches dont l'échéance est la plus proche seront exécutées avec une priorité plus élevée. De plus, il est généralement exécuté en mode préemptif, c'est-à-dire que la tâche en cours d'exécution est préemptée dès qu'un travail périodique avec une échéance plus proche est activé. Comme RMS, EDF est également optimal pour les systèmes mono-cœur préemptifs. Il a une limite d'utilisation du CPU de 100%. Comparé à RMS, EDF peut garantir les échéances des tâches avec un facteur d'utilisation du CPU plus élevé, mais il présente également un inconvénient, car EDF est plus difficile à implémenter parce que les priorités des tâches ne sont plus statiques mais dynamiques. Il mettre à jour la file d'attente à chaque fois qu'un travail est activé. Cela n'est pas vraiment la raison pour laquelle il est boudé par les industriels, car cette mise à jour peut s'effectuer en O(1) en utilisant un tas binaire. Cependant, en cas de non respect d'une échéance, on peut observer un effet domino (plusieurs échéances sont manquées en chaîne), de plus, n'importe quelle tâche peut être victime du non respect d'échéance. En priorités fixes aux tâches, la priorité peut être vue par certains concepteurs comme une notion d'importance entre les tâches : le fait que la tâche la moins prioritaire puisse manquer une échéance peut sembler ainsi moins "grave".

2.3.3 Ordonnancement multiprocesseurs

En ordonnancement multiprocesseur, d'un point de vue académique, il existe trois familles de stratégies :

- Stratégie partitionnée : chaque tâche se voit allouée à un cœur, il existe un ordonnanceur par cœur qui sera traité comme un cœur dans le cas monoprocesseur. Bien entendu, ce type de stratégie ne saura pas utiliser la totalité des ressources, puisque le partitionnement initial, identique à un problème de bin packing [75], a peu de chance d'aboutir à une occupation de 100% de chacun des cœurs. Ce type de stratégie est très utilisé sur des RTOS assez statiques, tels que ceux conformes à Autosar Classic [74] par exemple.
- Stratégie globale : il existe une file d'attente unique pour les tâches prêtes, qui peuvent, au gré des préemptions, se voir migrer d'un cœur à l'autre. Ce type de stratégie peut, sous certaines hypothèses négligeant les durées de préemption et migration, utiliser la totalité de la plateforme, y compris si elle est hétérogène [12]. Le problème est que les hypothèses considérées pour obtenir une pleine utilisation de la plateforme ne sont pas réalistes, car la migration ne saurait se faire en coût nul ou négligeable.
- Stratégie semi-partitionnée : afin de permettre en théorie une utilisation totale de la plateforme tout en réduisant les migrations de tâches, l'approche semi-partitionnée consiste à faire du partitionné pour la plupart des tâches, mais pour certaines, autoriser la migration.

2.4 Systèmes d'exploitation ciblés

Un système d'exploitation est une abstraction du matériel dans un système qui fournit une interface pour gérer les applications. Le système d'exploitation remplace l'interface directe avec le matériel par des fonctionnalités logicielles que l'utilisateur du système souhaite ou dont il a besoin. Il prend en charge les fonctions de base d'un système informatique et rend le système plus facile à maintenir. De plus, les applications sont plus indépendantes du matériel, et donc plus rapide et plus facile à écrire. Lors de la conception d'un système d'exploitation, divers paramètres sont pris en compte, notamment les performances, la gestion des ressources, la sécurité, les possibilités de commercialisation et la tolérance aux fautes. Il est responsable de la gestion des ressources matérielles et logicielles. Les ressources matérielles comprennent les processeurs, la mémoire et les périphériques d'entrée/sortie. Les ressources logicielles comprennent les programmes et les fichiers de données.

Un système d'exploitation est composé de couches qui créent un environnement qui cache et simplifie le matériel sous-jacent en fournissant des *Application Programming Interface* (API)s pour répondre aux besoins de l'utilisateur. Bien que la structure du noyau du système d'exploitation puisse varier, ils tentent tous de fournir à l'utilisateur une

plate-forme dans laquelle il peut utiliser le matériel du système. De nombreux systèmes d'exploitation donnent l'impression que plusieurs programmes et tâche fonctionnent en même temps grâce au multitâche. Cependant, un cœur de processeur ne peut gérer qu'un seul fil d'exécution à la fois.

Un ordonnanceur est utilisé pour gérer les tâches qui s'exécutent sur le processeur et pour créer l'illusion d'une exécution simultanée grâce à un mécanisme de partage du temps (time slicing) et à une commutation rapide entre les tâche ou processus.

Le type de système d'exploitation peut être défini par son ordonnanceur et par la manière utilisée pour choisir les tâche à exécuter. Un système d'exploitation multi-utilisateurs, comme Linux, garantit que le temps de traitement est partagé équitablement entre les utilisateurs. Un système d'exploitation de type bureautique, comme Windows, possède un ordonnanceur qui garantit que le système reste réactif aux utilisateurs lorsque cela est nécessaire. L'ordonnanceur d'un RTOS est conçu pour fournir des modèles d'exécution prévisibles aux systèmes qui ont des exigences en temps réel. Les systèmes embarqués ont souvent ces exigences, ce qui signifie que le système doit répondre à un événement donné dans un délai dont on peut prévoir une borne supérieure. Cela signifie que l'ordonnanceur du système d'exploitation doit être déterministe.

FreeRTOS est un RTOS que l'on peut trouver sur plusieurs familles de microcontrôleurs afin de répondre aux besoins temps réel de l'application, par ailleurs Linux est le système d'exploitation de choix pour les microprocesseurs, en particulier lorsqu'il est utilisé dans des applications embarquées, ceci est dû au fait qu'il est largement utilisé et très configurable afin de répondre à toutes les exigences concernant les applications embarquées. Sur les architectures AMP où nous trouvons des microprocesseurs et des microcontrôleurs sur la même puce, ces deux systèmes d'exploitation sont largement utilisés, où Linux et FreeRTOS sont installés respectivement sur les microprocesseurs et les microcontrôleurs.

2.4.1 FreeRTOS

FreeRTOS est un noyau de système d'exploitation temps réel [88] qui a été porté sur 35 plateformes de microcontrôleurs. FreeRTOS est distribué sous licence MIT. Il est conçu pour être simple et petit. Le noyau lui-même est constitué de seulement trois fichiers C. Pour que le code soit maintenable, facile à porter et lisible, il est écrit principalement en langage C, à l'exception de quelques fonctions assembleur lorsque cela est nécessaire. FreeRTOS fournit des stratégies multitâches, des tâches, des timers logiciels et des sémaphores. Un mode économie d'énergie est proposé pour les applications à faible consommation. Les priorités des tâches sont également prises en charge par FreeRTOS. L'ordonnanceur proposé est à priorités fixes aux tâches, même s'il est possible de modifier par appel explicite la priorité d'une tâche.

2.4.1.1 Ordonnanceur FreeRTOS

FreeRTOS dispose d'un ordonnanceur Round Robin (voir section 2.3.2.1), basé sur les priorités. Chaque tâche possède une priorité. Les tâches ayant la même priorité partagent le temps d'exécution de façon Round Robin. L'ordonnanceur FreeRTOS peut être configuré comme préemptif (voir section 2.3.1.1) ou coopératif (voir section 2.3.1.2). Le comportement d'applications complexes nécessite un ordonnancement préemptif. Pour des systèmes plus simples, l'ordonnancement coopératif peut être utilisé. L'ordonnanceur préemptif peut arrêter une tâche en cours d'exécution (préempter la tâche) pour donner des ressources CPU à une autre tâche prête. Cette fonction est utilisée pour le partage du temps du processeur entre les tâches prêtes ayant la même priorité. Elle est également utilisée dans le cas d'une interruption qui peut réveiller une tâche en attente d'un signal ou de données. La tâche réveillée doit avoir une priorité plus élevée que la tâche en cours d'exécution pour pouvoir utiliser directement le processeur.

Lorsque l'ordonnanceur coopératif est utilisé, un changement de contexte ne se produira que si l'une des conditions suivantes est satisfaite :

- Une tâche appelle explicitement taskYIELD().
- Une tâche appelle explicitement une fonction API qui aboutit au passage à l'état bloqué.
- Une interruption qui effectue explicitement un changement de contexte.

Les systèmes sont moins réactifs lorsque l'ordonnanceur coopératif est utilisé. Quand l'ordonnanceur préemptif est utilisé, il commence à exécuter une tâche avant un temps donné, appelé latence noyau, après que celle-ci soit devenue la tâche la plus prioritaire de l'état Prêt. Cela est souvent essentiel dans les systèmes temps réel qui doivent répondre à des événements de haute priorité dans une période de temps définie. En revanche, lorsque l'ordonnanceur coopératif est utilisé, le passage à une tâche qui est devenue la tâche la plus prioritaire de l'état Prêt n'est pas effectué avant que la tâche en cours d'exécution passe à l'état bloqué ou appelle taskYIELD().

2.4.1.2 Communication inter-tâches

FreeRTOS fournit plusieurs méthodes pour la communication inter-tâches, y compris les files d'attente de messages (Queues) et les sémaphores binaires. Le mécanisme de file d'attente de FreeRTOS peut être utilisé dans les communications entre deux tâches et la communication entre les tâches et la routine de service d'interruption. Une file d'attente est une structure capable de stocker et de récupérer des données. Les sémaphores dans FreeRTOS sont en fait implémentés comme un cas spécial de files d'attente. Un sémaphore est une file d'attente d'un seul élément de taille zéro. L'opération de prise d'un sémaphore est équivalente à la réception d'une file d'attente, alors que l'opération de libération (ou de remise) d'un sémaphore est équivalente à un envoi dans une file d'attente. Notez qu'à l'initialisation, la file d'attente du sémaphore est pleine. Les sémaphores sont utilisés pour la synchronisation des tâches et l'exclusion mutuelle. Une section de l'application peut être

protégée par un sémaphore pour permettre l'exclusion mutuelle. La première tâche qui exécute une section du code mutuel prend le sémaphore. Toute autre tâche qui souhaite exécuter ce code attendra sur le sémaphore jusqu'à ce que la première tâche le libère. Pour l'exclusion mutuelle il faut utiliser les mutexes, un type spécial du sémaphore binaire, car ils fournissent le mécanisme d'héritage de priorité. Une tâche qui souhaite recevoir un octet d'une file d'attente vide, envoyer un octet à une file d'attente pleine ou prendre un sémaphore déjà utilisé sera bloquée par le noyau. Une tâche choisit le temps maximum qu'elle autorise le noyau à la bloquer dans le paramètre d'appel système de communication inter-tâche. Lorsque le sémaphore ou la file d'attente redevient disponible, le noyau prépare la tâche. Elle sera autorisée à s'exécuter si elle a la plus haute priorité. Si le sémaphore ou la file d'attente reste occupé(e) et que le temps d'attente est écoulé, le noyau prépare à nouveau la tâche et lui renvoie une erreur. Il est de la responsabilité de la tâche de vérifier cette valeur de retour.

2.4.2 Aperçu du noyau Linux

Le noyau ou kernel est le composant clé du système d'exploitation. La principale responsabilité de l'ordonnanceur est de gérer les ressources du système. La gestion des ressources comprend plusieurs contextes. Premièrement, le noyau fournit une interface permettant aux applications d'avoir accès aux ressources matérielles. Lorsque les applications envoient des demandes de ressources matérielles telles que l'espace d'adressage, le noyau reçoit des requêtes et utilise des appels système pour communiquer avec ces applications. Deuxièmement, le noyau alloue des ressources telles que le processeur et la mémoire. Enfin, le logiciel du noyau est généralement organisé en sous-systèmes. Les sous-systèmes correspondent logiquement aux ressources avec lesquelles le noyau traite. Ces systèmes comprennent les processus, la gestion de la mémoire, les systèmes de fichiers, le contrôle des périphériques et la mise en réseau. Le noyau crée et détruit les processus, et ordonnance l'exécution de ces processus à l'aide de son ordonnanceur. La figure 2.7 illustre le noyau Linux et ses sous-systèmes.

En résumé, les tâches du noyau comprennent la gestion des ressources, le traitement des demandes, le suivi des processus et l'allocation des ressources. L'allocation et le traitement des demandes font partie de la gestion des ressources. Les noyaux effectuent également une gestion interne qui n'est pas directement liée aux services. Le noyau doit suivre les ressources qu'il utilise et collecte souvent des informations sur divers aspects du système.

L'ordonnanceur joue un rôle clé au sein du noyau. Le système d'exploitation Linux est maintenant utilisé pour de nombreuses applications différentes, telles que les serveurs, les ordinateurs de bureau et les systèmes embarqués. L'ordonnanceur a été développé et modifié en même temps que le noyau. Le premier ordonnanceur Linux était très simple. Il utilisait une file d'attente circulaire de tâches qui fonctionnait avec une politique d'ordonnancement round-robin. Les ordonnanceurs O(1) et *Completely Fair Scheduler* (CFS) ont ensuite été introduits.

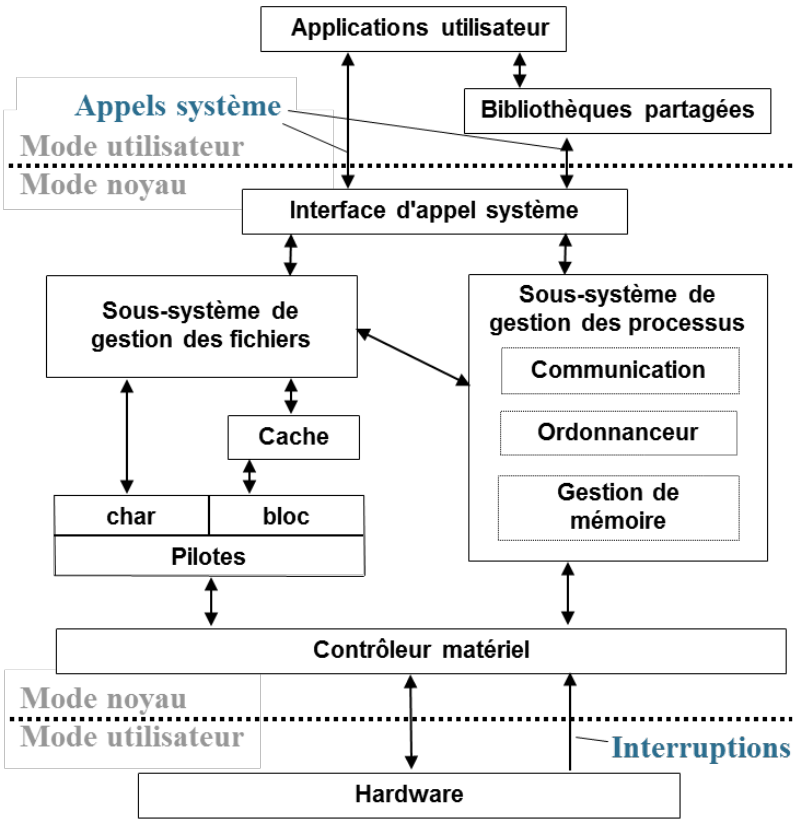

FIGURE 2.7 – Noyau Linux

2.4.2.1 Ordonnanceur O(1)

La première version du noyau Linux a été publiée par Linus Torvalds en 1991. Les versions initiales ne s'attachaient pas au fait d'avoir un ordonnanceur très efficace. Jusqu'à la version 2.6 ou plus précisément Linux 2.6.8.1, publié en 2004, ce composant était très simple et s'adaptait assez mal à l'augmentation du nombre de processus ou du nombre de processeurs disponibles.

Ingo Molnar a été le premier développeur à apporter une contribution cruciale à l'ordonnanceur Linux (intégrée à la version 2.6.8.1 de Linux), à la fois utilisé dans les serveurs et les ordinateurs de bureau, deux ordinateurs avec des objectifs d'ordonnancement différents, cet ordonnanceur de processus prend en compte la variété de cibles Linux. Ce dernier avait même la capacité de choisir la prochaine tâche à exécuter en un temps constant et non dépendant du nombre de tâches à ordonnancer ; un algorithme avec cette caractéristique est noté O(1). Ainsi, cette implémentation de l'ordonnanceur Linux est couramment appelée ordonnanceur O(1)[120]. Elle permet de dépasser les problèmes du précédent ordonnanceur et propose de nouvelles fonctionnalités et caractéristiques de performance.

La priorité attribuée à une tâche dans cet ordonnanceur est dépendante de la valeur "nice" qui est la valeur statique (selon la norme POSIX) et d'une composante dynamique calculée pour améliorer l'interactivité. Pour classer les tâches comme étant liées au CPU ou aux E/S, l'ordonnanceur O(1) utilise une heuristique assez complexe et donne une

priorité plus élevée aux tâches qui nécessitent une meilleure interactivité, classées en E/S. De plus, les mécanismes d'équilibrage de la charge de travail sont pris en charge par cet ordonnanceur afin de les répartir sur les processeurs disponibles. Pour ordonnancer les tâches en temps réel, le support temps réel souple très simple utilise les files d'attente *First In First Out* (FIFO) ou round robin avec une priorité maximale.

Avec le temps, alors que l'ordonnanceur O(1) était satisfaisant pour les grandes charges de travail des serveurs ne disposant pas de processus interactifs, ses performances étaient inférieures à la normale sur les ordinateurs de bureau. Il est devenu évident que l'ordonnanceur O(1) présentait plusieurs défaillances pathologiques liées à l'ordonnancement des applications sensibles à la latence.

2.4.2.2 Completely Fair Scheduler (CFS)

Introduit dans la version 2.6.23 du noyau, publiée en octobre 2007, l'algorithme d'ordonnancement actuellement utilisé dans Linux s'appelle CFS [85]. Il a été conçu par le même développeur pour résoudre les limitations de l'ordonnanceur O(1) pour les priorités non temps réel.

Comme son nom l'indique, l'objectif principal de CFS est d'être équitable dans l'affectation des ressources informatiques aux tâches en cours d'exécution. Cela a été repris de l'ordonnanceur Rotating Staircase Deadline Scheduler (RSDL), qui implique que sans essayer de caractériser le comportement des tâches, il faut être équitable dans l'attribution des CPU. Ceci est obtenu en écartant le concept de tranche de temps et en introduisant l'idée de temps d'exécution virtuel d'une tâche. Ce dernier, aussi appelé vruntime d'une tâche qui représente la durée pendant laquelle une tâche a occupé le processeur. En résumé, l'ordonnanceur choisit simplement la tâche avec le vruntime le plus bas et la sélectionne pour être exécutée. Par conséquent, sous Linux, la quantité de temps processeur que reçoit une tâche est en fonction de la charge du système. La valeur nice agit comme un poids affectant cette proportion du temps processeur que chaque tâche reçoit.

Ainsi, une tâche ayant le temps d'exécution virtuel le plus bas de sa file d'attente, est laissée en exécution étant donné que le CFS n'a pas de concept de tranche de temps. Il y a une file d'attente d'exécution pour chaque processeur disponible (comme dans l'ordonnanceur O(1)). La seule différence réside dans la mise en oeuvre de cette file d'attente.

Les files d'attente propres à chaque processeur sont implémentées dans le CFS, sous forme d'arbres rouge-noir (voir figure 2.8), qui sont une classe particulière d'arbres binaires équilibrés. Cette structure de données permet l'insertion et la suppression avec une complexité de O(log(n)), où n est le nombre de nœuds (c'est-à-dire le nombre de tâches dans la file d'attente) et elle est topologiquement ordonnée de sorte que le nœud avec l'indice minimum (c'est-à-dire la tâche avec le temps d'exécution minimum) sera toujours celui le plus à gauche de l'arbre.

De cette façon, le CFS choisit simplement comme prochaine tâche à exécuter le noeud ou la feuille le plus à gauche de l'arbre rouge-noir (ce qui peut être fait en temps constant).

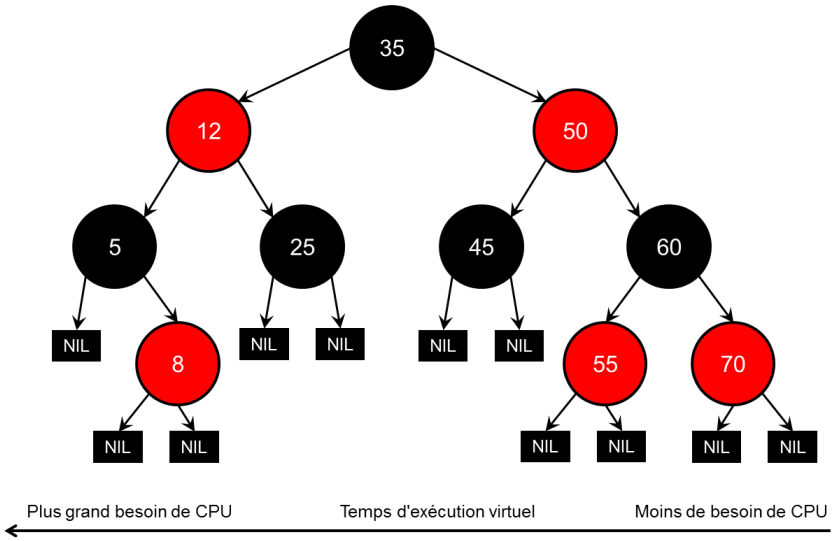

FIGURE 2.8 – Arbre rouge-noir

Les tâches exécutées auront un temps d'exécution plus élevé et seront donc placées dans l'arbre de plus en plus à droite, donnant ainsi à toutes les tâches la chance d'être exécutées sur le CPU dans un délai déterministe. Grâce à son approche équitable Le CFS modélise l'ordonnancement des tâches comme si le système disposait d'un processeur multitâche idéal en ce qui concerne à la fois l'interactivité et le flux de données et obtient ainsi de bonnes performances d'ordonnancement. Ce fonctionnement rend les ordonnanceurs Linux classiques peu compatibles avec le temps réel, car il est complexe de calculer un pire temps de réponse par rapport à un algorithme à priorités fixes aux tâches par exemple.

2.4.2.3 Linux temps réel

L'intérêt pour l'utilisation de Linux en tant que système d'exploitation temps réel s'est accru avec la montée en popularité de ce noyau libre et open-source. Ce dernier n'était pas initialement destiné à fonctionner en temps réel, mais il y a eu de nombreuses tentatives pour faire que ce noyau soit adapté à cette fin. Ainsi, Linux temps réel permet de favoriser et généraliser l'adoption du temps réel. Or, avant de faire de ces tentatives une réalité, différentes approches ont été adoptées[106].

Grâce à une API temps réel, les deux extensions RTAI et Xenomai apportent des performances temps réel au noyau Linux. Un nano-noyau est exécuté en dessous du noyau Linux, se positionnant entre le matériel et le noyau ou l'approche à double noyau (voir figure 2.9). Le nano-noyau exécute également l'extension en question. Avant d'être propagées plus loin, les interruptions matérielles sont alors interceptées par le nano-noyau : l'extension temps réel a une chance de réagir aux interruptions avant d'impliquer le noyau Linux parce qu'elle est positionnée avant le noyau Linux dans le pipeline de réception de ces interruptions. Cela permet des temps de réponse déterministes pour les tâches temps réel. Les tâches non temps réel sont simplement propagées vers le noyau Linux où elles peuvent être programmées selon l'ordonnanceur en vigueur [5]. Depuis la version 3, Xe-

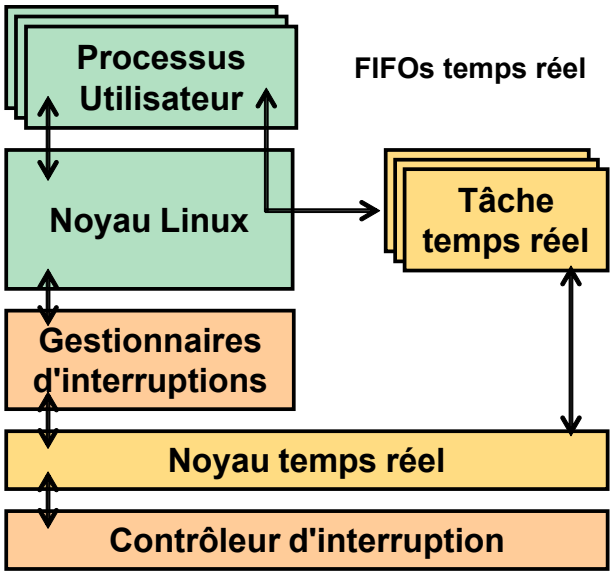

FIGURE 2.9 – Organisation du flux de données et du contrôle dans un système Linux à double noyau

nomai est également disponible dans une configuration à noyau unique, où il s'appuie sur les capacités temps réel du noyau Linux. Cette configuration est généralement utilisée en conjonction avec RT-Preempt [95].

Ingo Molnar a créé un patch visant à rendre le noyau Linux adapté au fonctionnement temps réel dur. Nommé RT-Preempt, ce patch met en œuvre l'héritage de priorité pour les constructions de synchronisation dans le noyau, empêchant ainsi l'inversion de priorité dans le noyau. A travers quelques modifications du code source, le patch fonctionne en rendant le noyau entièrement préemptible réduisant ainsi la latence noyau. En comparant le patch RT-Preempt avec les extensions mentionnées ci-dessus, son avantage majeur est qu'il est compatible avec le noyau Linux standard : cela implique qu'aucune recompilation n'est nécessaire pour faire fonctionner différentes applications utilisateur ; les avantages du temps réel s'appliquent aux API Linux standard. Inversement, on peut aussi exécuter l'application temps réel développée pour fonctionner sur le noyau temps réel sans aucune modification ou recompilation, sur le noyau Linux standard.

Conçu initialement pour coordonner les efforts de mise en place de la RT-Preempt et aider les mainteneurs dans leur effort de développement continu et de support à long terme, il a été constaté que l'enthousiasme pour l'utilisation du projet Real-Time Linux (RTL) ne se limitait pas aux anciens systèmes temps réel tels que l'aérospatiale et l'automobile. En effet, Linux temps réel est utilisé dans de nombreux domaines, par exemple dans la robotique ou le réseau. Il est également utilisé dans des domaines non embarqués tels que les applications multimédia, le calcul haute performance et les frameworks de simulation. Il a également été envisagé par la National Aeronautics and Space Administration (NASA) et l'Agence spatiale européenne (ESA) pour gérer diverses applications. [105]

Le noyau Linux standard est, quant à lui, préemptif mais pas temps réel. Il permet cependant d'exploiter les priorités temps réel telles quelles, ce qui donne à l'utilisateur

un meilleur contrôle sur la réactivité d'un système au niveau de chaque tâche. Dans la plupart des environnements multitâches, le noyau préemptif permet à la tâche qui a une priorité plus élevée que les autres d'obtenir un temps plus long sur le processeur. Cependant, dans un noyau standard, aucune tâche spécifique n'est autorisée à occuper en permanence les ressources du processeur, peu importe sa priorité. C'est là où RT-Preempt joue un rôle important. Par exemple, lorsque des projets embarqués ont besoin d'un système d'exploitation temps réel, ils peuvent utiliser RT-Preempt. Dans ce cas, RT-Preempt transforme un Linux standard en système temps réel.

L'absence d'un second noyau dédié à la gestion des applications temps réel représente la différence la plus évidente entre les approches de Linux temps réel et RT-Preempt. Cela rend l'implémentation des processus temps réel dans l'espace utilisateur similaire à celle des processus non temps réel. En pratique, quelques autres aspects doivent être réellement pris en compte dans le développement d'applications Linux temps réel. Les tâches qui s'exécutent simultanément peuvent en fait interférer du point de vue de la synchronisation [94]. Pire encore, le swapping de mémoire peut conduire à des défauts de page, impactant lourdement la latence et la prédictibilité. Ce problème peut être évité en exploitant le mécanisme de protection classique consistant à verrouiller les pages mémoire allouées à un processus temps réel, un autre risque est lié à l'utilisation de pilotes de périphériques, l'ordonnancement des entrées/sorties n'étant généralement pas compatible avec le temps réel.

D'une manière générale, le développement logiciel de RT-Preempt est complètement différent de celui des doubles noyaux : une application peut être exécutée en mode temps réel sans être réécrite. Le modèle de programmation des approches à double noyau utilise des appels système spécialisés. Même si l'on considère les avantages des modèles de programmation spécialisés, ils suppriment l'un des avantages les plus importants de l'utilisation de Linux dans les environnements temps réel : l'exploitation de l'énorme ensemble de pilotes et de bibliothèques déjà disponibles pour accélérer le processus de développement des applications. Il convient de mentionner que même si ces pilotes et bibliothèques sont disponibles dans Linux, des efforts supplémentaires peuvent être nécessaires pour les rendre conformes au temps réel. Dans les approches à double noyau, les tâches temps réel peuvent toujours accéder aux fonctionnalités de Linux en utilisant des appels IPC dédiés aux tâches temps réel logicielles de Linux. Cette partie de l'exécution ne doit pas être critique en termes de temps ; sinon, des latences non bornées peuvent se produire en raison de la présence d'une tâche temps réel non dure s'exécutant au-dessus du noyau Linux.

Toutes les approches à double noyau nécessitent par nature des modifications invasives du code du noyau, et leurs interactions ne sont pas faciles à analyser d'un point de vue fonctionnel et temporel. Cette exigence introduit également une dépendance stricte par rapport à la version du noyau, ce qui peut représenter une limitation en termes de maintenabilité et de portabilité. Ce problème ne concerne pas RT-Preempt, qui a l'avantage d'être développé directement par la communauté Linux. Au contraire, les approches à double noyau sont généralement basées sur d'anciennes versions du noyau Linux. De plus, ces approches sont généralement moins stables et moins sûres en raison de l'interaction

complexe entre les deux noyaux, ce qui demande un effort supplémentaire aux développeurs temps réel [16]. Dans l'ensemble, RT-Preempt permet aux développeurs de travailler dans un environnement Linux réel dans lequel ils peuvent facilement réutiliser la plupart des bibliothèques et outils existants, y compris tous les ensembles de fonctions spécifiés par la norme POSIX [95].

2.4.2.4 POSIX

La norme *Portable Operating System Interface* (POSIX) est, à proprement parler, toute une collection de normes IEEE qui décrivent, à la base, l'interface entre l'application et le système d'exploitation. Fondamentalement, POSIX est utilisé dans les systèmes embarqués les plus complexes équipés de processeurs puissants. Il supporte le langage de programmation C ainsi que le langage de programmation Ada, ce dernier étant principalement utilisé dans les systèmes de sécurité de l'aviation, des chemins de fer, de l'armée et de l'énergie nucléaire.

Le standard spécifie un ensemble d'appels système pour faciliter la programmation concurrente. Les services comprennent l'exclusion mutuelle avec héritage de priorité et priorité plafonnée, la synchronisation d'attente et de signal via des variables de condition offrant la puissance des moniteurs de Hoare, des objets de mémoire partagée pour le partage des données et des files d'attente de messages priorisés pour la communication inter-tâches. Il spécifie également les services permettant d'obtenir un comportement temporel prévisible, tels que l'ordonnancement préemptif à priorité fixe, l'ordonnancement sporadique, la gestion du temps à haute résolution, les opérations de mise en veille, les compteurs multi-usages et la gestion de la mémoire virtuelle.

Etant donnée la richesse de POSIX et son besoin de ressources (multi-utilisateur, multi-processus, système de fichiers complexes, etc.), POSIX propose différents "profils" [113]. La figure 2.10 illustre cela. La configuration minimale, appelée PSE51, définit la mise en œuvre de l'interface la plus simple entre l'application et le système d'exploitation. Lorsque le niveau de fonctionnalité augmente (PSE52, PSE53, et PSE54), la portée de l'interface augmente également. Chaque niveau d'extension contient à quelques exceptions près les fonctionnalités des niveaux d'extension précédents.

La liste suivante présente les caractéristiques de chaque version :

PSE51 : Profil minimal du système temps réel

- Le système n'a qu'un seul processeur mais peut avoir plusieurs cœurs.
- L'application consiste en un seul processus avec un ou plusieurs threads.
- Le système d'exploitation fournit une interface de communication basée sur les messages pour échanger des données avec les systèmes d'exploitation POSIX sur d'autres processeurs.
- Aucune gestion de la mémoire n'est mise en œuvre (unité de gestion de la mémoire, MMU).

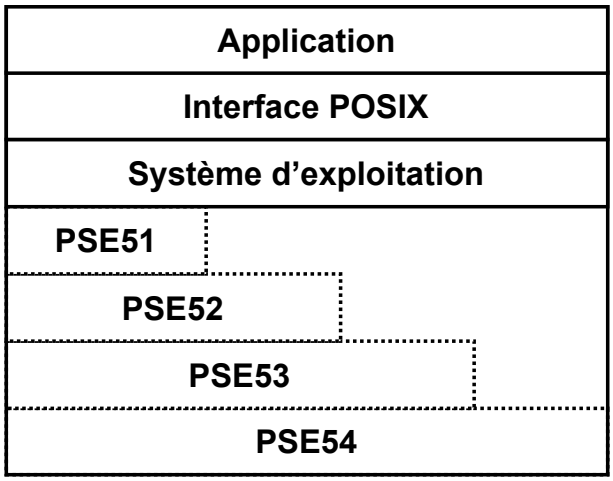

FIGURE 2.10 – Hiérarchie POSIX

- Il n'y a pas de services de gestion de périphériques d'entrée et de sortie.

PSE52 : Profil du système de contrôle temps réel

- La gestion de la mémoire n'est pas nécessaire, mais peut être implémentée.
- Les périphériques d'entrée et de sortie sont pris en charge, mais les interfaces doivent être non bloquantes. Cela signifie qu'un service appelé ne doit pas attendre en interne des événements et ainsi retarder indûment la suite de l'exécution du programme.

PSE53 : Profil du système temps réel dédié

- Le système utilise un ou plusieurs processeurs, et chaque processeur a sa propre gestion de la mémoire.
- Plusieurs processus sont supportés. Chaque processus possède un ou plusieurs threads.
- Les processus doivent être isolés les uns des autres pour limiter les interférences.

PSE54 : Système multi-usages temps réel

- Couvre la totalité des services d'un système Unix généraliste.

2.5 Conclusion

Ce chapitre se concentre sur les éléments principaux liés aux architectures opérationnelles des systèmes embarqués notamment les systèmes temps réel, ainsi que les stratégies d'ordonnancement et les systèmes d'exploitation. Diverses architectures opérationnelles pour les systèmes embarqués ont été abordées. Nous avons présenté le type d'architecture logicielle qui pouvait être utilisé lors du développement d'un système embarqué en fonction du matériel, des performances des systèmes embarqués et de diverses autres exigences. Les programmes relativement simples peuvent être gérés à l'aide d'une architecture simple basée sur l'exécutif cyclique c'est à dire ce que l'on appelle aussi une superloop, généralement couplée à l'utilisation d'interruptions matérielles afin de limiter l'impact pénalisant des E/S. Cependant, si le programme n'est pas si petit, en particulier si certains traitements s'avèrent longs et que la préemption devrait être utilisée, alors il faut installer une

architecture logicielle plus avancée utilisant les services d'un système d'exploitation. Ces architectures opérationnelles de systèmes embarqués permettent de gérer des programmes complexes sur différentes plateformes.

Le domaine du système embarqué s'est développé comme un grand domaine de recherche au cours des dernières années, plusieurs systèmes d'exploitation sont disponibles mais Linux et FreeRTOS sont les plus utilisé sur des cartes MPSoCs hétérogènes du marché grâce à deux raison principale, la première est la gratuité et la seconde car ils sont très configurables et compatibles avec plusieurs architectures et cibles.

Linux, étant un système d'exploitation à usage général, nous pouvons l'installer sur des systèmes embarqués nécessitant du temps réel mou ou sans des besoins temps réel. En revanche, il est possible de le rendre plus adapté à des systèmes avec plus d'exigence temps réel grâce au patch RT-Preempt ou à l'implémentation à double noyau. Cela permet de créer un système plus déterministe avec une latence noyau plus faible. FreeRTOS, étant un système d'exploitation temps réel, il est utilisé dans des systèmes avec des exigences temps réel dur. Les deux systèmes d'exploitation peuvent cohabiter, mais séparément, sur les puces asymétriques hétérogènes. Par exemple, dans un drone, FreeRTOS peut être utilisé pour héberger les taches du système de contrôle de vol, en outre, Linux et ses riches bibliothèques pourra être utilisé pour gérer le flux vidéo de la caméra et effectuer du traitement d'image si nécessaire.

Chapitre 3

Architecture des processeurs hétérogènes

Sommaire

3.1 Introduction 53
3.2 Pipeline 54
3.3 Mémoire virtuelle 55
3.4 Mémoire cache 55
3.4.1 Cohérence de cache 56
3.4.2 Le protocole Snooping 56
3.5 Interruptions 57
3.6 Bus et périphériques sur une puce 58
3.6.1 Bus 58
3.6.2 Périphériques 58
3.7 Architecture ARM 60
3.7.1 Comparaison entre le Cortex-M et le Cortex-A 61
3.7.2 Système sur une puce 63
3.7.3 Système sur une puce à architecture hétérogène 63
3.8 ARM big.LITTLE 64
3.8.1 Gestion de l'énergie sur big.LITTLE 66
3.8.2 DynamIQ 67
3.9 Système asymétrique hétérogène 68
3.9.1 Exemples de puces asymétriques hétérogènes 68
3.10 Communication inter-processeurs 70
3.10.1 Contrôleur de communication inter processeur 70
3.10.2 Les enjeux de la communication inter processeurs 71
3.10.3 Sémaphore matériel 72
3.11 Développement sur des processeurs ARM 73
3.12 Débogage 74
3.12.1 Débogueur ARM et JTAG 75
3.12.2 Débogueur GNU 76
3.13 Firmware 77

3.13.1 CMSIS 78
3.13.2 Couche d'abstraction matérielle 79
3.13.3 Protocole de transmission de messages asymétriques 79
3.14 Conclusion 83

3.1 Introduction

Depuis les années 1960, les processeurs ont beaucoup évolué, leurs performances ont augmenté de manière exponentielle selon ce que l'on appelle la loi de Moore [81]. Il y a toujours eu des prédictions selon lesquelles la loi de Moore prendrait fin. Néanmoins, la micro-électronique a réussi à maintenir un taux d'amélioration exponentiel grâce au développement de nouvelles technologies et architectures, telles que les architectures multicœurs, les unités de traitement graphique *Graphics Processing Unit* (GPU) et les *Field Programmable Gate Arrays* (FPGA). Aujourd'hui, les problèmes de traitement informatique sont de plus en plus nombreux. En outre, les processeurs ont vu, pendant ces dernières années, une résurgence des nouvelles approches comme les systèmes hétérogènes. Les architectes de processeurs ont largement bénéficié des progrès constants des techniques de fabrication des puces, qui ont considérablement augmenté le nombre de transistors disponibles à utiliser dans leurs conceptions. Auparavant, les architectes utilisaient des transistors supplémentaires en concevant des CPU à un seul cœur plus puissant et plus complexe, jusqu'à ce que cette tendance soit moins intéressante en raison des contraintes de puissance et des limites de l'exécution parallèle. Les tendances actuelles exploitent les densités de transistors supplémentaires en implémentant plusieurs cœurs de calcul identiques sur une seule puce, améliorant ainsi les performances lors de l'exécution simultanée de plusieurs applications ou de charges de travail parallélisées. Ces types de cœurs homogènes sont connus sous le nom de SMP. Tant qu'un seul type de cœur est utilisé, soit des cœurs complexes et puissants, soit des cœurs simples et peu puissants, il y a un inconvénient pour le SMP, c'est que tous les cœurs devront être implémentés avec la même complexité que le cœur le plus puissant. C'est une limitation car toutes les applications et tous les environnements n'ont pas les mêmes contraintes. Cet inconvénient est apparent dans les conceptions SMP qui sont de plus en plus limitées par les problèmes de dissipation d'énergie ainsi que par les exigences spécifiques aux applications. La nécessité de développer des processeurs plus rapides, plus petits et moins puissants a motivé la recherche liée aux processeurs hétérogènes ou asymétriques AMP.

ARM conçoit les processeurs les plus populaires, particulièrement utilisés dans les appareils portables et les systèmes embarqués grâce à leurs faible consommation d'énergie, leurs hautes performances et des prix raisonnables. L'architecture ARM est très modulaire et compatible avec des composants tiers, permettant ainsi d'intégrer de nombreux éléments sur une même puce. ARM est largement utilisé dans les SoC hétérogènes, il est possible de créer toutes les différentes formes de systèmes hétérogènes, comme big.LITTLE, microcontrôleur avec microprocesseurs ou puces avec FPGA. Ce chapitre aborde tous les éléments principaux que l'on peut trouver dans un SoC ainsi que dans un système multiprocesseur hétérogène sur une puce HMPSoC, comme les différents types de processeurs, plus particulièrement la famille ARM, ainsi que les bus et tous les éléments qui peuvent interférer sur le déterminisme d'un système comme les caches, les pipelines et les interruptions.

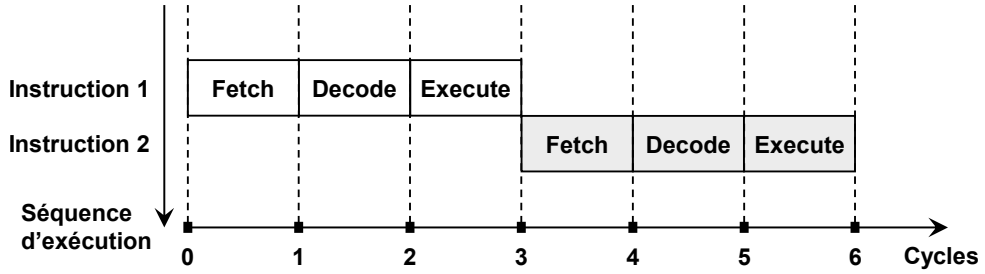

FIGURE 3.1 – Séquence d'exécution pour une architecture non pipelinée

3.2 Pipeline

Les trois étages de base du traitement d'une instruction sont le chargement (Fetch), le décodage (Decode) et l'exécution (Execute). Les performances d'un microprocesseur qui suit ces trois étages de manière séquentielle sont illustrées sur la figure 3.1, nous pouvons remarquer que six cycles d'horloge sont nécessaires pour achever l'exécution de deux instructions. Cela est dû au fait qu'une instruction doit attendre et rester inactive.

Il est possible de faire les trois étages en même temps, mais pas sur la même instruction. En effet, le temps d'inactivité aurait pu être utilisé pour précharger la deuxième instruction, car le décodage de la première instruction et le chargement de la deuxième instruction peuvent être effectués simultanément sans s'affecter mutuellement [91]. Il en résulterait donc une amélioration du nombre d'instructions exécutées par le microprocesseur. Un tel microprocesseur est dit basé sur l'architecture pipelinée et est représenté sur la figure 3.2. Nous voyons qu'un microprocesseur à architecture pipelinée décode la première instruction et précharge la suivante. De la même façon, lorsque la première instruction est exécutée, la deuxième instruction est décodée, tandis que la troisième instruction est chargée et que ces trois opérations sont effectuées simultanément. L'architecture non pipelinée prend six cycles d'horloge pour compléter deux instructions alors que le microprocesseur basé sur une architecture pipelinée prend quatre cycles d'horloge pour compléter l'exécution de ces deux instructions. Le processeur Cortex-M4 possède un pipeline à trois étages : Fetch, Decode et Execute. Toutefois, les microprocesseurs ayant une architecture plus complexe divisent les trois phases de traitement en plusieurs sous-phases, ce qui fait que le pipeline comporte un plus grand nombre d'étages et nécessite donc un plus grand nombre de phases par exécution d'instruction. Par exemple, le Cortex-A53 d'ARM a un pipeline de 8 étages [41].

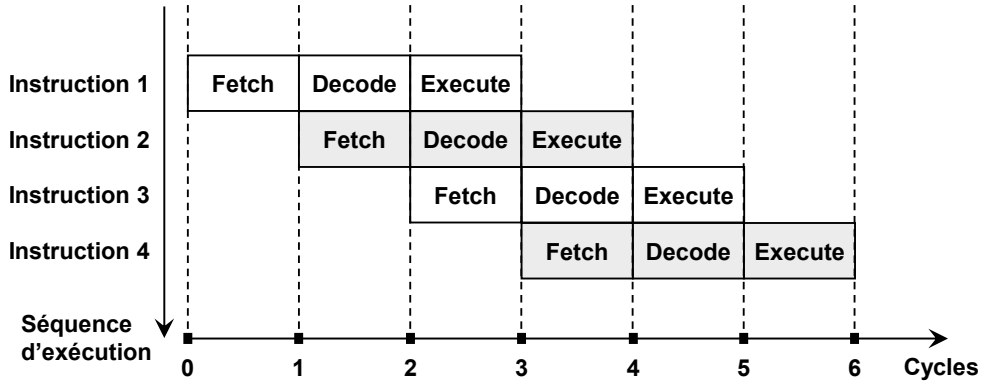

FIGURE 3.2 – Séquence d'exécution pour une architecture pipelinée

3.3 Mémoire virtuelle

La mémoire virtuelle est une technique de gestion de la mémoire qui permet au système d'exploitation de créer l'illusion de fournir plus de mémoire que celle disponible en tant que mémoire physique en utilisant le stockage secondaire. Elle fait correspondre les adresses virtuelles, qui sont utilisées par les programmes, aux adresses physiques de la mémoire principale. Tous les accès à la mémoire doivent d'abord être traduits avant de pouvoir accéder aux données. Le système d'exploitation déplace les données entre la mémoire secondaire et la mémoire principale en fonction des modèles d'accès. L'unité de gestion de la mémoire MMU est chargée d'effectuer les traductions d'adresses virtuelles en adresses physiques. Elle assure également la protection de la mémoire en empêchant les processus d'accéder à une mémoire à laquelle ils ne sont pas autorisés à accéder. Cela empêche un processus malveillant ou erroné d'affecter le noyau ou d'autres processus s'exécutant sur le même système. La traduction effectuée par la MMU est relativement lente car elle doit parcourir le répertoire de pages du processus, qui est une structure de données contenant toutes les associations entre pages virtuelles et adresses physiques. Pour une récupération plus rapide, une mémoire tampon de traduction (Translation Lookaside Buffer, TLB) met en cache les traductions récentes effectuées par la MMU. Lorsqu'une adresse de mémoire virtuelle est référencée par un processus, la TLB est d'abord consultée pour rechercher la traduction. Dans le cas d'une absence de TLB, ou autrement dit lorsque la traduction n'est pas trouvée, la MMU parcourt alors le répertoire de pages pour effectuer la traduction.

3.4 Mémoire cache

La mémoire a généralement une fréquence plus basse que le processeur, c'est pourquoi elle représente souvent un goulot d'étranglement, c'est pourquoi une mémoire cache peut être utilisée sur les processeurs à une fréquence plus élevée mais cela augmente la complexité. C'est une mémoire rapide qui se situe entre la mémoire principale et les registres du processeur dans la hiérarchie de la mémoire. La mémoire cache stocke les données et les instructions les plus récemment accédées, afin de permettre au processeur de traiter

beaucoup plus rapidement les accès futurs à des données et instructions particulières. Les caches ont leur propre hiérarchie. La plupart des systèmes possèdent trois niveaux de cache [52]. Une configuration typique de la hiérarchie des caches peut inclure des caches d'instructions et de données de niveau 1 (L1) séparés mais privés, des caches L2 combinés mais privés, et un cache de dernier niveau (LLC) partagé entre tous les cœurs. Le cache L1 comprend généralement des dizaines de kilo-octets (Ko), le L2 contient jusqu'à quelques méga-octets (Mo), et le L3 ou LLC a une capacité encore plus importante (par exemple, des dizaines de Mo).

3.4.1 Cohérence de cache

La cohérence de cache est un aspect préoccupant dans un environnement multi-cœur en raison des caches locaux distribués. Comme chaque cœur possède son propre cache, la copie des données dans ce cache n'est pas toujours la version la plus récente. La cohérence de cache désigne le mécanisme qui assure la cohérence des données stockées dans les caches locaux.

Le moyen le plus simple et le plus sûr pour garantir sa cohérence est de mettre à jour toutes les copies dès qu'un processeur écrit dessus. Cependant, cela conduit à une baisse des performances du traitement parallèle car tous les processeurs doivent attendre que leurs copies soient mises à jour. C'est pourquoi il existe plusieurs modèles dans lesquels la mise à jour des copies peut être retardée. La cohérence du cache est une propriété d'un espace mémoire individuel, tandis que la cohérence de la mémoire fait référence à l'ordre des accès à tous les espaces mémoire. La cohérence de la mémoire garantit que le résultat correct du programme ne sera pas affecté par des retards dans les mises à jour de la mémoire. Il existe de nombreux modèles de cohérence dans lesquels les copies de cache doivent être mises à jour, tels que la cohérence séquentielle et la cohérence du processeur. L'implémentation de la cohérence de la mémoire est appelée synchronisation. Dans les systèmes de mémoire partagée, la synchronisation peut être implémentée implicitement en utilisant la cohérence de cache. De nombreux chercheurs pensent que la cohérence de cache ne sera pas adaptée à un grand nombre de cœurs [43], [53]. Cependant, Martin et d'autres [76] montrent que c'est possible, grâce à une combinaison des techniques utilisées pour mettre à jour la cohérence de cache. Il existe deux méthodes de base pour assurer la cohérence de cache, notamment le snooping.

3.4.2 Le protocole Snooping

Les protocoles de cohérence de cache snooping ne peuvent être utilisés que dans des systèmes où plusieurs processeurs sont connectés par un bus partagé [27]. Par conséquent, le protocole snooping profite de ce bus en utilisant la méthode de diffusion (broadcast). Les protocoles snooping exploitent principalement deux techniques différentes pour assurer la cohérence : l'invalidation en écriture et la diffusion en écriture. Dans le cas de l'invalidation en écriture, un processeur envoie des messages d'invalidation à tous les autres processeurs

qui ont des copies en cache des données partagées, puis il met à jour sa propre copie. Dans le cas de la diffusion en écriture, un processeur diffuse les mises à jour effectuées sur les données partagées aux autres processeurs qui ont une copie en cache, de sorte que toutes les copies des données partagées soient identiques. Les processeurs peuvent lire les données partagées sans aucun problème de cohérence ; en revanche, un processeur doit avoir un accès exclusif au bus pour pouvoir écrire. Le protocole MESI [25] et le protocole MOESI [7] sont des exemples de protocoles de cohérence de cache de type snooping. Les protocoles snooping ne sont pas modulables car ils nécessitent une connexion de bus partagée, ce qui limite le nombre de processeurs pouvant être attachés au bus. En outre, les protocoles snooping utilisent des techniques de diffusion qui limitent les performances en raison de la concurrence pour les ressources partagées.

3.5 Interruptions

Les interruptions sont un signal envoyé au processeur pour lui signaler qu'un événement important doit être traité. Lorsqu'une interruption se produit, le système d'exploitation sauvegarde le contexte d'exécution de la tâche en cours et traite l'interruption. Toutes les interruptions ont un gestionnaire d'interruption associé qui est invoqué en réponse à l'arrivée de cette interruption. Il existe différentes variantes d'interruptions :

1. Les exceptions : Les exceptions sont déclenchées par le processeur pour informer le système d'exploitation d'une condition qui a empêché l'exécution d'une instruction. Certaines exceptions, telles que les exceptions de division par zéro ou de virgule flottante, sont causées par des erreurs logiques dans l'application, ce qui conduit généralement à son arrêt. D'autres, comme les défauts de page ou les défauts de débogage, nécessitent l'intervention du système d'exploitation pour traiter l'exception, après quoi le processus peut continuer à fonctionner normalement.
2. Interruptions matérielles : Les interruptions matérielles sont provoquées par des périphériques matériels, tels que le clavier ou un timer, ou même un autre processeur. Elles informent le système d'exploitation que l'état du périphérique matériel a changé d'une manière qui pourrait intéresser le système d'exploitation. Par exemple, un clavier peut informer le système d'exploitation que l'utilisateur vient d'appuyer sur une touche. Parfois, il peut être gênant pour le système d'exploitation de s'occuper d'une interruption matérielle. Dans ce cas, les interruptions peuvent être temporairement désactivées.
3. Interruptions logicielles : Les interruptions logicielles sont souvent mises en œuvre en tant qu'appels système. Les interruptions logicielles sont similaires aux exceptions, mais elles ne sont pas causées par une condition survenue lors de l'exécution d'une instruction, mais plutôt par le résultat direct de l'exécution d'une instruction spéciale "trap". Les processus utilisent les interruptions logicielles pour demander des services au système d'exploitation. Pour accéder à ces services, un processus peut effectuer un appel système, qui commence par une interruption logicielle.

3.6 Bus et périphériques sur une puce

Le SoC comprend des bus et des périphériques, comme le montre la figure 3.3, où les bus servent à la communication entre les différents blocs à l'intérieur de la puce et les périphériques permettent de communiquer avec le monde extérieur. Les bus sont les réseaux d'interconnexion de SoC les plus simples et les plus utilisés pour connecter les différentes entités physiques internes [79].

3.6.1 Bus

Le bus est un ensemble de fils de connexion auquel sont reliés un ou plusieurs composants (qui doivent communiquer entre eux). Un seul composant peut transférer des données sur le bus partagé à la fois. Des normes pour les bus sont disponibles afin de faciliter la connexion de différents composants [80]. Elles spécifient l'interface entre les composants et l'architecture du bus, ainsi que le protocole de transfert des données. ARM définit une norme pour les bus appelée Advanced Microcontroller Bus Architecture (AMBA), elle comporte de nombreuses spécifications, versions, types de bus, etc. Advanced High-Performance Bus (AHB) est utilisé comme structure de base pour les systèmes à haute performance et prend en charge les connexions entre les processeurs, les communications sur puce et les communications hors puce. Il y a aussi l'Advanced eXtensible Interface (AXI), qui est destinée aux conceptions de systèmes à haute performance et à haute fréquence d'horloge et comprend des caractéristiques qui la rendent adaptée à l'interconnexion à grande vitesse. ARM utilise l'Advanced Peripheral Bus (APB) qui a des accès de contrôle à faible bande passante et qui est connecté aux périphériques du système.

3.6.2 Périphériques

Un SoC typique intègre un processeur complexe, composé d'un cœur et de dizaines de périphériques. Ces périphériques sont les composants destinés à un usage spécifique situé à l'extérieur du cœur. Ils génèrent habituellement des interruptions pour signaler un événement au cœur. Certains d'entre eux ont des connecteurs sur le SoC qui sont utilisés pour communiquer avec un dispositif extérieur à travers une interface comprenant un connecteur spécifique. Différents types d'interfaces existent sur un SoC pour connecter des dispositifs physiques. Certains dispositifs physiques sont des sources d'information (par exemple, les capteurs) et nécessitent une interface de type entrée du point de vue du processeur, tandis que d'autres dispositifs sont des consommateurs d'information (par exemple, les écrans) et nécessitent une interface de sortie pour la connectivité. Les connecteurs physiques du SoC sont très paramétrables et peuvent être configurés soit en entrée, soit en sortie, et ils peuvent également être configurés pour d'autres fonctionnalités.

La communication entre les périphériques se fait via un ou des bus qui connectent les différents composants entre eux comme illustré sur la figure 3.3. Il existe de nombreux périphériques, les plus populaires sont les contrôleurs : GPIO, UART, SPI, I2C et CAN.

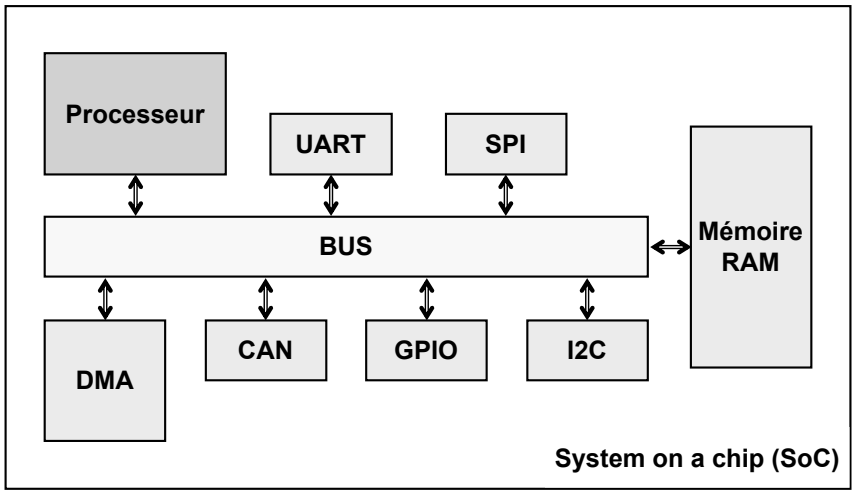

FIGURE 3.3 – Bus et périphériques du SoC

- Un GPIO est un connecteur physique dans un SoC qui peut être configuré par le programme pour devenir soit une entrée numérique, soit une sortie numérique. Lorsqu'il est configuré comme une sortie numérique, il permet de transformer les niveaux logiques du programme en niveaux de tension correspondants sur le connecteur associé, en effectuant une opération d'écriture sur l'adresse du GPIO. Lorsque GPIO est configuré en entrée, un programme peut lire des signaux numériques externes afin d'obtenir l'état actuel du matériel connecté.

- UART est l'un des protocoles de communication série les plus simples et les plus utilisés. Les normes spécifiques basées sur l'UART, notamment RS232, RS422 et RS485, sont largement utilisées dans l'industrie pour les communications longue distance, qui peuvent être de l'ordre de plusieurs kilomètres pour certaines de ces normes.

- La communication série SPI et I2C est largement utilisée pour les communications à courte distance comme la communication entre processeurs ou entre processeurs et périphériques.

- Le bus CAN a été développé à l'origine pour les applications automobiles. Cependant, grâce à sa capacité de communication très fiable, cette technologie est également utilisée dans les domaines de l'automatisation des usines, de l'avionique, de la robotique, du médical, du militaire, etc.

Les périphériques sont généralement mappés en mémoire, ce qui signifie que l'accès aux registres d'un périphérique (en lecture comme en écriture) se fait à travers des adresses mémoire spécifiques. Les périphériques ne sont pas autorisés à accéder directement à la mémoire, cela doit être fait en utilisant un DMA, qui est un périphérique spécial dont le rôle est de permettre aux périphériques d'accéder directement à la mémoire indépendamment du processeur.

3.7 Architecture ARM

La société ARM ne fabrique pas ses propres puces, mais accorde des licences de sa propriété intellectuelle à d'autres entreprises. Certains des principaux acteurs du secteur créent leurs propres puces ARM, parmi lesquels STMicroelectronics, Texas Instruments, NXP et Xilinx. C'est l'un des points forts d'ARM ; il existe une grande variété de processeurs basés sur ARM, et leur utilisation et leur fonctionnement varient considérablement. Il existe de petits processeurs ARM avec des options limitées. Il existe également des systèmes complets, contenant tout ce qui est nécessaire pour un petit ordinateur. Les processeurs i.MX6 de NXP, par exemple, contiennent un contrôleur DDR, un contrôleur ATA, un contrôleur Ethernet, une mémoire flash, un contrôleur USB et un contrôleur vidéo, le tout sur une seule puce, ce qui réduit considérablement le besoin de composants externes.

Arm a plusieurs versions d'architecture dont ARMv7, ARMv8 et récemment ARMv9, elles sont largement utilisées dans des puces utilisées par les systèmes embarqués. Les processeurs basés sur l'ARMv7 n'ont que des registres 32 bits, tandis que l'ARMv8 introduit un mode 64 bits à côté du mode 32 bits existant et peut donc supporter ces deux modes d'exécution [117] :

- *AArch64* qui présente des améliorations pour les registres 64 bits, les accès à la mémoire et les instructions 64 bits.
- *AArch32* qui est optionnel dans la spécification de l'architecture ARMv8. Son rôle est de maintenir la compatibilité rétroactive avec l'*Instruction Set Architecture* (ISA) 32 bits ARMv7.

Les premiers enjeux du développement de l'ISA ARMv8 étaient la possibilité d'accéder à un grand espace d'adressage virtuel (jusqu'à 48 bits à partir d'un registre de base de la table de translation) et des performances natives plus élevées. L'ARMv8 comporte également une unité Single instruction, multiple data (SIMD) avancée prenant en compte l'intégralité de la norme IEEE 754 et des instructions à virgule flottante supplémentaires pour IEEE754-2008 [36]. Ces extensions ont été conçues spécifiquement pour répondre aux exigences de haute performance et permettre l'expansion des processeurs ARM des systèmes embarqués aux mobiles (smartphones/tablettes), ordinateurs de bureau et serveurs. Il existe trois familles de processeurs ARM : Cortex-A, Cortex-R et Cortex-M.

- *Cortex-A*

  Les puces les plus puissantes sont basées sur les processeurs d'application, Cortex-A, qui exécutent des systèmes d'exploitation complets, des codecs multimédia hautes performances et des applications exigeantes. La série Cortex-A comprend tout ce qui est nécessaire pour les systèmes d'exploitation complexe. Ils sont suffisamment puissants et sont dotés d'une MMU permettant l'utilisation de mémoire virtuelle, et l'isolation mémoire de processus. En ajoutant quelques composants externes, il est possible de créer des plateformes avancées. Elles sont destinées principalement aux appareils mobiles qui nécessitent une puissance de calculs ou graphiques avancés.

Les smartphones, les tablettes et les téléviseurs numériques sont quelques exemples qui utilisent le Cortex-A, et même des ordinateurs portables ont été développés en fonctionnant avec des cœurs Cortex-A multi-cœurs. Le premier cœur Cortex-A a été annoncé en 2005, et son développement s'est poursuivi depuis.

- *Cortex-R*

  La gamme Cortex-R est destinée aux applications temps réel et aux systèmes critiques où la fiabilité est cruciale et la vitesse est décisive. Ils sont conçus pour être plus déterministe, capable de traiter des données qui changent rapidement et doivent être suffisamment réactifs pour gérer le flux de données. C'est la raison pour laquelle on trouve des processeurs Cortex-R dans les disques durs, les équipements de réseau et les systèmes critiques embarqués, comme l'assistance au freinage des voitures. Dans le modèle multicœur, jusqu'à 4 cœurs peuvent être utilisés sur la même puce, et le monocœur est également disponible.

- *Cortex-M*

  Le processeur Cortex-M est conçu pour les applications de microcontrôleur. Ces applications nécessitent généralement peu de puissance de traitement, mais beaucoup de lignes d'entrée et de sortie, une réponse déterministe aux interruptions et une consommation d'énergie très faible. Ils tournent généralement à un niveau de performance inférieur à celui des séries A ou R. Les puces Cortex-M sont largement utilisées dans les dispositifs bluetooth, les contrôleurs d'écran tactile, les dispositifs de commande à distance et sont même intégrées directement dans certains câbles. Certains appareils utilisant le Cortex-M se vantent d'avoir une autonomie de plusieurs années, et pas seulement de quelques heures. La carte Cortex-M est souvent extrêmement petite, dans certains cas, elle ne fait que quelques millimètres carrés.

3.7.1 Comparaison entre le Cortex-M et le Cortex-A

Grâce au système de nommage d'ARM, il est plus facile de savoir immédiatement à quoi sert un cœur. La famille Cortex compte trois familles : le Cortex-A, le Cortex-R, et le Cortex-M. Un Cortex-A, pour Application, est connecté à une grande quantité de mémoire et fonctionne à une vitesse d'horloge relativement élevée. Il est conçu pour gérer plusieurs processus à la fois, tout en exécutant un système d'exploitation complet. Il peut être utilisé comme processeur principal sur les appareils mobiles qui nécessitent une puissance de calcul rapide, tout en consommant peu d'énergie. Il est présent dans les téléphones mobiles, les tablettes, les appareils photo numériques et à peu près tous les appareils mobiles grand public. Un Cortex-M, en revanche, conçu pour le monde des microcontrôleurs, a beaucoup moins de mémoire, tourne à une vitesse d'horloge plus lente mais nécessite beaucoup moins d'énergie et est beaucoup plus petit. Il est souvent utilisé pour contrôler des périphériques matériels ou pour servir d'interface entre le matériel et un autre processeur, la plupart des clés USB Bluetooth contiennent un processeur Cortex-M. Ces deux cœurs peuvent être utilisés pour des fonctions distinctes, mais ils sont souvent utilisés ensemble. Un Cortex-M plus petit peut prendre en charge une partie du travail

d'un Cortex-A plus grand en gérant la connexion des périphériques, la sortie des données ou la régulation de l'alimentation électrique.

3.7.1.1 Modes de consommation d'énergie sur Cortex-M

Par défaut, le microcontrôleur démarre en mode normal après la mise sous tension ou la réinitialisation. Le microcontrôleur peut être configuré pour fonctionner dans l'un des modes basse consommation suivants.

- *Mode veille* : En mode veille, le processeur est arrêté, le contenu des registres et de la mémoire SRAM est conservé, tous les périphériques continuent de fonctionner et peuvent réveiller le processeur lorsqu'une interruption ou un événement se produit.
- *Mode d'arrêt* : Le mode d'arrêt permet d'obtenir une plus faible consommation d'énergie tout en conservant le contenu des registres et de la mémoire SRAM. Toutes les horloges sont arrêtées, la boucle à phase asservie (PLL) et les oscillateurs à cristal sont désactivés. Le régulateur de tension peut également être mis en mode basse consommation.
- *Mode d'attente* : Le mode d'attente est utilisé pour obtenir la plus faible consommation d'énergie. Le régulateur de tension interne est désactivé de sorte que l'ensemble du domaine de tension est désactivé. La boucle à phase asservie et les oscillateurs à cristal sont également désactivés. Après l'entrée en mode d'attente, le contenu de la SRAM et des registres est perdu. Les dispositifs sortent de ce mode lorsqu'une réinitialisation externe ou un événement spécial de réveil se produit.

3.7.1.2 Dynamic Voltage and Frequency Scaling

Dynamic Voltage and Frequency Scaling (DVFS) [66] est l'une des techniques de gestion thermique dynamique les plus connues, utilisée pour économiser de l'énergie sur une large gamme de systèmes informatiques [56]. Elle est largement mise en œuvre dans les microprocesseurs récents comme le Cortex-A. La DVFS permet de réduire dynamiquement la consommation d'énergie des processeurs informatiques en réduisant la fréquence à laquelle ils fonctionnent, comme le montre la formule suivante :

$$P = CfV^2 + P_{static}$$

où P est le total de la consommation d'énergie, C est la capacité des portes du transistor, f est la fréquence, V est la tension d'alimentation et P_{static} est la consommation d'énergie statique. La formule ci-dessus met en évidence le fait que la dissipation de puissance dynamique dépend quadratiquement de la tension, et qu'en la réduisant, une réduction significative de la consommation d'énergie peut être obtenue. Cependant, V ne peut pas être réduit à volonté, car sinon le système deviendrait instable et lorsque V atteint la tension de sous-seuil, une grave dégradation de la performance est constatée. Cela décourage les techniques agressives de réduction de tension. La tension requise pour un fonctionne-

ment stable est déterminée par la fréquence à laquelle le circuit fonctionne, et peut être réduite si la fréquence est également réduite. Ainsi, en sélectionnant une configuration stable de la combinaison tension/fréquence, le DVFS choisit dynamiquement le meilleur compromis entre la consommation d'énergie et les performances.

3.7.2 Système sur une puce

Certains systèmes embarqués sont très petits et ne contiennent que les composants strictement nécessaires à l'application. L'avantage d'un tel système est souvent le coût et la conservation de l'énergie. Pour d'autres designs, il est possible d'utiliser un système sur puce (SoC) qui est une seule puce contenant le processeur et presque tous les composants nécessaires à un système complet [93]. Il y a quelques années, les systèmes SoC étaient très chers, mais avec le marché actuel et la quantité de processeurs fabriqués, cela a permis de fabriquer certaines puces SoC à un prix relativement bas, et dans certains cas, moins cher que d'avoir un simple processeur et d'ajouter du matériel pour répondre aux besoins. La quantité de recherche et développement requise pour créer un circuit imprimé avec tous les composants nécessaires l'emporte souvent sur l'avantage d'avoir une seule puce. La plupart des SoC offrent un support complet pour au moins un système d'exploitation complet, et souvent plusieurs. L'installation d'un système d'exploitation sur une carte opérationnelle ne prend souvent que quelques minutes. ARM conçoit et accorde des licences pour ses processeurs, mais les utilisateurs de ces licences créent souvent des systèmes SoC très performants, comprenant tout ce qui est nécessaire à un ordinateur monocarte. Le processeur de la série i.MX7ULP de NXP contient un contrôleur de mémoire externe LPDDR2/LPDDR3, ports USB, Ethernet, PCI Express, un GPU, aux côtés d'un Cortex-A7 et un Cortex-M4, le tout sur une seule puce [100].

Dans certains cas, le projet exige un matériel spécifique et nécessite des périphériques personnalisés. Dans ce cas, on peut utiliser un SoC contenant un FPGA, un cœur ARM intégré et suffisamment de cellules logiques pour compléter la conception.

3.7.3 Système sur une puce à architecture hétérogène

Les architectures hétérogènes peuvent inclure divers processeurs à usage général ainsi que des accélérateurs dans le même système sur une puce. Il existe plusieurs types d'hétérogénéité. Elle peut être basée sur la diversité des ressources matérielles de tous les éléments que l'on trouve dans les SoC telles qu'un CPU, un GPU, un processeur de signal numérique (DSP), et d'autres accélérateurs.

Face à la demande croissante en termes de puissance des processeurs, plusieurs études [8] ont montré que l'utilisation de différents cœurs aux caractéristiques matérielles hétérogènes au sein d'un même processeur présente de nombreux avantages : elle permet d'obtenir de meilleures performances et de réduire la consommation d'énergie par rapport aux cœurs homogènes [63]. Les systèmes AMP présentent de nombreux défis pour les

systèmes d'exploitation qui doivent prendre en compte le matériel partagé en implémentant des algorithmes de synchronisation capables d'éviter les conflits d'accès. Cependant, les ordonnanceurs actuels ne sont pas capables de répartir la charge de calcul sur chaque cœur proportionnellement à son efficacité de calcul, sauf pour certaines plateformes comme big.LITTLE car ils appartiennent à la même famille.

Encore récemment, la plupart des plates-formes multicœurs étaient homogènes, comprenant plusieurs cœurs identiques capables d'accéder à la mémoire principale en utilisant le même bus, ce qui permettait à un ordonnanceur global de répartir les tâches de manière globale sur chaque cœur. Suite aux besoins du marché, plusieurs fabricants de SoC proposent aujourd'hui de nouveaux systèmes multicœurs hétérogènes où ils combinent au moins deux types de cœurs différents, par exemple un microcontrôleur et un ensemble de microprocesseurs. Cette technologie devient très populaire dans le monde de l'informatique, ceci n'est pas seulement dû au fait qu'on peut dédier un processeur à une fonction spécifique, mais aussi à la possibilité d'utiliser un système d'exploitation entièrement différent sur chaque cœur, cependant, à part sur quelques plateformes spécifiques où les types de cœurs sont de la même famille, il n'existe pas de système d'exploitation capable de gérer conjointement les différents types de cœurs. Les architectes de SoC créent des types mixtes de cœurs de traitement pour les systèmes complexes afin d'effectuer des opérations complexes de manière efficace, ils combinent les microcontrôleur et microprocesseurs sur le même système multicœur. Par exemple, la carte Ultrascale+ est composée d'un processeur ARM Cortex-A53 à quatre cœurs, deux ARM Cortex-R5 et un FPGA [14] ou la carte STM32MP1 ayant un SoC avec un ARM Cortex-A7 à deux cœurs et un ARM Cortex-M4 [108].

Les microcontrôleurs offrent généralement une puissance de traitement plus faible, mais consomment moins d'énergie et dissipent moins de chaleur que les microprocesseurs.

Dans un contexte temps réel, ces caractéristiques soulignent la question de savoir comment mesurer le temps d'exécution d'une application qui commence sur un cœur, puis migre à travers le mécanisme de communication interprocesseur, intégré physiquement dans la puce, pour finir sur un autre type de cœur.

Dans cette thèse, nous allons considérer ces dimensions et les approches utilisées pour chacune d'entre elles afin d'obtenir le système le plus efficace dans un environnement AMP.

3.8 ARM big.LITTLE

Un cas particulier de calcul hétérogène est le SoC big.LITTLE développé par ARM. Ce système a été annoncé pour la première fois en 2011 et a rapidement gagné en popularité. Le système associe, en utilisant un bus, un processeur à haute performance et un processeur à basse consommation dans une combinaison configurable. Les deux processeurs utilisent la même architecture de jeu d'instructions ISA et sont donc capables d'exécuter

le même code compilé [35]. L'objectif est d'obtenir une meilleure efficacité énergétique en utilisant l'hétérogénéité du système pour diriger les tâches vers le type de processeur pour lequel elles sont le plus optimisées.

Il possède deux types de processeurs avec des cœurs de traitement compatibles au niveau binaire, qui diffèrent en termes de complexité de la microarchitecture. Le processeur le plus puissant est appelé big et le processeur le plus économe est appelé LITTLE. Le big est destiné aux applications exigeantes des performances élevées, comme la navigation sur Internet, le multimédia et les jeux. En revanche, le LITTLE est destiné aux applications à faible demande énergétique, comme les appels téléphoniques, la messagerie, le courrier électronique et l'écoute de musique.

Kumar et al. [64] démontrent les capacités de tels systèmes à un seul ISA. Ils ont testé diverses configurations matérielles et politiques d'ordonnancement et ont conclu que les systèmes ISA hétérogènes atteignent globalement une amélioration de 18 à 30 % de l'efficacité énergétique par rapport à une architecture homogène de même surface, selon le degré de complexité de l'ordonnanceur.

Hill et al. [40] ont réalisé une étude complète sur l'extension de la loi d'Amdahl [37] dans le contexte du multicœur. Ils explorent les conceptions multi-cœurs symétriques, asymétriques et dynamiques. Dans le cas de la conception symétrique, tous les cœurs sont parallèles ou regroupés en unités plus grandes, conformément à la règle de "Pollack". Cette règle stipule que l'augmentation des performances est proportionnelle à la racine carrée de l'augmentation de la complexité de la microarchitecture [15]. Une interprétation de cette règle est que la plus grande performance par surface peut être obtenue en utilisant une grande quantité de cœurs de traitement à faible complexité dans un réseau de processeurs massivement parallèles. À l'inverse, le système asymétrique comporte un grand cœur combiné et de nombreux petits en parallèle.

Il est évident que les systèmes ISA hétérogènes tels que big.LITTLE d'ARM pourraient permettre au marché des appareils mobiles ou des systèmes embarqués de répondre à la demande des consommateurs et représenter la voie à suivre pour l'ensemble du secteur. Cependant, en raison de la complexité accrue de ces systèmes et de la variation de leur consommation d'énergie, une attention particulière doit être accordée à l'aspect logiciel et notamment aux politiques de gestion de l'énergie et de l'ordonnancement.

L'architecture big.LITTLE bénéficie du fait que les deux processeurs séparés sont identiques et peuvent accéder à la mémoire centrale en utilisant l'interconnexion, capables de traiter les mêmes instructions et les mêmes applications logicielles de haut niveau. Cela s'avère extrêmement bénéfique dans les applications consommant peu d'énergie. Pour réaliser de manière efficace un système basé sur l'architecture big.LITTLE, le temps nécessaire à la migration d'une tâche entre les cœurs doit être pris en compte. Un temps déterminé sera nécessaire pour sauvegarder l'état (qui inclut le contenu de tous les registres et caches) des cœurs, et ce temps ne doit pas être trop long. Si la latence de changement de contexte dépasse un certain seuil, le système subira une dégradation sensible de ses performances. Pour effectuer le changement en un minimum de temps, un bus d'interconnexion spécia-

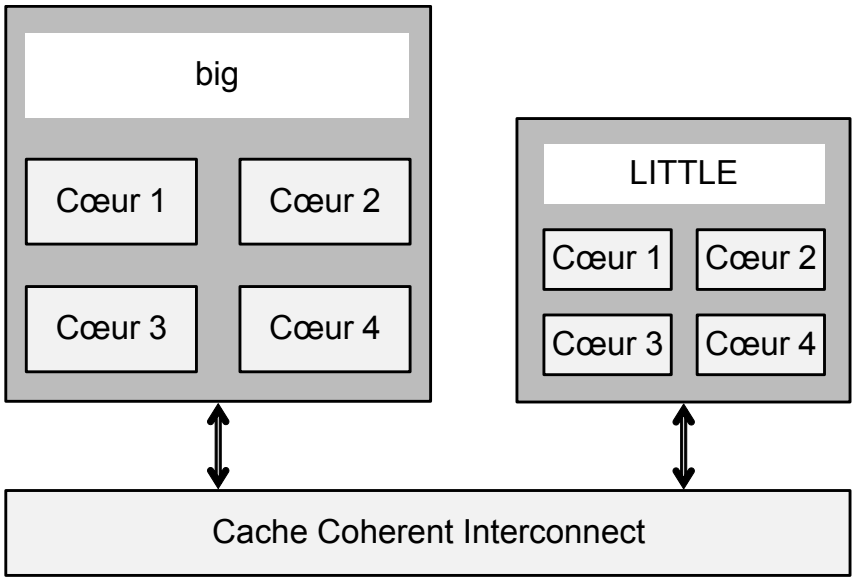

FIGURE 3.4 – Interconnexion de big.LITTLE

lement conçu, appelé CCI (Cache Coherent Interconnect), est utilisé pour transférer les données entre les cœurs, comme le montre la figure 3.4.

3.8.1 Gestion de l'énergie sur big.LITTLE

Dans le contexte des systèmes embarqués hétérogènes, en particulier le SoC big.LITTLE, le système de gestion de l'énergie présente un niveau de complexité supplémentaire. À cette fin, ARM a amélioré les systèmes d'exploitation qui prennent en charge un ordonnanceur spécialisé pour big.LITTLE [92]. L'ordonnanceur inclue une extension de DVFS, qui permet de migrer les tâches d'un cluster de CPU à l'autre. Les cœurs de traitement big et LITTLE sont regroupés en cœurs virtuels, chacun donnant accès à un cœur physique de chaque type [89].

Comme illustré dans la figure 3.5, chaque cœur virtuel ne peut avoir qu'un seul cœur physique actif à la fois (soit le big, soit le LITTLE), contrôlé par la politique d'ordonnancement. La migration des tâches ne se produit qu'à l'intérieur du cœur virtuel, donc le système ne peut pas avoir plus de 4 cœurs actifs à la fois.

Le système d'exploitation peut ordonnancer indépendamment des tâches pour chaque cœur big ou LITTLE [48]. La migration des tâches peut être effectuée entre deux cœurs, quelle que soit leur attribution physique dans le cluster.

Grâce à l'interconnexion de cohérence du cache, le coût de la migration de la tâche reste bas. L'ordonnanceur a 3 modes de fonctionnement, en fonction de l'implémentation [89] [47] [48] :

- *Migration de cluster* : La migration des clusters est la première solution développée pour la migration des tâches. L'ordonnanceur du système d'exploitation ne peut sélectionner qu'un seul des clusters pour exécuter toutes les tâches. Cela signifie

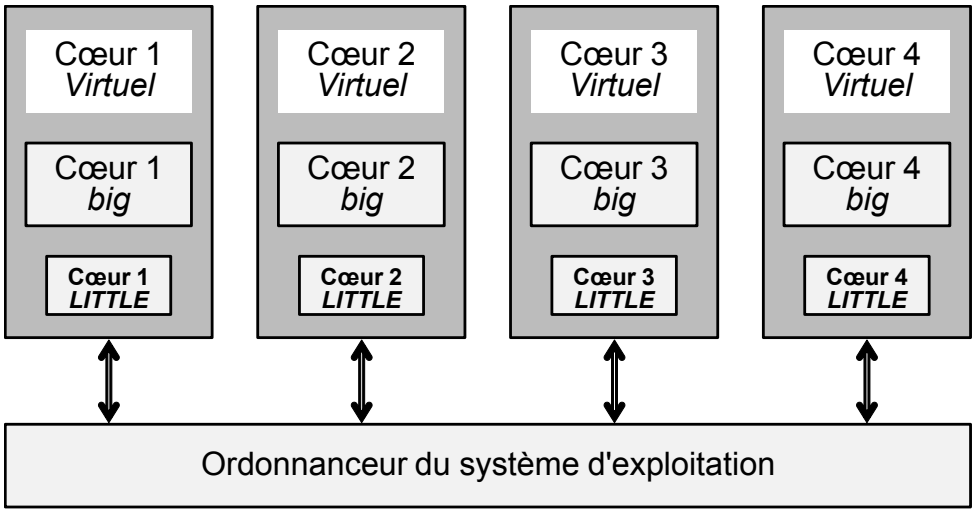

FIGURE 3.5 – Cœurs virtuels big.LITTLE

qu'un seul groupe de cœurs peut être utilisé à la fois.

- *In-Kernel Switcher* : L'In-Kernel Switcher est la deuxième solution d'ordonnancement développée par ARM. Au lieu de migrer toutes les tâches entre les clusters, il associe un noyau big et un noyau LITTLE en un seul noyau virtuel. Le cœur qui est utilisé dépend des besoins en performances, l'autre cœur est mis à l'arrêt. Cela limite toujours le nombre de cœurs pouvant être utilisés par le système, mais permet une plus grande flexibilité et les cœurs big et LITTLE peuvent être activés en même temps, tant qu'ils ne font pas partie du même cœur virtuel [89].

- *Ordonnanceur de tâches global* : Cet ordonnanceur permet la migration entre deux différents cœurs de processeur, y compris ceux de même type. Cela permet par ailleurs d'exploiter toutes les capacités du système avec la possibilité d'utiliser tous les cœurs en même temps. La priorité d'allocation des tâches est donnée aux cœurs big.

3.8.2 DynamIQ

DynamIQ est le successeur de big.LITTLE, il permet plus de flexibilité lors de la conception de processeurs multi-cœurs. DynamIQ combine les clusters big et LITTLE pour former un seul cluster de CPU, composé de processeurs big et LITTLE, appelé DynamIQ big.LITTLE. Ces systèmes intègrent des fonctions de puissance intelligentes dans le cluster qui permettent de bénéficier des performances dans les limites thermiques fixes. Cela signifie plus de traitement de données et de performances, offrant de meilleurs rendements, quelle que soit l'application utilisée.

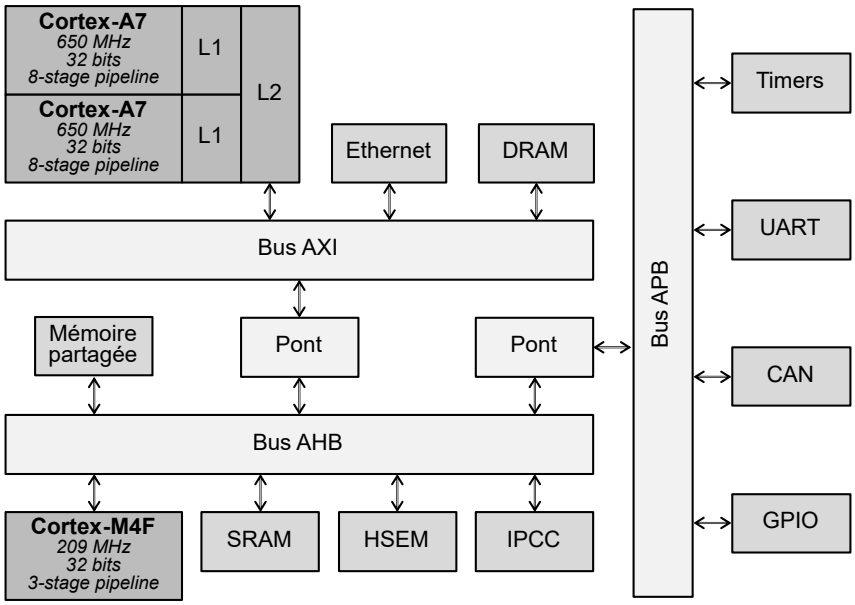

FIGURE 3.6 – Schéma bloc simplifié de la carte STM32MP1

3.9 Système asymétrique hétérogène

Un système HMPSoC embarqué est caractérisé par plusieurs cœurs de calcul homogènes et/ou hétérogènes intégrés dans un SoC. On parle d'hétérogénéité lorsque les cœurs ont différents ISA, conceptions matérielles et/ou caractéristiques (comme la consommation d'énergie et les performances) différents. Ces cœurs ont des capacités d'exécution différentes, c'est pourquoi chaque cœur ou groupe de cœurs est dédié un type d'application spécifique, en fournissant les systèmes les plus efficaces. Il existe trois marchés principaux pour les systèmes hétérogènes : mobile, IoT et embarqué. Dans les deux, le but principal de l'hétérogénéité est de réduire la consommation d'énergie, mais pour les systèmes embarqués, il y a parfois une autre exigence qui est le déterminisme et la sûreté de fonctionnement. C'est pourquoi, dans les systèmes embarqués, la tendance est à ajouter un cœur dédié au traitement temps réel, qui se chargera de toutes les tâches temporellement critiques pour ensuite différer les tâches non critiques sur les autres cœurs. Pour l'architecture ARM, le cœur dédié au temps réel appartient soit à la famille Cortex-M, soit à la famille Cortex-R. Le cœur à usage général est celui de la famille Cortex-A. On peut également parler d'hétérogénéité lorsqu'il est combiné à un FPGA ou à un processeur d'une architecture totalement différente comme l'architecture RISC-V.

3.9.1 Exemples de puces asymétriques hétérogènes

3.9.1.1 Série STM32MP1

La série STM32MP1 de STMicroelectronics comprend plusieurs cartes comme les modèles STM32MP1-DK2 ou STM32MP1-EV1. La carte utilisée dans les expérimentations des chapitres suivants est la STM32MP1-DK2 [108], elle possède, comme le montre la

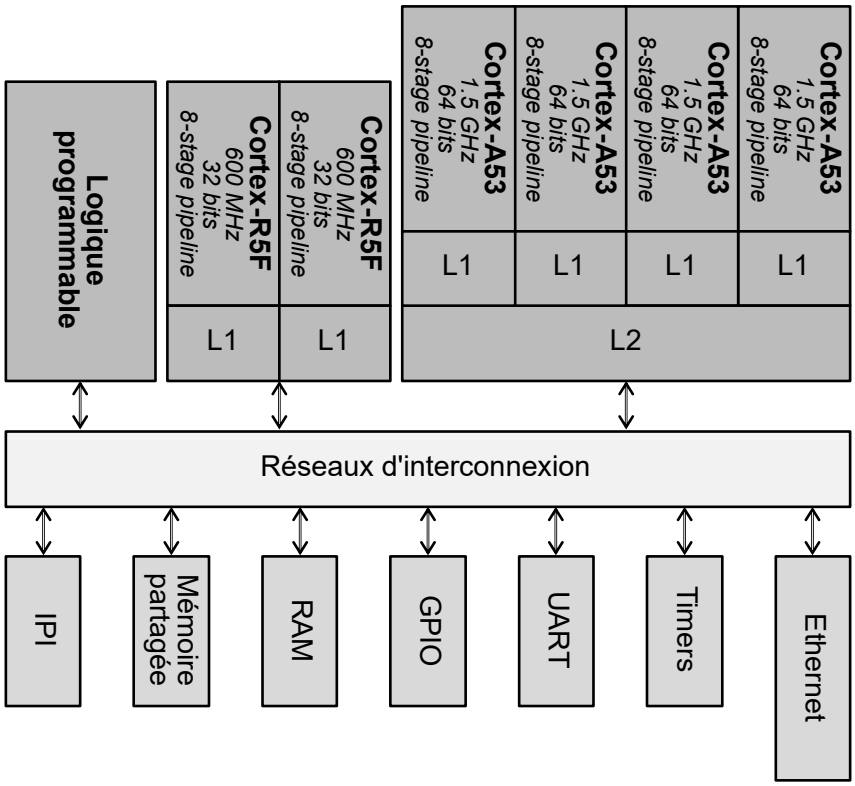

FIGURE 3.7 – Schéma bloc simplifié de la carte Zynq UltraScale+ MPSoC

figure 3.6, deux cœurs Cortex-A7 basé sur ARM-v7 avec une fréquence de 650 MHz avec deux niveaux de mémoire cache et huit étages de pipeline et un seul Cortex-M4F avec une fréquence de 209 MHz, trois étages de pipeline et sans cache. Elle possède de nombreux périphériques, 29 timers, 37 périphériques de communication (dont UART, Ethernet, USB, ...), ainsi que des périphériques mémoires comme 1 Go de RAM externe et plusieurs SRAM internes. Pour la partie communication AMP, il y a de la mémoire partagée, un contrôleur de communication inter-processeurs *Inter-Processor Communication Controller* (IPCC) et des périphériques de sémaphore matériel (HSEM). Il existe trois interconnexions principales. Le bus "AXI" où les deux Cortex-A7 sont connectés avec quelques-uns de leurs périphériques, "AHB" où le Cortex-M4F est connecté avec des SRAMs, des périphériques spécifiques AMP et d'autres périphériques. De plus, tous les autres périphériques comme les timers et l'UART sont connectés au bus "APB". Les trois bus sont interconnectés et peuvent communiquer grâce à des ponts spécifiques entre eux.

3.9.1.2 Zynq UltraScale+ MPSoC

Le Zynq UltraScale+ Multiprocessor System on a Chip (MPSoC) [14] regroupe le processeur d'application 64 bits haute performance et à faible consommation énergétique Cortex-A53 basé sur ARM-v8, le processeur temps réel Cortex-R5F d'ARM-v7 et un FPGA basé sur l'architecture UltraScale afin de créer les premiers MPSoCs programmables du secteur. Il permet de réaliser des économies d'énergie, un traitement hétérogène et une accélération programmable. Comme le montre la figure 3.7, ces derniers

sont connectés via un bus d'interconnexion et peuvent accéder aux périphériques du SoC, comme les timers, UART, ... Pour les composants spécifiques AMP, principalement pour les communications, il y a un contrôleur appelé "Inter-Processor communication Interface" (IPI) qui gère la signalisation entre les cœurs et une mémoire partagée. La particularité de cette carte est qu'il est toujours possible de concevoir un périphérique personnalisé et de l'implémenter sur le circuit logique programmable.

3.10 Communication inter-processeurs

L'un des problèmes supplémentaires qu'apportent les HMPSoCs par rapport aux systèmes homogènes est la coopération et la communication entre leurs éléments, ce qui n'est pas simple et nécessite des solutions logicielles et matérielles. Le côté matériel est nécessaire afin de constituer un composant intermédiaire qui est identifiable et contrôlable par les différents types de cœurs. Au-dessus de ces composants, une couche logicielle permet de les contrôler et fournit ainsi le protocole de communication inter-processeurs.

3.10.1 Contrôleur de communication inter processeur

IPCC [108] est le module chargé de permettre aux microcontrôleurs et aux processeurs d'échanger des messages non bloquants en utilisant la mémoire partagée qui ne fait pas partie de ce contrôleur. Le module permet aux composants de communiquer en utilisant différents modes comme Simplex, Half-duplex ou Full-duplex, il permet également à un processeur d'envoyer des signaux à l'autre processeur en utilisant des interruptions. La communication simplex utilise un canal dédié pour transmettre un message d'un cœur (ou cluster de cœurs) A au cœur (ou cluster de cœurs) B. Dans la communication semi-duplex, A et B utilisent un canal de communication partagé afin d'envoyer et de recevoir des messages entre eux. En Full-duplex, les deux sous-canaux sont utilisés, ce mode peut être considéré comme une combinaison de deux canaux simplex où les deux parties communiquent de manière asynchrone, ce qui signifie qu'elles n'attendent pas de réponse. Les informations du canal sont stockées dans la mémoire partagée créée entre les processeurs. Ce contrôleur fournit un support de communication au niveau matériel, où les deux parties (émetteur et récepteur) ont leurs propres bancs de registres et interruptions. Le message en cours de transmission est stocké dans la RAM en tant que mémoire partagée, indépendante du contrôleur IPCC. Plusieurs canaux sont disponibles pour la communication, chacun d'entre eux étant composé de deux sous-canaux :

- Sous-canal du cœur (ou cluster) 1 vers le cœur (ou cluster) 2
- Sous-canal du cœur (ou cluster) 2 vers le cœur (ou cluster) 1

La mémoire contenant le message étant partagée, il est important de s'assurer que les processeurs n'accèdent pas simultanément à cette mémoire et ne provoquent pas d'erreurs. Pour cela, les sous-canaux disposent d'un flag pour contrôler la communication. Un flag peut avoir deux valeurs, générant deux interruptions différentes :

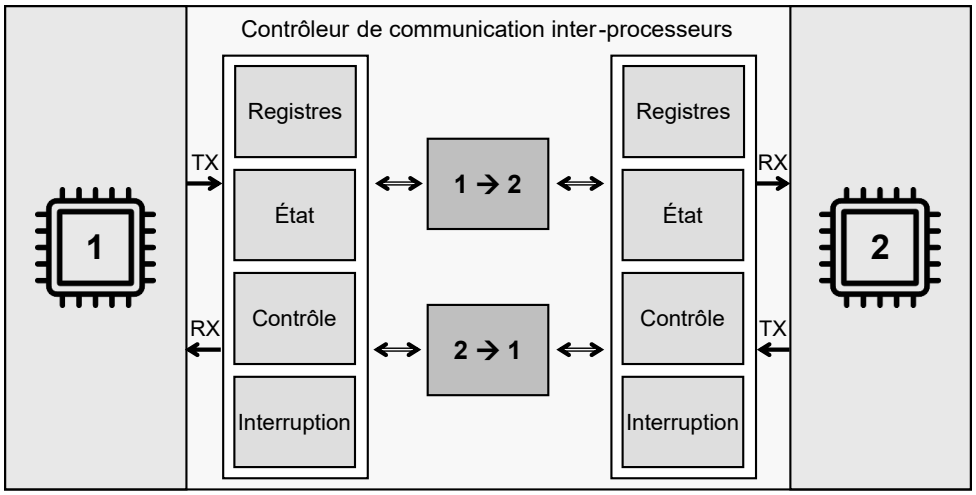

FIGURE 3.8 – Contrôleur de communication inter processeurs

- *TX* : Le processeur émetteur est informé qu'un sous-canal est prêt pour une nouvelle transmission.
- *RX* : Le processeur récepteur est informé que le sous-canal est occupé, ce qui indique un message arrivant.

La figure suivante 3.8 illustre comment les canaux et leurs sous-canaux peuvent générer une interruption qui informe le processeur destinataire de la réception d'un message entrant ou le processeur expéditeur qu'un canal est libre.

3.10.2 Les enjeux de la communication inter processeurs

La communication inter processeurs qui utilise la mémoire partagée peut causer des problèmes de concurrence. Dans un système où deux tâches ont accès à une ressource partagée, il est nécessaire d'empêcher une tâche de lire cette ressource partagée, lorsque cette dernière est encore en cours de modification par une autre tâche, afin d'éviter les incohérences de données [65].

En outre, le fonctionnement d'un système avec une stratégie d'ordonnancement à priorité statique peut entraîner des deadlocks. Les deadlocks peuvent se produire dans un système qui utilise un mécanisme basé sur le verrouillage pour éviter les inconsistances de données. Par exemple, un deadlock peut se produire si une tâche à haute priorité préempte une tâche à basse priorité, avant que cette dernière puisse libérer le verrou. Une requête de la ressource partagée de la part de la tâche la plus prioritaire conduirait à une boucle sans fin, car la ressource partagée est toujours détenue par la tâche la moins prioritaire [65].

Un tel blocage est causé par la gestion inefficace des ressources et par le manque de stratégie de synchronisation. La conséquence de telles implémentations est une perte de temps qui pourrait conduire au non-respect des contraintes temporelles, ce qui n'est pas acceptable pour les systèmes critiques.

3.10.3 Sémaphore matériel

Le sémaphore est un mécanisme utilisé dans un environnement multitâche pour garantir que deux ou plusieurs segments de code ne soient pas appelés simultanément. Pour que cela fonctionne entre différents cœurs dans un système multicœur hétérogène, un sémaphore matériel (Hardware Semaphore HSEM) est nécessaire. Il a été initialement proposé par [31] dans lequel les cœurs tournent sur une mémoire temporaire locale (connectée directement à chaque cœur) dans le but de réduire le nombre d'accès au bus partagé.

Il est aussi appelé spinlock matériel car il bloque un processeur et non une tâche du système d'exploitation. Les sémaphores matériels permettent de répondre aux besoins de synchronisation et d'exclusion mutuelle entre des processeurs hétérogènes et ceux qui ne fonctionnent pas sous le même système d'exploitation. Il n'existe pas d'autre mécanisme pour accomplir ces opérations entre des processeurs dans des sous-systèmes hétérogènes [44]. Ils ne sont pas adaptés à la synchronisation entre des tâches sur un seul processeur. Ils sont plutôt destinés à être utilisés pour la synchronisation entre différents sous-systèmes du périphérique qui ne disposent d'aucun autre moyen de synchronisation matérielle.

Les sémaphores matériels ne règlent pas tous les problèmes de synchronisation du système. Ils ont une applicabilité limitée et doivent être utilisés avec précaution pour mettre en œuvre des protocoles de synchronisation de plus haut niveau.

Le sémaphore matériel peut être utilisé pour l'exclusion mutuelle lors de l'accès à une structure de données partagée. Cependant, il ne doit être utilisé que lorsque le temps de maintien du verrou est prédictible et court, et que la tâche verrouillée ne peut pas être préemptée, suspendue ou interrompue pendant qu'elle détient le verrou. Le module de sémaphore matériel varie selon l'architecture et se trouve comme un périphérique à l'intérieur du SoC. Celui-ci possède plusieurs instances de ce sémaphore (16 par exemple sur la carte i.MX8) et il utilise plusieurs registres ; des registres pour stocker l'état (verrouillé ou déverrouillé) et des registres pour le contrôler. Il stocke également, dans un registre l'ID du cœur de CPU du détenteur du sémaphore, ce qui est nécessaire pour savoir quel cœur est autorisé à le déverrouiller.

Par exemple, pour l'exclusion mutuelle, le cœur A essaie de prendre le sémaphore avant d'accéder à un périphérique partagé, une fois qu'il est acquis, il utilise ce périphérique, si un deuxième cœur B essaie d'y accéder, celui-ci sera bloqué, jusqu'à ce que le cœur A ait terminé d'utiliser le périphérique. Il libère alors le sémaphore ce qui débloque le cœur B, il peut donc réessayer de prendre le sémaphore et d'utiliser le périphérique.

La figure 3.9 illustre le mécanisme de verrouillage du sémaphore matériel, il doit être utilisé à chaque fois que le cœur veut accéder à une ressource partagée, dans cet exemple il y a deux cœurs, le premier avec un ID égal à 1 et 2 pour le second. Chaque cœur appelle l'API de verrouillage, qui va vérifier si l'ID du cœur dans le registre interne du sémaphore est égal à l'ID du cœur qui a appelé l'API, s'ils sont égaux, alors il accédera à la ressource partagée, sinon le cœur doit être bloqué.

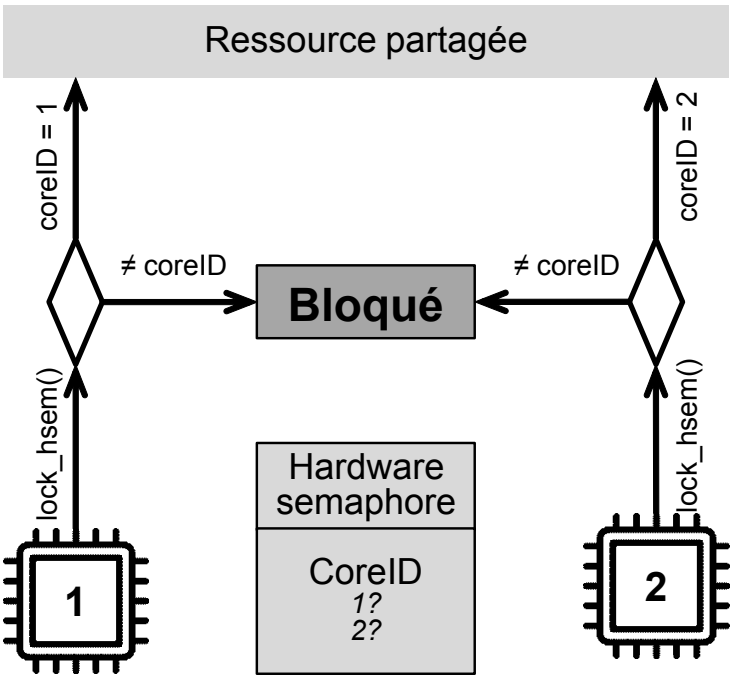

FIGURE 3.9 – Procédure de verrouillage du sémaphore matériel

3.11 Développement sur des processeurs ARM

Les architectures RISC comme ARM ont été conçues à l'origine avec un jeu d'instructions petit et simple. Cela permettait aux compilateurs d'utiliser efficacement les instructions disponibles. ARM est une architecture "load-store", cela signifie que le développeur doit explicitement charger (read) les données d'entrée de la mémoire dans les registres avant que les données puissent être traitées. De même, le développeur doit stocker explicitement les données de sortie en mémoire après leur traitement. Toutes les instructions arithmétiques utilisent le contenu des registres à la fois comme entrées et comme résultats. Les registres peuvent également être utilisés pour stocker des résultats temporaires ou intermédiaires, tels que des compteurs de boucle ou des valeurs de sous-expression. Le développeur (ou le compilateur lorsqu'il utilise un langage de haut niveau) a un contrôle total sur l'état des registres. Par exemple, lors de l'addition de deux valeurs, le développeur doit décider lequel des registres doit être temporairement affecté à chaque valeur et à la somme calculée. Les registres peuvent être réutilisés de manière arbitraire lorsque leur contenu précédent n'est plus nécessaire.

Les processeurs ARM peuvent être programmés à l'aide d'une variété de langages de programmation de haut niveau. Certains processeurs ARM peuvent même exécuter nativement le "bytecode" Java. Néanmoins, cela a un impact sur les performances, pour cela le langage de programmation le plus utilisé est le langage C, il nécessite une chaîne de compilation (formée de plusieurs outils) pour transformer le code en exécutable compatible avec la cible d'une manière très efficace.

Les deux chaînes de compilation C/C++ open source les plus populaires, GCC et Clang, incluent un support pour les processeurs ARM, permettant le développement

C/C++ dans son intégralité, ainsi que le support d'une variété de bibliothèques et du débogage. De nombreux compilateurs commerciaux pour ARM sont disponibles. Les compilateurs commerciaux, généralement, peuvent générer un code plus rapide et efficace que les compilateurs open source.

Sur les systèmes embarqués, ce n'est pas seulement une chaîne de compilation dont nous avons besoin, mais d'une chaîne de compilation croisée. En effet, l'ordinateur hôte est utilisé pour développer et compiler l'application, mais les binaires résultants sont exécutés sur la cible, qui a généralement une architecture différente.

En réalité, il existe plusieurs types de chaîne de compilation, basés sur l'architecture de la machine qui exécute la chaîne de compilation (machine hôte) et construit des binaires, et sur celle de la machine qui exécute ces derniers (machine cible) :

- *Compilateur natif* : Un exemple de ceci est une machine x86 exécutant un compilateur qui crée des binaires qui s'exécutent sur une machine x86. Ce type de compilateur est souvent utilisé pour les ordinateurs de bureau.
- *Compilateur croisé* : C'est le plus utilisé sur les systèmes embarqués ; par exemple, une machine x86 exécutant un compilateur qui génère des binaires pour une autre architecture, comme ARM.

3.12 Débogage

Un débogueur est une application logicielle capable d'exécuter un programme, ligne par ligne, et d'afficher plusieurs informations, telles que les variables et le contenu de la mémoire. Les débogueurs sont utilisés principalement pour suivre, étape par étape, l'exécution d'un programme et pour comprendre pourquoi certaines parties de ce programme ne fonctionnent pas comme prévu. Les débogueurs sont des logiciels qui ont normalement besoin d'une méthode de communication externe : une ligne série, Ethernet, un moniteur, etc. Le débogueur peut être logiciel ou matériel. Un débogueur logiciel peut prendre un programme binaire et l'exécuter exactement comme s'il s'exécutait normalement sur un système d'exploitation. Il peut mettre en pause l'exécution d'un programme, effectuer une exécution pas à pas et examiner de façon approfondie un programme. Les débogueurs matériels sont souvent utilisés pour les systèmes embarqués. Non seulement ils peuvent aider au débogage, mais aussi, en ayant un accès direct au matériel, ils peuvent transmettre des programmes à la mémoire, flasher la mémoire non volatile et configurer les périphériques matériels. Les débogueurs matériels ne servent pas uniquement à déboguer des logiciels. Ils peuvent également déboguer une grande partie du matériel, en examinant tous les registres d'un système, et pas seulement les processeurs. Si la ligne série ne fournit pas de données correctes, regarder les registres série est souvent un excellent moyen de débogage sans ajouter de code supplémentaire.

3.12.1 Débogueur ARM et JTAG

Les processeurs ARM disposent de fonctionnalités de débogage exceptionnelles grâce au matériel intégré. *Portable Joint Test Action Group* (JTAG) a été conçu à l'origine pour tester les interconnexions sur les cartes de circuits imprimés. Depuis, JTAG a été utilisé pour bien d'autres choses, notamment le débogage. Certains cœurs ARM classiques possèdent un élément de débogage matériel qui peut recevoir des instructions et accéder au processeur et aux périphériques externes. Ces périphériques ont un support matériel pour ajouter des points d'arrêt et prendre le contrôle du processeur lorsqu'une pause a lieu. Ce matériel a ensuite été remplacé par CoreSight, une version améliorée. N'étant plus basé sur JTAG, ce dispositif communique à l'aide de Serial Wire Debug (SWD), une alternative à JTAG à haut débit et à faible nombre de connecteurs. Ne nécessitant que deux connecteurs au lieu des cinq de JTAG, il permet de déboguer les modules dont le nombre de connecteurs est très limité, ce qui permet au débogueur de contrôler même les plus petites puces. CoreSight permet aux utilisateurs de disposer d'un plus grand nombre de points d'arrêt matériels et de traces matérielles, ce qui permet aux débogueurs de savoir quelles routines sont utilisées, à quel moment et pendant combien de temps.

Voici les fonctionnalités les plus importantes fournies par les débogueurs :

- *Point d'arrêt*

  aussi appelé "breakpoint", est un endroit dans le code d'instruction où le processeur s'arrête et donne le contrôle au débogueur. Dans un logiciel, cela bloque le programme à l'endroit spécifié, laissant l'utilisateur décider de ce qu'il doit faire. Dans le matériel, cela fige le processeur, de sorte que rien ne se passe en arrière-plan. Un point d'arrêt est déclenché lorsque le compteur de programme est égal à l'adresse, ou lorsque l'instruction est sur le point d'être exécutée. Les cœurs ARM équipés de matériel de débogage intégré peuvent avoir des points d'arrêt matériels, permettant à un programme de s'exécuter à pleine vitesse avant d'être arrêté à un emplacement mémoire particulier. Le nombre de points d'arrêt disponibles dépend du cœur et de l'architecture (généralement de deux à huit points d'arrêt matériels).

- *Point d'observation*

  aussi appelé "watchpoint", est légèrement différent d'un point d'arrêt, et les deux sont souvent confondus. Bien qu'un point d'arrêt puisse stopper le processeur lorsqu'une instruction spécifique est sur le point d'être exécutée, un point d'observation peut se déclencher à un emplacement mémoire et peut être réglé pour se déclencher en lecture ou en écriture. Il est extrêmement utile de savoir quelle partie d'un programme met à jour quelle partie de la mémoire. Par exemple, pour savoir ce qui modifie la valeur d'un registre, il faut définir un point d'observation sur l'écriture des données. Si le programme lit ce registre, le point d'observation est ignoré, mais dès que le programme modifie la mémoire, le système s'arrête à l'instruction. Les cœurs ARM fournissent généralement peu de points d'observation, moins que de points d'arrêt.

- *Pas-à-pas*

est une fonctionnalité importante qui permet au débogueur de parcourir le code, instruction par instruction. Lorsqu'un point d'arrêt est défini, l'instruction suivante devient visible et attend la confirmation de l'utilisateur avant de continuer. Il est possible de consulter les valeurs des variables avant de poursuivre l'exécution de l'application, mais il est parfois utile d'observer le résultat de chaque instruction. L'exécution pas-à-pas permet de le faire. Dans une boucle, il est possible d'observer le résultat de l'exécution de chaque ligne et d'afficher les variables. La plupart des débogueurs permettent le pas-à-pas dans le langage natif et en assembleur. Le débogueur peut recevoir la commande de poursuivre l'exécution jusqu'à ce qu'il quitte la routine en cours d'exécution via un retour, de sauter une fonction (après tout, il faut bien déboguer son propre code, pas toute la bibliothèque C) ou d'entrer spécifiquement dans une fonction.

3.12.2 Débogueur GNU

Le débogueur GNU, ou gdb, est un logiciel qui permet à l'utilisateur de prendre le contrôle d'un programme, de le démarrer et de l'arrêter, d'insérer des points d'arrêt, d'évaluer des variables et quelques autres fonctions importantes. Il peut déboguer des programmes écrits en C, C++ ainsi qu'un grand nombre d'autres langages. Il y a deux façons d'utiliser gdb : soit en exécutant gdb sur la cible, soit en configurant gdb pour exécuter une application cible. Le débogueur GNU est régulièrement utilisé pour déboguer des applications PC, en s'exécutant sur le même système, mais cette méthode est rarement utilisée pour les systèmes embarqués, simplement parce que l'exécution de gdb peut demander trop de ressources pour un système embarqué. Sur les systèmes embarqués, le débogueur GNU peut être utilisé de manière maître/esclave. Dans ce cas, le débogueur GNU fonctionne en deux étapes. Tout d'abord, le serveur gdb, nommé gdbserver, doit être compilé pour ce système, puis être copié sur la plate-forme cible. Il nécessite une méthode de communication avec l'ordinateur de compilation. Cela signifie que la cible doit avoir une configuration réseau fonctionnelle ou une connexion UART. Deuxièmement, sur le PC de débogage, il faut exécuter gdb puis se connecter à gdbserver. Une fois encore, une version spécifique de gdb croisé doit être utilisée et plus précisément une version ARM de gdb pour les puces basées sur ARM qui tourne sur un PC d'une architecture différente. Bien que gdbserver soit souvent logiciel, certaines implémentations matérielles existent, et sont connectées directement au port USB du système de développement. En outre, certains systèmes d'exploitation permettent aux utilisateurs de prendre le contrôle via gdb.

La figure 3.10 illustre un exemple de session de débogage sur un système embarqué, contrôlé depuis un système de développement.

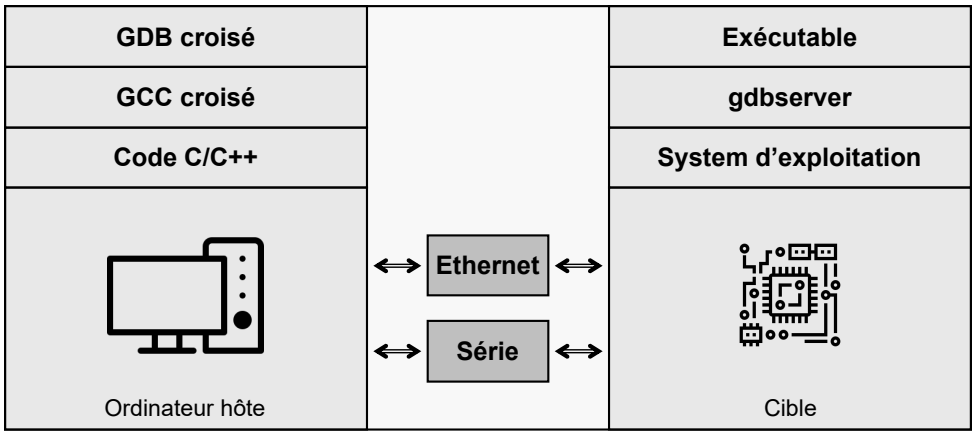

FIGURE 3.10 – Débogage croisé

3.13 Firmware

Un firmware, aussi appelé micrologiciel, est la couche de logiciel située entre le matériel et le système d'exploitation, dont le but principal est d'initialiser et d'abstraire suffisamment de matériel pour que le système d'exploitation et ses pilotes puissent configurer davantage le matériel pour qu'il atteigne toutes ses fonctionnalités. Pour rendre les systèmes embarqués plus robustes et plus efficaces et pour éviter la redondance, il existe une relation de coopération entre le firmware et le système d'exploitation, le firmware étant utilisé par le pilote [110]. Le firmware est souvent considéré comme un élément appartenant au matériel plutôt qu'au logiciel, car il gère et contrôle un composant matériel. Toutefois, en ce qui concerne la programmation, les outils et la méthodologie, un firmware est manifestement un type de logiciel, même s'il est fortement lié au matériel dans la plupart des cas. ARM a conçu *Common Microcontroller Software Interface Standard* (CMSIS) qui permet à ses fournisseurs d'utiliser une infrastructure logicielle uniforme pour développer des solutions logicielles pour les puces basées sur ARM. Avec un écosystème aussi grand, une certaine forme de normalisation du fonctionnement de l'infrastructure du firmware est nécessaire pour assurer la compatibilité logicielle avec divers outils de développement et entre différentes solutions logicielles. En outre, les systèmes embarqués deviennent de plus en plus complexes, et les efforts de développement et de test des logiciels ont considérablement augmenté. Afin de réduire le temps de développement ainsi que le risque d'avoir des défauts dans les produits, la réutilisation des logiciels devient de plus en plus fréquente.

La figure 3.11 illustre les différentes couches de firmware pour un système basé sur ARM Cortex-M, commençant par le niveau matériel dans lequel se trouve le cœur ARM et ses périphériques, juste à coté des périphériques spécifiques à l'implémentation comme le sémaphore matériel. Le deuxième niveau est le firmware où se trouvent les différentes bibliothèques ARM-CMSIS et le *Hardware abstraction layer* (HAL) qui sert à contrôler les autres périphériques. Il y a au dessus le système d'exploitation et le middleware juste avant le niveau du code de l'application utilisateur.

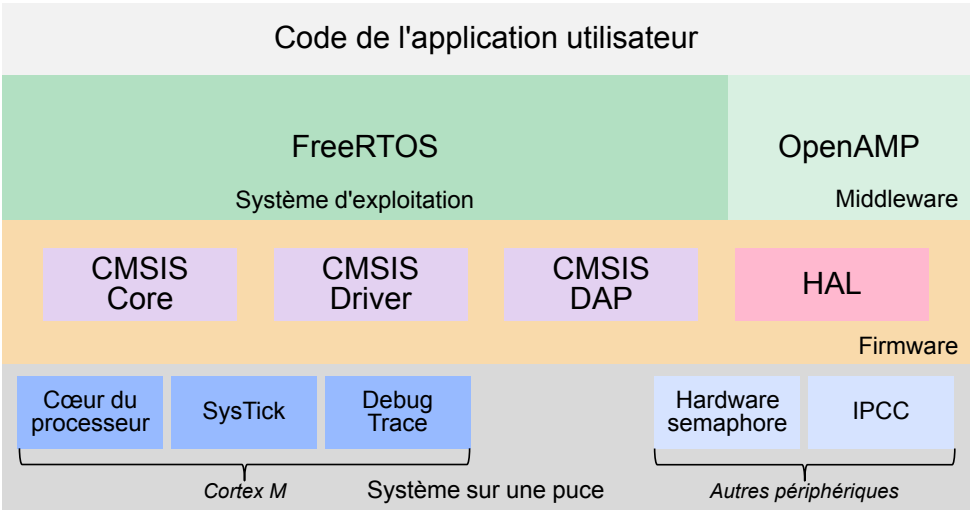

FIGURE 3.11 – Aperçu de la structure du firmware

3.13.1 CMSIS

ARM a collaboré avec plusieurs fabricants de puces, vendeurs d'outils et fournisseurs de solutions logicielles pour développer CMSIS, un firmware qui supporte la plupart des processeurs et microcontrôleurs de la famille Cortex. Son objectif est de faciliter la réutilisation du code dans différents projets, réduisant ainsi le temps nécessaire pour commercialiser les produits et diminuant les coûts de vérification, tout en améliorant la compatibilité logicielle, ce qui permet à des logiciels de différentes sources de fonctionner ensemble [51]. CMSIS est un projet qui a évolué avec le temps, dont le projet principal est CMSIS-Core. Il a commencé comme un moyen pour rendre les bibliothèques de pilotes de périphériques destinées aux microcontrôleurs plus uniformes. Il fournit un ensemble d'interfaces au processeur et aux périphériques, au système d'exploitation et aux composants middlewares (ou intergiciels). Le middleware est un élément du programme qui relie le firmware ou le système d'exploitation d'un côté et l'application de l'autre côté. Il gère en particulier les opérations complexes impliquant plusieurs applications. Actuellement, CMSIS comporte plusieurs projets tels que :

- *CMSIS-Core* C'est un ensemble d'API normalisées permettant aux développeurs d'accéder aux fonctionnalités du processeur, quels que soient les périphériques ou la chaîne de compilation utilisés. Il fournit les API nécessaires pour gérer les interruptions et modifier les registres de contrôle du cœur.
- *CMSIS-Driver* C'est une interface générique de pilote de périphérique pour les middlewares. Il connecte les périphériques comme WIFI, Ethernet ou USB, à des middlewares.
- *CMSIS-DAP* DAP (Debug Access Port) est la conception de référence pour un adaptateur de débogage, il supporte la conversion du protocole USB en JTAG/Série. Cela permet de développer des adaptateurs de débogage à bas prix qui fonctionnent pour plusieurs chaînes de compilation.

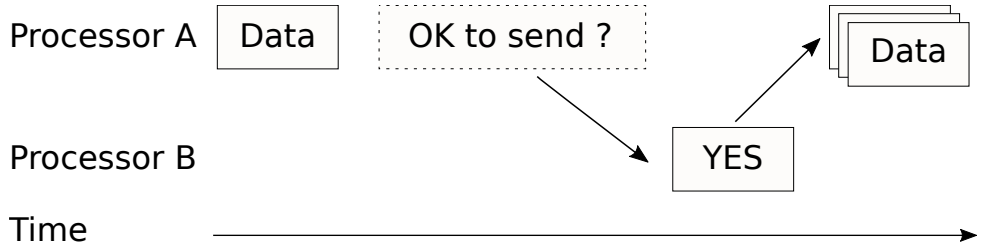

FIGURE 3.12 – Modèle de transmission de messages

3.13.2 Couche d'abstraction matérielle

La couche d'abstraction matérielle, HAL, est une bibliothèque qui fournit une représentation abstraite du matériel physique. Elle contient des routines avec une implémentation spécifique au matériel à la fois pour les pilotes de périphériques et le système d'exploitation. Ces sous-routines permettent aux développeurs de travailler avec différents matériels de la même manière. L'interface de ces routines reste inchangée. De plus, l'interface ne dépend pas du matériel sous-jacent. Par conséquent, les développeurs peuvent ainsi porter leur code source sur de nouvelles plateformes avec peu de modifications.

Par exemple, le pilote Linux du sémaphore matériel, hwspinlock, est basé sur le HAL du sémaphore matériel développé par STMicroelectronics pour le contrôler, mais pour le périphérique SEMA4 de NXP, son propre HAL est utilisé. Toutes les cartes STMicroelectronics qui utilisent le même périphérique matériel utiliseront le même HAL même si le code interne peut être légèrement différent.

3.13.3 Protocole de transmission de messages asymétriques

Un protocole de transmission de messages est utilisé pour assurer la synchronisation et la communication. Il est basé sur une mémoire partagée utilisée pour la communication entre les cœurs (ou clusters de cœurs) hétérogènes. Le contenu des messages peut être tout contenu compréhensible sur les deux processeurs, il respecte généralement un protocole prédéterminé. Les deux opérations requises sont "envoyer" et "recevoir", et sont généralement inspirées du monde Unix, comme défini dans la norme POSIX 1003.1 sur les communications interprocessus IPC. Dans les systèmes embarqués multicœurs, ce protocole assure la synchronisation ainsi que le transfert de données. Comme le montre la figure 3.12, l'expéditeur détermine s'il peut envoyer les données ou non, les données ne sont envoyées qu'après avoir reçu la confirmation de l'autre processeur.

Un autre modèle utilisé par les architectures hétérogènes exige une mémoire partagée accessible par les différents cœurs, ainsi qu'un système d'interruption interprocesseur où chaque côté peut interrompre l'autre. Il étend le modèle existant, pour les systèmes AMP en se basant sur "RPMsg" de Linux qui nécessite un maître Linux.

Open Asymmetric Multi Processing (OpenAMP) est un effort de la communauté afin de standardiser la manière dont les différents environnements embarqués interagissent

entre eux en utilisant AMP [9]. Il fournit des conventions et des normes ainsi qu'une implémentation open source pour faciliter le développement AMP pour les systèmes embarqués. Le but est que, indépendamment de l'environnement ou du système d'exploitation, il soit possible d'utiliser des interfaces identiques pour interagir avec d'autres environnements dans un même SoC. En outre, ces systèmes peuvent interagir grâce à un protocole standardisé, ce qui permet de combiner deux systèmes d'exploitation ou plus dans un même appareil.

OpenAMP [8] est un framework logiciel basé sur les technologies open source existantes dans Linux qui offre trois fonctionnalités aux développeurs AMP :

1. `virtIO` (voir section 3.13.3.3) est un standard de communication virtualisée qui fournit une interface de communication pour les logiciels de couche haute en virtualisant le périphérique esclave.

2. `remoteproc` (voir section 3.13.3.2) est un outil de gestion du cycle de vie des processeurs distants, utilisé pour activer, charger, réinitialiser et redémarrer un cœur ou un cluster de cœurs. Le côté configuré comme maître peut contrôler l'esclave mais pas l'inverse.

3. `RPMsg` (voir section 3.13.3.4) est un outil de messagerie inter-processeur asymétrique. Il fournit des canaux de communication pour les systèmes multicœurs hétérogènes.

AMP est supporté dans de nombreux environnements comme par exemple l'espace utilisateur Linux, l'espace noyau, baremetal (sans système d'exploitation) sur des hyperviseurs et sur de nombreux RTOS comme FreeRTOS et VxWorks.

RPMsg, remoteproc et virtIO font partie du noyau Linux. Cela permet aux applications Linux de gérer des processeurs distants en utilisant remoteproc et de communiquer avec eux en utilisant RPMsg. OpenAMP étend cela et ajoute les mêmes fonctionnalités à d'autres systèmes d'exploitation ou à bare-metal, grâce à lui, il sera possible d'utiliser ces fonctionnalités sur un RTOS, un firmware bare-metal et Linux, où le cœur maître ou le cœur esclave peut être Linux ou tout autre système utilisant OpenAMP, comme le montre la figure suivante 3.13

3.13.3.1 Contrôle du cœur distant à l'aide d'OpenAMP

Tout d'abord, le cœur maître démarre et utilise remoteproc pour charger le binaire exécutable, qui alloue les ressources comme la *Phase-Locked Loop* (PLL), la mémoire et les interruptions, puis il déclenche le démarrage du processeur distant. Ce dernier notifie alors le maître lorsque le démarrage et l'initialisation sont achevés. Les deux cœurs s'exécutent et communiquent, et à la fin, le maître envoie une demande d'arrêt au système distant qui désinitialise ses ressources et arrête l'exécution, le maître désinitialise alors ses ressources et s'arrête, ce qui entraîne l'arrêt de tout le système.

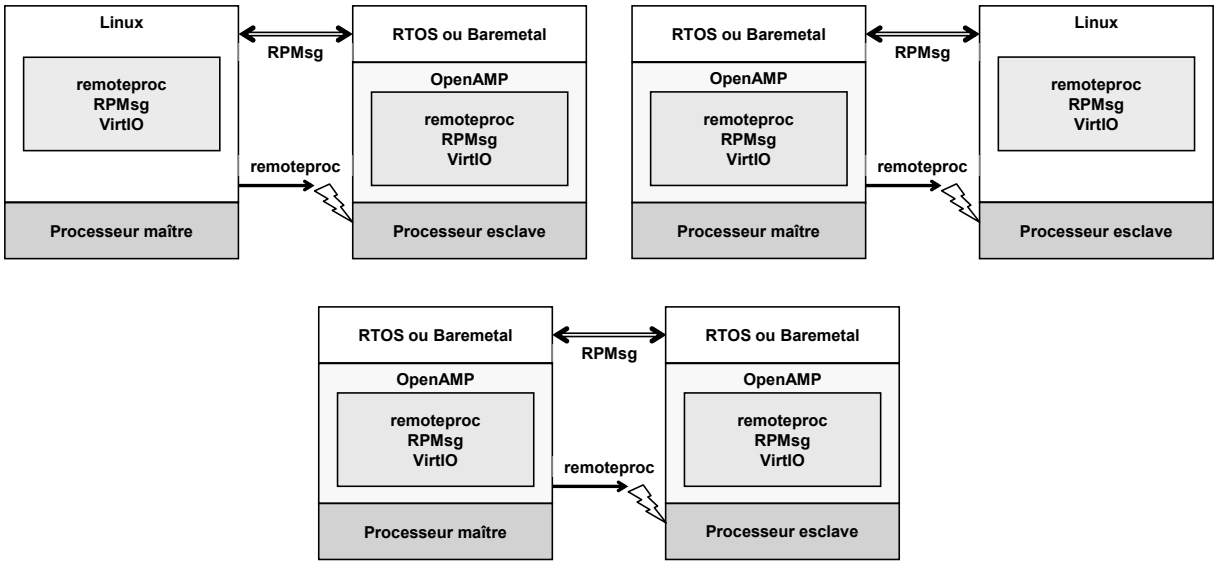

FIGURE 3.13 – Ensemble de configurations d'OpenAMP

3.13.3.2 Remoteproc

Remoteproc est un outil de gestion du cycle de vie des processeurs distants, indépendant de la plateforme. Il supporte les opérations suivantes sur le processeur distant :

- `Démarrer`
- `Charger l'exécutable`
- `Éteindre`
- `Redémarrer`

Il établit également des canaux de communication RPMsg pour les communications en cours d'exécution avec le cœur distant, et il fournit un service de supervision et de débogage du logiciel distant.

3.13.3.3 VirtIO

VirtIO ou Virtual Input-Output, est un ensemble des API définies dans le noyau Linux qui sert à créer des périphériques d'entrées et sorties virtuelles. Virtio dans OpenAMP est constitué de trois composants :

- *"VirtIO Device"* est une abstraction qui permet à un pilote d'instancier sa propre instance d'un périphérique virtio, ce qui permet d'accéder à toutes ses fonctionnalités.
- *"virtqueue"* est une API qui permet aux pilotes de transmettre et de recevoir des données en utilisant l'infrastructure vring de virtqueue. Il est formé principalement d'une file d'attente où les données sont envoyées d'un côté et consommées de l'autre.
- *"Vring"* est une structure de données qui gère les données envoyées et reçues. Il comporte un mécanisme de notification pour connaître la disponibilité des données.

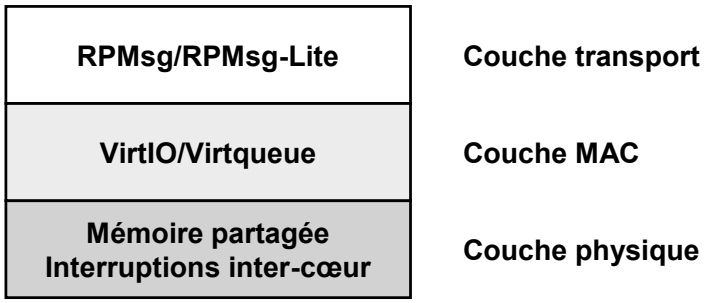

FIGURE 3.14 – Couches du protocole RPMsg

3.13.3.4 RPMsg

Le framework RPMsg est un bus de communication basé sur Virtio qui permet à un processeur local de communiquer avec les processeurs distants disponibles sur le système. Il permet à deux cœurs hétérogènes asymétriques de communiquer en utilisant une mémoire partagée basée sur des buffers circulaires à lecture et écriture unique pour transférer des messages entre les cœurs. Ce protocole ne nécessite aucun élément de synchronisation multicœur tel que le spinlock. La figure 3.14 représente la hiérarchie de ce protocole et son alignement avec le modèle ISO/OSI.

Il étend le VirtIO/virtqueue en ajoutant l'en-tête RPMsg, et comme les composants vrings sont unidirectionnels, sur un système AMP, deux vrings sont nécessaires, l'un est dédié aux messages envoyés au processeur distant, et l'autre est utilisé pour les messages reçus du processeur distant. En outre, des mémoires partagées sont créées dans l'espace mémoire accessible par les deux processeurs asymétriques. La communication est basée sur des canaux. Les canaux sont identifiés par leurs noms, nous pouvons par exemple créer deux canaux, l'adresse RPMsg locale (”source”) et l'adresse RPMsg distante (”destination”). RPMsg est conçu pour permettre uniquement la communication cœur à cœur. Si la communication cœur à multicœur est nécessaire, plusieurs instances de RPMsg doivent être adoptées. Chaque instance agit alors indépendamment des autres, sans avoir besoin de régler les problèmes dûs à la synchronisation multicœur.

Ce protocole est en train de devenir un standard dans la communication des systèmes hétérogènes multicœurs grâce à sa présence dans le noyau Linux [1]. Les grandes entreprises de semi-conducteurs telles que NXP, STMicroelectronics, Qualcomm ou Xilinx adoptent ce protocole pour leurs plateformes.

RPMsg-Lite est une autre méthode de communication entre plusieurs cœurs dans un environnement hétérogène. C'est une implémentation plus légère de RPMsg qui offre une réduction de la taille du code, une simplification de l'API et une plus grande modularité [83].

3.14 Conclusion

Ce chapitre nous a permis de faire l'état des lieux de l'architecture des processeurs. Plus spécifiquement l'architecture hétérogène utilisée dans une application temps réel embarquée et les composants clés retrouvés dans un SoC qui peuvent avoir un effet sur le déterminisme de cette application. Le pipeline, la mémoire cache et la mémoire virtuelle sont des technologies ayant un impact significatif sur le déterminisme. De plus, les différentes formes d'interruptions et leurs effets sur l'exécution du code. Tous ces éléments reviendront par la suite et devront être pris en considération lors de la mesure de temps d'exécution. Il existe des périphériques spécifiques pour architecture asymétrique ayant un rôle principal de gestion de la communication et la synchronisation entre cœurs hétérogènes qui seront mesurés dans le chapitre 5. Finalement, il faut souligner que l'outil développé dans le cadre de ce travail de recherche se repose en partie sur les outils de débogage présentés dans ce chapitre.

Chapitre 4

Mesure de durée d'exécution

Sommaire

4.1 Introduction 87
4.2 Pire durée d'exécution 87
4.3 Méthodes d'analyse de WCET 88
4.3.1 Analyse statique 89
4.3.2 Analyse dynamique 91
4.3.3 Analyse hybride basée sur les mesures 92
4.3.4 Analyse probabiliste basée sur les mesures 93
4.3.5 Problèmes d'analyse du temps d'exécution sur les architectures modernes 93
4.3.6 Apport de la mesure à l'estimation de WCET 96
4.3.7 Les marges de sécurité 96
4.4 Caractéristiques des techniques de mesure 97
4.5 Méthodes de mesure sur un cœur 98
4.5.1 Chronomètre 98
4.5.2 Les outils de mesure du temps sous Linux 99
4.5.3 Fonctions standard C 101
4.5.4 Compteur de cycles d'horloge 101
4.5.5 Timer/Compteur sur puce 102
4.5.6 Analyseur logique 103
4.6 Comparaison expérimentale 104
4.7 Mesure expérimentale de durée d'exécution sur un système multicœur SMP 106
4.8 Contribution industrielle 108
4.9 Conclusion 108

4.1 Introduction

Plusieurs types de programmes doivent respecter des contraintes de temps, notamment dans le domaine des systèmes temps réel. La première phase nécessaire à la vérification des contraintes est la détermination de la durée d'exécution d'une portion de code prise en isolation sur la plateforme d'exécution cible. Les portions de code composant un programme temps réel sont donc caractérisées, au minimum, par leur WCET [118] [90, 87]. Il existe différents types d'analyse de WCET : statique, dynamique, hybride et probabiliste. Ce chapitre a pour objectif de les présenter en soulignant leurs avantages et leurs inconvénients. Ceci permet d'illustrer dans quel contexte les méthodes de mesures proposées dans cette thèse seront amenées à être utilisées. Par la suite, différentes méthodes de mesure de temps d'exécution sont étudiées et comparées par rapport à la disponibilité de la méthode dans le système, la simplicité de mise en place et la précision requise selon les contraintes temporelles de l'application. Ces techniques de mesure reviendront dans le chapitre 6 où elles sont utilisées dans la nouvelle fonctionnalité de l'outil développé qui sert à simplifier les mesures de temps d'exécution sur n'importe quel environnement Eclipse.

4.2 Pire durée d'exécution

Afin de démontrer le respect de contraintes temporelles par les systèmes temps réel, une analyse d'ordonnançabilité doit être effectuée. Cependant, l'analyse de l'ordonnançabilité ne peut être effectuée que lorsque le WCET de chaque tâche est connu. Le problème de l'estimation du WCET a été étudié pendant plusieurs décennies, et de nombreux outils ont été développés à cette fin, comme l'analyseur WCET aiT d'AbsInt [104] qui a été utilisé pour analyser le logiciel de commande de vol de l'avion A380.

Il est très difficile, voire impossible, de déterminer un WCET exact, à cause d'une part de la complexité croissante des processeurs, et le nombre très important d'états internes qu'il est possible d'atteindre, la durée d'une instruction dépendant de l'état interne du processeur, des caches, des bus, et de la mémoire ; et d'autre part des autres traitements parallèles (liés à la préemption, ou bien à l'exécution sur d'autres cœurs partageant des ressources (cache, bus, mémoire, etc.) avec le cœur exécutant le code. Par conséquent, l'estimation du WCET utilisée est une borne supérieure du WCET atteignable. Une estimation valide et utile du WCET doit satisfaire les critères de sécurité et de précision :

- l'estimation doit être sûre, c'est-à-dire qu'elle constitue une limite supérieure du WCET réel.
- l'estimation doit être la plus précise possible afin de minimiser sa surestimation, qui pourrait entraîner un sur-dimensionnement des ressources de calcul nécessaires.

La plupart des outils d'analyse du temps se concentrent uniquement sur la détermination de la borne supérieure du WCET d'un programme ou d'un code de fonction qui s'exécute de manière isolée et sans interruption. Plus précisément, ces outils ne tiennent pas compte de toutes les interférences que l'exécution du code analysé peut subir lorsqu'il

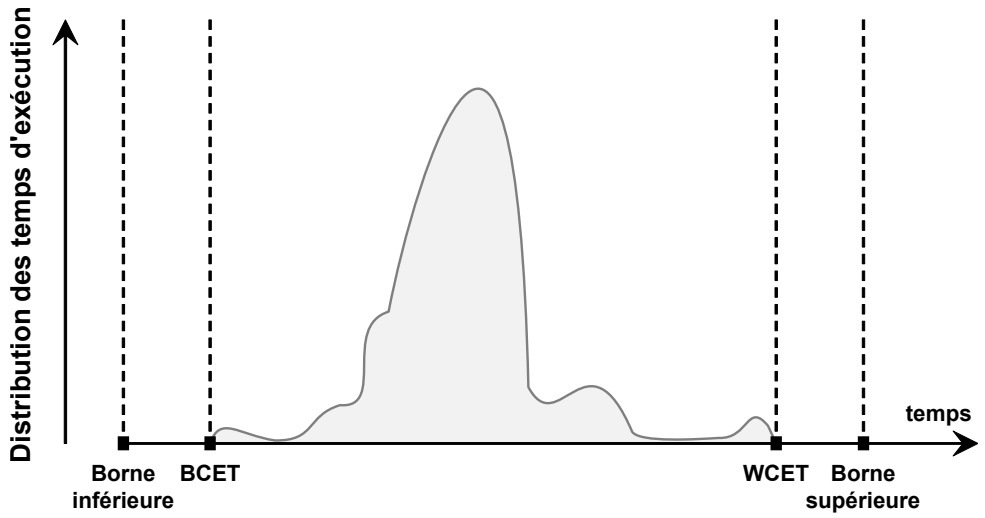

FIGURE 4.1 – Distribution possible du temps d'exécution par rapport au BCET et au WCET

s'exécute simultanément avec d'autres tâches ou programmes sur le même matériel. En général, ils ignorent toutes les interférences d'exécution dues aux conflits d'accès aux ressources logicielles partagées (comme les données partagées entre tâches) et les ressources matérielles partagées (comme un périphérique partagé). Les interférences du système d'exploitation qui réordonnancent régulièrement les tâches et qui interrompent les programmes sont également ignorées par ces analyseurs. Toutes ces interactions entre la tâche analysée, le système d'exploitation et toutes les autres tâches en cours d'exécution dans le système sont évaluées séparément et parfois elles sont intégrées dans une analyse d'ordonnançabilité de plus haut niveau. Pour que les exigences de synchronisation soient respectées, il n'est ni acceptable ni réaliste d'ignorer ces sources de conflit et d'interférence au niveau de l'analyse de l'ordonnançabilité.

La figure 4.1 montre une distribution possible du temps d'exécution d'un bloc de code spécifique en mode séquentiel, sans aucune interférence provoquée par les interruptions ou le système d'exploitation. Le "Best Case Execution Time" (BCET) représente le meilleur temps d'exécution et le WCET le pire. Les bornes ajoutées sont essentielles pour prendre en compte les interférences.

4.3 Méthodes d'analyse de WCET

Il existe de nombreuses techniques d'analyse de WCET : l'approche statique, dynamique et hybride. Alors que l'approche dynamique, basée sur des mesures sur une cible ou un simulateur, ne peut pas être utilisée pour les applications critiques au niveau de sûreté, l'approche statique considère toutes les exécutions possibles et peut fournir les bornes du temps d'exécution plus sûres. Comme le montre la figure 4.2, la méthode basée sur les mesures peut ne pas détecter tous les temps d'exécution possibles. L'approche hybride combine la méthode dynamique et statique, mais elle souffre toujours des inconvénients de la dynamique. Les trois méthodes sont généralement reconnues comme étant de même

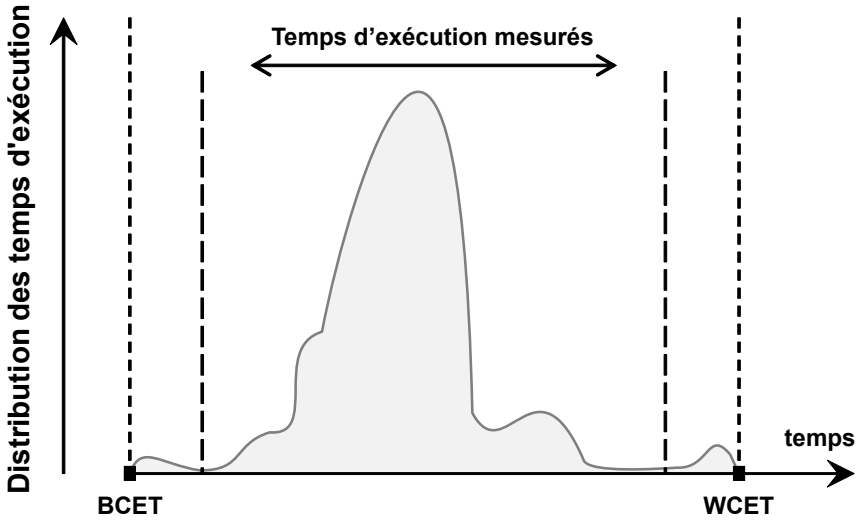

FIGURE 4.2 – Temps d'exécution mesurés par rapport au BCET et au WCET

importance et efficacité, car elles visent différents types d'applications. D'autres méthodes, comme la méthode probabiliste, étendent les solutions statiques et dynamiques, dans le but de rendre explicite la rareté d'occurrence de certaines durées d'exécution très longues par rapport à la vaste majorité des cas.

4.3.1 Analyse statique

Cette méthode est basée sur des modèles mathématiques qui dépendent à la fois du logiciel et du matériel. Elle comporte généralement deux phases : l'analyse du flux qui détermine tous les chemins possibles dans le graphe du flot de contrôle du programme, et l'analyse temporelle qui évalue le temps d'exécution de chacun de ces chemins. Le modèle matériel permet de déterminer le temps d'exécution des instructions individuellement. Le modèle logiciel représente les flux d'exécution possibles. La combinaison de ces modèles avec des informations sur le nombre maximal d'itérations des boucles, les chemins possibles dans le programme, la fréquence d'exécution des parties du code, la modélisation des accès aux caches, etc. permet d'obtenir une borne supérieure du WCET. Tant que les modèles sont corrects, l'estimation du WCET est fiable, elle est, en général, supérieure à tout temps d'exécution observable.

La méthode statique repose sur un modèle précis du comportement temporel du matériel cible, y compris la modélisation des caractéristiques telles que les pipelines, les caches, la mémoire et les bus ; l'ensemble affecte la durée de la tâche en cours d'exécution [13]. Le but est de déterminer les bornes sans exécuter réellement le programme sur la cible, tout en prenant en compte les changements d'état du matériel sous-jacent [72]. Ces changements peuvent impliquer l'effacement d'une ligne de cache, le vidage complet d'un pipeline, etc.

Cette méthode consiste à calculer le chemin d'exécution le plus défavorable en construisant un graphique de flot de contrôle, ou *Control Flow Graph* (CFG), à partir d'un programme donné et en considérant tous les chemins du graphique. Fournir des limites de

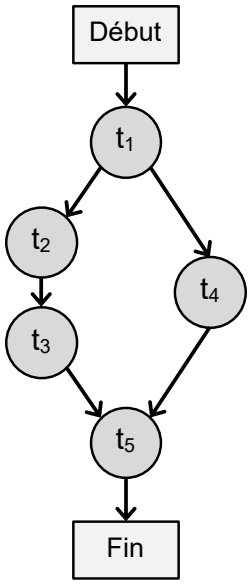

FIGURE 4.3 – Graphe de flot de contrôle

boucle et d'autres annotations à l'outil d'analyse peut l'aider à simplifier l'analyse. Le CFG présenté sur la figure 4.3, est un graphe connecté composé de trois éléments : les nœuds, les extrémités et les points d'entrée. Généralement, un nœud d'un CFG est un ensemble d'instructions qui s'exécutent séquentiellement. Après l'exécution de la première instruction, les autres s'exécuteront toujours l'une après l'autre, sauf en cas d'erreur ou d'interruption. Ils sont identifiés par leur instruction de départ. L'exécution commence par l'instruction de départ, qui peut atteindre n'importe quel autre nœud du graphe. Tous les chemins à travers le CFG commencent par cette instruction et se terminent par un seul nœud final.

Le système d'analyse de temps permet non seulement d'analyser la structure principale du programme, mais aussi d'autres modules tels les caches et les pipelines afin d'obtenir des estimations tenant compte des spécificités de la cible. Aujourd'hui, des outils statiques de WCET sont disponibles sur le marché, notamment aiT [30] et Bound-T [42]. Il existe également plusieurs prototypes dans le domaine de la recherche, notamment Chronos [67], Heptane [38], SWEET [70] et OTAWA [111].

Les bornes obtenues par les approches statiques dépendent beaucoup du modèle abstrait de la cible. Auparavant, le marché des systèmes embarqués était dominé par des processeurs simples et prédictibles, faciles à modéliser et permettant de déterminer des bornes sûres et précises. Mais en raison des besoins de calcul accrus des systèmes embarqués modernes, les concepteurs se sont tournés vers des processeurs plus complexes qui sont principalement conçus pour la performance et non pour la prédictibilité. Dans un tel cas, toutes les difficultés contribuant à une telle imprédictibilité doivent être capturées par le modèle abstrait afin de fournir des bornes acceptables. La modélisation du matériel dépend des fabricants de puces qui doivent publier les détails de leur fonctionnement interne, ce qui n'est généralement pas le cas pour différentes raisons comme la propriété intellectuelle.

Les modèles doivent donc être vérifiés pour s'assurer qu'ils reflètent bien le matériel cible. Le fait de ne pas saisir les caractéristiques de performance peut entraîner une surestimation du temps d'exécution. Capturer tous les états du système dans une machine complexe peut aboutir à des temps d'analyse inacceptables. De plus, la construction et la vérification du modèle de synchronisation pour chaque modèle de processeur coûte cher, prend du temps et provoque des erreurs. Cela se reflète dans le coût élevé des outils commerciaux d'analyse statique et dans le délai parfois important entre sortie d'un nouveau calculateur et sa modélisation.

Si le CPU n'est qu'un simple microcontrôleur (monocœur sans les fonctions d'amélioration des performances, comme les caches, les pipelines et les prédicteurs de branchement), l'analyse WCET devient beaucoup plus facile. Pourtant, lorsque les besoins de calcul augmentent, de nombreuses fonctions sont ajoutées au processeur pour améliorer ses performances, ce qui rend en même temps le comportement temporel très difficile à prévoir. Ainsi, les approches d'estimation du WCET dépendent largement de la micro-architecture du processeur [39].

4.3.2 Analyse dynamique

La méthode classique et la plus courante dans l'industrie pour estimer le WCET d'un programme consiste à effectuer des mesures. Selon le principe qu'applique cette méthode, le processeur est le meilleur modèle matériel. Le programme est exécuté plusieurs fois sur le matériel réel, avec des entrées différentes et de manière isolée, et le temps d'exécution est mesuré pour chaque exécution d'une manière intensive en instrumentant le code source à différents niveaux. Chaque exécution de mesure n'exerce qu'un seul chemin d'exécution tout au long du programme, et donc pour le même ensemble de valeurs d'entrée, plusieurs milliers d'exécutions du programme doivent être effectuées pour capturer les variations du temps d'exécution dues à la fluctuation des états du système. Pour ces approches basées sur la mesure, le principal défi consiste essentiellement à identifier l'ensemble des valeurs d'entrée qui conduisent à son WCET.

Cette méthode présente de nombreux avantages, car les mesures sont souvent directement disponibles pour le développeur, et elles sont utiles principalement lorsque la valeur moyenne ou approximative de la durée d'exécution est recherchée. De plus, la plupart des mesures ont l'avantage d'être effectuées sur le matériel réel, ce qui évite de devoir construire un modèle matériel.

En revanche, les mesures exigent que le matériel soit disponible, ce qui peut ne pas être le cas pour les systèmes pour lesquels le matériel et le logiciel sont développés en parallèle. En outre, la taille réelle du code pour effectuer ces mesures est plus importante en raison du code d'instrumentation intrusif ajouté ainsi que les mesures elles-mêmes ajoutent au temps d'exécution du programme analysé. Ce dernier inconvénient peut être réduit en utilisant des outils de mesure matériels peu ou pas intrusifs, ou en laissant simplement le code de mesure ajouté, et donc le temps d'exécution supplémentaire, dans le programme

final (ce qui, dans certains cas, n'est pas optimal).

En outre, pour la plupart des programmes, le nombre de chemins d'exécution possibles est tellement grand qu'il est impossible d'effectuer des tests exhaustifs et que les mesures ne sont effectuées que pour un certain nombre de valeurs d'entrée possibles. Cependant, dans de nombreux cas, les temps mesurés sous-estiment le WCET, en particulier lorsque des logiciels et/ou des matériels complexes sont analysés. C'est pourquoi il est courant d'ajouter une marge de sécurité au pire cas de temps mesuré, en espérant que le WCET réel soit inférieur à l'estimation du WCET qui en résulte. La question principale est de savoir si la marge de sécurité supplémentaire fournit une limite sûre, puisqu'elle est basée sur des estimations. Une marge très élevée entraînera un surdimensionnement des ressources, conduisant à une utilisation plus faible, tandis qu'une petite marge peut donner lieu à un système non sûr.

4.3.3 Analyse hybride basée sur les mesures

Cette méthode a été présentée pour la première fois par Kirner et al [58]. C'est une technique standard de l'industrie, déployée dans des outils commerciaux, qui fonctionne de manière similaire à l'analyse statique, sauf qu'elle ne repose pas sur un modèle matériel.

L'approche hybride, comme son nom l'indique, associe les avantages de l'analyse statique avec ceux de l'approche basée sur les mesures de temps d'exécution. Elle utilise des mesures pour extraire la durée de petites sections de code, et l'analyse statique pour déduire l'estimation du WCET. L'approche identifie certains chemins de flux en utilisant l'analyse statique et le temps d'exécution de ces chemins est mesuré sur du matériel réel ou par des simulateurs précis.

Le code source est instrumenté avec des points d'instrumentation qui indiquent qu'une section spécifique du code a été exécutée. L'application est ensuite exécutée sur la plateforme matérielle cible ou sur le simulateur pour collecter les traces d'exécution. Ces traces sont une séquence de valeurs temporelles qui montrent quelles parties de l'application ont été exécutées et à quel instant. Enfin, les outils hybrides produisent des mesures de performance pour chaque partie du code exécuté et, en utilisant ces données et la structure du code, ils estiment le WCET.

Enfin, les informations sur le flux d'exécution sont combinées avec les techniques de l'approche statique pour déterminer le chemin le plus long. L'avantage de l'approche hybride est qu'elle ne repose pas sur des modèles abstraits complexes de l'architecture matérielle. Bien que l'estimation du WCET soit généralement plus précise que celle calculée par l'analyse statique, il existe une possibilité de sous-estimation si les tests n'ont pas été suffisamment effectués sur les petites parties du programme. Toutefois, l'incertitude quant à la couverture du comportement le plus défavorable par la mesure existe toujours, car il n'est pas possible de supposer un état initial sûr et une entrée la plus défavorable dans tous les cas. De plus, un code instrumenté est nécessaire, ce qui peut ne pas être autorisé dans une certification particulière. Parmi les exemples d'outils, on trouve Rapitime

et MTime [78].

4.3.4 Analyse probabiliste basée sur les mesures

Avec les architectures matérielles actuelles, le temps d'exécution d'une application donnée dépend des états des composants matériels, et ces états dépendent à leur tour de ce qui a été exécuté auparavant. Parmi les exemples typiques de cette relation entre l'application et l'architecture matérielle sous-jacente, l'écart de temps d'exécution observé lorsqu'un programme s'exécute sur un processeur équipé d'un système de cache. Lors de la première exécution du programme, chaque demande de lecture d'instructions et de données entraîne un défaut de cache et doit alors être chargées à partir de la mémoire principale. Lors de la deuxième exécution, ces informations se trouvent déjà dans le cache et n'ont pas besoin d'être chargées à nouveau, ce qui se traduit par un temps d'exécution considérablement plus court que celui de la première exécution. En raison de cette dépendance aux événements précédents, l'ensemble du temps d'exécution mesuré du même programme ne peut être considéré comme un ensemble de variables aléatoires et la plupart des outils statistiques ne peuvent être appliqués pour analyser les traces d'exécution collectées.

L'objectif des techniques probabilistes basées sur la mesure est de faire une estimation statistique et de ne plus dépendre des événements précédents, afin de pouvoir tester le temps d'exécution d'une application et d'en déduire des estimations probabilistes qui s'appliquent à son comportement global, en toutes circonstances et dans toutes les situations.

Les approches probabilistes ont comme objectif d'appliquer les résultats de la théorie des valeurs extrêmes au problème de l'estimation du WCET [99]. Les solutions basées sur la théorie des valeurs extrêmes consistent d'abord à mesurer le temps d'exécution d'une application en l'exécutant en utilisant plusieurs entrées. Ensuite, ces solutions basées sur la théorie des valeurs extrêmes répartissent les valeurs en plusieurs intervalles, analysent la distribution des valeurs maximales locales dans ces intervalles, puis estiment de combien le temps d'exécution peut s'écarter de la moyenne de cette "distribution des extrêmes". Les techniques probabilistes basées sur les mesures ont fait l'objet d'efforts de recherche ces dernières années, et des innovations dans ce domaine ont été réalisées notamment dans le cadre des projets européens PROARTIS [21] et PROXIMA [20].

4.3.5 Problèmes d'analyse du temps d'exécution sur les architectures modernes

L'analyse statique sur les plateformes modernes comme les HMPSoC est confrontée à des problèmes importants : la différence de plus en plus importante, notamment sur puce multiprocesseur entre durée d'exécution observée et durée d'exécution pire cas, ainsi que l'absence du modèle des nouvelles plateformes. En effet, le problème pour les supporter est que cela demanderait une quantité de travail très importante. En outre, si cela

devait être fait, les interférences possibles aux différents niveaux (bus mémoire, mémoire externe, mémoire cache, composants heuristiques, etc.) conduiraient à une très forte surestimation du WCET due aux nombreux comportements possibles du programme et de la plateforme d'exécution. Par ailleurs, les architectures modernes ont tendance à optimiser le temps d'exécution moyen au détriment du déterminisme. L'autre façon de calculer le temps d'exécution moyen ou le pire cas sur une plateforme est appelée *dynamique*, et consiste à mesurer le temps d'exécution réel du programme analysé sur la plateforme cible. Cette technique peut être appliquée sur une nouvelle plateforme, à l'aide d'environnements de développement adaptés, comme celui que nous présenterons dans cette thèse. Cette technique n'est pas sûre, car il est possible d'observer, à un certain moment pendant l'exécution, un temps d'exécution qui pourrait être plus grand que celui mesuré pendant l'analyse dynamique du temps. Cependant, pour les systèmes ayant des contraintes temporelles moins strictes, appelés systèmes temps réel *mous*, une mesure dynamique intensive du temps d'exécution pourrait être suffisante, et souvent être le seul choix applicable. En effet, le temps d'exécution d'une instruction peut varier en fonction de l'état interne du processeur. Cet état devient de plus en plus complexe avec la présence de composants heuristiques. Par conséquent, il devient de plus en plus difficile de prédire les parties pertinentes de l'état interne du processeur qui influencent le comportement du temps d'exécution. L'exécution d'un bout de code dépend non seulement de la valeur de ses données d'entrée (par exemple, produire des branchements différents dans les tests) mais aussi pour avoir un comportement identique, avec les mêmes conditions de branchement, dépend de l'état interne du CPU et de la mémoire, notamment en ce qui concerne l'optimisation locale du temps d'exécution moyen. Ces techniques d'optimisation impliquent des variations du temps d'exécution, et peuvent même aggraver le temps d'exécution du pire cas. En général, un processeur traite chaque instruction en plusieurs étapes : chargement, décodage, exécution, écriture, etc. Pour une même instruction, une seule de ces étapes est utilisée en même temps. En pratique, les étapes disponibles sont utilisées en parallèle pour différentes instructions.

Cela conduit à un chevauchement dans l'exécution des instructions, connu sous le nom de *pipelining* détaillé dans la section 3.2. Comme il peut y avoir des dépendances entre les instructions (par exemple, une valeur calculée est nécessaire pour une instruction suivante), il peut y avoir une interaction entre les instructions qui sont exécutées l'une après l'autre.

La plupart des processeurs à haute fréquence utilisent une mémoire cache pour accélérer le temps moyen d'accès à la mémoire en stockant les valeurs lues dans la mémoire cache. La localité de la référence implique qu'une instruction ou une donnée accédée a de fortes chances d'être accédée à nouveau, et que le prochain accès à une valeur mise en cache ne nécessitera pas d'accéder à la mémoire, ce qui réduit le temps d'exécution. Toutefois, la préemption altère le contenu du cache et crée des délais de préemption liés au cache. Ceci est un problème qui a reçu beaucoup d'attention dans l'analyse statique [4] [115].

De plus, afin de maintenir le pipeline rempli lors des branchements dans un programme,

les processeurs implémentent une prédiction de branche, dont le but est de prédire ce qui sera exécuté après une branche conditionnelle lorsque la condition n'est pas encore connue. Ces processeurs récupèrent alors de manière spéculative les instructions de la mémoire, qui sont ignorées si la prédiction s'avère être fausse. Cela influence souvent le comportement du cache de manière très complexe.

Sur les architectures multicœurs, qui partagent le même bus mémoire, ou sur les architectures manycores [33] qui partagent les mêmes adresses mémoires, l'exécution sur un cœur peut interférer avec l'exécution sur un autre cœur en raison de la contention des accès au bus mémoire et aux adresses.

En outre, les programmes peuvent nécessiter l'utilisation de circuits externes, qu'il s'agisse d'unités de calcul spécialisées externes (par exemple, unité de calcul en virgule flottante - FPU, unité de traitement graphique - GPU, circuits intégrés spécifiques à une application - ASIC, etc.), qui fonctionnent de manière asynchrone par rapport aux cœurs de calcul et peuvent également être partagés entre plusieurs cœurs (et donc entre plusieurs blocs de code en cours d'exécution). Par exemple, si le système doit accéder à un périphérique par des adresses mappées en mémoire, alors le calcul du WCET doit prendre en compte à la fois le temps pris par le bus de communication pour accéder au périphérique et le temps de réponse du dispositif. C'est également le cas pour la mémoire. Le temps de réponse du périphérique, de la mémoire ou du bus dépend de leurs taux d'occupation respectifs et de leur politique d'arbitrage, qui est souvent peu documentée.

Sur les architectures matérielles complexes, il est donc extrêmement complexe de déterminer le WCET réel d'un système. En effet, il y a de nombreux paramètres à prendre en compte. En outre, la surestimation par rapport aux cas réels observables est très importante ce qui peut amener à sous-exploiter très largement la plateforme. De plus, l'évolution des technologies depuis trois décennies a toujours privilégié le temps de calcul moyen, au détriment du temps d'exécution le plus défavorable. Par exemple, si le WCET d'une plateforme matérielle moderne surestime dix fois le WCET par rapport au WCET réel observable, quel serait l'intérêt d'utiliser cette plateforme par rapport à une plateforme plus ancienne, plus lente et plus simple où le WCET calculé serait plus proche du WCET réel observable ? Nous ne pensons pas qu'il soit acceptable d'utiliser une plateforme à 5 ou 10 % de sa capacité. C'est pourquoi nous nous concentrons sur des outils de mesure dynamiques sur des plateformes modernes. L'un des effets recherchés est de trouver la valeur la plus proche possible du pire temps d'exécution observable.

Ce travail de recherche porte sur les méthodes de mesure du temps d'exécution sur les plateformes informatiques modernes. Le déploiement d'un programme, ainsi que de son système d'exploitation hôte, de sa configuration et de ses options, nécessite beaucoup de temps. De plus, plusieurs horloges, timers, et interfaces de programmation d'applications (API), permettent théoriquement à un développeur de mesurer le temps d'exécution d'un bloc de code. Dans cette recherche, nous discutons ces méthodes, afin qu'elles puissent être utilisées dans une méthode d'analyse de temps plus large pour obtenir un temps d'exécution de pire cas. Ceci peut être utilisé dans les familles de méthodes d'analyse du

temps qui peuvent être identifiées.

4.3.6 Apport de la mesure à l'estimation de WCET

En principe, il est toujours possible d'extraire des modèles temporels sûrs et fiables et de définir des abstractions mathématiques pour étudier le comportement d'un système déterministe. Cependant, il est pratiquement difficile de définir et d'utiliser des modèles mathématiques statiques pour toutes les plateformes, principalement en raison de la complexité du système, surtout pour les HMPSoC. La complexité accrue du système entraîne un temps de modélisation plus long. Le développement d'un modèle d'une plateforme peut prendre jusqu'à plusieurs années pour atteindre le niveau de précision souhaité puis être validé. A cela s'ajoutent de nombreux autres inconvénients comme une moindre portabilité. Tous ces éléments conduisent à offrir, à défaut de mieux en terme de sûreté, une technique basée sur la mesure.

Toutes les techniques basées sur la mesure estiment un WCET à partir de valeurs pour lesquelles la distance par rapport au pire cas réel est inconnue. Le WCET réel est inconnu et il est très probable qu'il ne sera pas observé pendant les tests. Pire encore, en l'absence de modèle statique sûr, il n'est même pas possible de savoir si le pire cas a été observé ou non. En bref, cela signifie qu'il n'y a aucune garantie qu'une telle approche puisse prévoir la valeur exacte du WCET. Une conséquence directe est que, bien que ces techniques fassent des prédictions basées sur des calculs avancés et approfondis, elles ne peuvent jamais garantir que leurs estimations sont "sûres". Cela peut être problématique pour les applications nécessitant des garanties temps réels strictes, typiquement dans les systèmes temps réel durs.

Cependant, on peut noter que dans de nombreux domaines, les garanties certifiables ne sont pas nécessaires. Par exemple, de nombreuses applications ne nécessitent que des estimations fiables, dans le sens où l'on doit pouvoir se baser sur ces valeurs et évaluer le risque qu'elles soient fausses. En outre, le fait que cette approche ne soit pas liée à des infrastructures matérielles et des types d'applications spécifiques lui permet de bénéficier d'une plus grande flexibilité et portabilité que les méthodes d'analyse statique, ainsi que de réduire considérablement le temps de modélisation et les coûts du projet.

4.3.7 Les marges de sécurité

En mesurant le temps d'exécution d'une application sur la cible, il est très probable, si pas certain, que la situation où elle prend son temps d'exécution maximal ne se produise pas [17]. Cela est dû au fait que le test n'a pas réussi à identifier l'ensemble d'arguments d'entrée qui conduit au chemin d'exécution le plus long du programme, ou bien il a trouvé le chemin d'exécution qui aboutit au WCET mais il n'a pas généré le maximum d'interférences possibles. Cela signifie que le WCET réel n'est pas observé uniquement parce que les interférences générées pendant les tests n'ont pas placé l'application dans

les pires conditions d'exécution.

Le nombre de chemins d'exécution possibles pour toute application est tellement important qu'il est difficile de réaliser des tests exhaustifs, sous peine de faire face à une explosion combinatoire de cas à tester. Par ailleurs, les mesures ne sont effectuées que pour un certain nombre de valeurs d'entrée. En général, le processus de test commence par l'identification d'un ensemble d'entrées qui sont susceptibles de permettre au programme d'emprunter le chemin d'exécution le plus long à travers son code et de provoquer son WCET. Cette étape est généralement supervisée et basée sur une inspection manuelle du code. Ou bien elle peut être basée sur des outils avancés qui peuvent aider à identifier les pires jeux de paramètres d'entrée.

Il faut toujours supposer que le niveau d'interférence le plus élevé n'est pas observé pendant les essais et que, par conséquent, le temps d'exécution maximal enregistré est une sous-estimation du WCET réel. Pour compenser cela, il est important d'ajouter une marge de sécurité dans l'espoir que le WCET réel soit inférieur à cette estimation majorée.

Il reste à savoir si la marge de sécurité supplémentaire fournit une borne sûre, puisqu'elle est basée sur des estimations non vérifiables. En principe, une marge très élevée donne une borne supérieure qui est probablement sûre, mais résulte en un système surdimensionné avec une faible utilisation de ses ressources, tandis qu'une petite marge peut conduire à une sous-estimation de pire cas réel du système.

Généralement, la valeur de la marge de sécurité appliquée au temps d'exécution maximal mesuré est basée sur une estimation de l'interférence maximale. Ces interférences sont produites par le système d'exploitation ou d'autres applications qui n'ont pas été observées pendant la phase de test, mais que l'application analysée pourrait potentiellement rencontrer au moment de l'exécution. Il est courant, sur un monocœur, d'ajouter simplement une marge de 50%, ou tout autre pourcentage selon les préférences de l'utilisateur et son niveau de confiance dans ces marges [82], au temps d'exécution maximal observé. Mais sur les architectures multicore et manycore, les experts ne sont pas encore en mesure d'estimer de manière sûre des marges fiables.

4.4 Caractéristiques des techniques de mesure

Afin de mesurer une durée d'exécution (par "une" durée, ce qui est entendu est la durée d'une mesure individuelle d'exécution), il existe plusieurs méthodes qui présentent une sélection d'attributs[107]. Il s'agit notamment de :

- *Précision* représente la proximité de la valeur mesurée par rapport à la valeur réelle en utilisant une technique de mesure. Autrement dit, lors de la mesure du temps d'exécution, il existe toujours une marge d'erreur. La valeur réelle est certainement comprise entre la valeur mesurée plus ou moins la précision de la méthode.
- *Difficulté* d'utilisation d'une technique est une caractéristique subjective, c'est l'effort mis en place pour mettre en œuvre une méthode. Par exemple, si une méthode

donnée nécessite l'utilisation de matériel avancé (par exemple, un analyseur logique), elle est généralement moins accessible aux utilisateurs qu'une méthode purement logicielle.

- *Granularité* représente la taille du code qui peut être mesurée, par exemple, certaines méthodes sont capables de mesurer le temps d'exécution par fonction, thread ou processus, d'autres techniques avancées sont capables de mesurer le temps d'exécution d'une seule instruction [69]. La méthode de mesure, en particulier par des moyens logiciels, a un impact sur le résultat.
- *Résolution* d'une mesure par rapport au temps, est la limitation matérielle associée à chaque méthode. Par exemple, certaines horloges physiques ont une résolution d'une microseconde.

4.5 Méthodes de mesure sur un cœur

4.5.1 Chronomètre

L'utilisation de chronomètres est une méthode de base pour mesurer le temps écoulé entre le début et la fin d'une mesure. Elle nécessite un périphérique de mesure, par exemple une horloge numérique, qui démarre au début et s'arrête à la fin du morceau de code concerné. Le chronomètre peut être soit externe, où l'horloge n'affecte pas le code mesuré, soit instrumenté, où nous devons ajouter du code aux limites du bloc de code mesuré, modifiant ainsi la durée d'exécution du bloc mesuré. Le "bruit" ajouté est d'autant plus gênant que la résolution nécessaire est faible.

La résolution et la précision de cette méthode dépendent de l'outil utilisé pour la mesure. Lorsque ce code est exécuté plusieurs fois, les données de mesure doivent être stockées quelque part dans la mémoire ou les registres du matériel cible. Il y a deux façons simples de les traiter :

1. les données stockées localement sur la cible pendant la session de mesure, et envoyées à la plateforme hôte à la fin, créent plus de contraintes de mémoire et peuvent interférer avec la mesure elle-même, ou être limitées par la taille de la mémoire locale.
2. les données envoyées après chaque mesure à l'ordinateur hôte, ce qui peut ralentir l'ensemble du processus.

Dans les deux cas, les données sont envoyées à l'ordinateur hôte en utilisant une connexion entre la cible et l'hôte (Ethernet, série, etc.).

Toutes ces méthodes nécessitent donc un code supplémentaire pour récupérer les données, ce qui augmente la complexité du système de mesure. Mais il existe un autre moyen de récupérer les données sans ajouter de code. Cette méthode utilise un système de débogage basé sur une sonde de débogage matérielle et un débogueur logiciel. Elle permet à la machine hôte d'avoir une vue complète sur la mémoire et les valeurs des registres.

Elle permet également de transférer la complexité de la récupération des données du système mesuré vers la machine hôte où nous pouvons simplement l'encapsuler dans une partie logicielle d'un environnement de développement intégré (IDE). Nous pouvons ensuite le réutiliser pour effectuer des mesures sur d'autres systèmes, tout en profitant de la possibilité d'afficher les données sur l'hôte avec des statistiques et des graphiques.

Quel que soit le chronomètre utilisé, de nombreux problèmes peuvent survenir en raison de la préemption et des interruptions. Pendant la mesure d'un bout de code, une interruption provoque une fausse valeur de mesure. Nous avons deux solutions pour faire face à ce problème. La première consiste à masquer les interruptions au début de la mesure et à les démasquer à la fin. Mais cette solution n'est pas applicable tout le temps, car le masquage n'est pas toujours disponible et il pourrait aussi affecter le comportement du système ou modifier sa durée. La deuxième solution est de simplement écarter les valeurs qui n'ont pas de sens, par exemple lorsque nous avons une durée beaucoup plus longue que les autres mesures très probablement due à l'interruption. Cette solution doit être appliquée avec précaution pour éviter de rejeter une valeur représentative.

Dans les sous-sections suivantes, nous présenterons les différentes méthodes basées sur la technique du chronomètre qui peuvent être implémenté sur plusieurs types de plateformes.

4.5.2 Les outils de mesure du temps sous Linux

Linux fournit de nombreux outils et fonctionnalités pour mesurer le temps d'exécution, chacun ayant ses propres caractéristiques, avantages et inconvénients.

Les commandes Linux `date` et `time` sont deux outils populaires donnant le temps d'exécution du programme entier. Ils ne peuvent pas être utilisés pour mesurer le temps d'exécution d'une fonction spécifique ou d'un bloc de code. Le principal avantage de ces outils est leur facilité d'utilisation comme montré sur les exemples suivants 4.5.2 et 4.5.2. Néanmoins, ils ne sont pas très précis et ont une granularité relativement élevée (un programme). GProf [34] est un outil de profiling open source pour application Unix, il fait partie du projet GNU et peut être utilisé pour faire le profiling des programmes compilés avec GCC. Afin de recueillir des données de mesure, GProf a besoin d'une instrumentation du code, ce qui peut être fait automatiquement en utilisant le compilateur GCC en passant des options spécifiques.

```
date_de_debut=‘date +%s‘
#Execution du programme
date_de_fin=‘date +%s‘
temps_execution=$((date_de_fin - date_de_debut))
```

```
$ time ./programme
Sortie:
temp

real    0m0.064s
user    0m0.001s
sys     0m0.015s
```

Pour une mesure plus précise avec une granularité plus fine, des horloges POSIX peuvent être utilisées. Deux horloges populaires sont généralement utilisées :

1. CLOCK_REALTIME qui représente l'horloge murale actuelle qui est affectée par la modification de l'horloge de l'heure du système.

2. CLOCK_MONOTONIC qui représente le temps écoulé depuis un point fixe dans le passé, cette horloge n'est pas affectée par les changements de l'horloge.

Linux définit deux fonctions pour la configuration de l'horloge, la première est utilisée pour obtenir la résolution de l'horloge : `clock_getres()`, qui dépend de l'implémentation, et a une précision allant jusqu'à la nanoseconde. La deuxième, `clock_settime()`, est utilisée pour définir l'heure d'une horloge.
La fonction `clock_gettime()` retourne la valeur actuelle de l'horloge sélectionnée. Cette fonction doit être utilisée comme un enveloppement du chronomètre afin de mesurer le segment de code, ceci est fait en prenant sa valeur au début et à la fin du bloc mesuré, comme indiqué ci-dessous :

```
struct timespec start_time, stop_time;
unsigned int measurement;
clock_gettime(CLOCK_MONOTONIC, &start_time);

/* code to measure */

clock_gettime(CLOCK_MONOTONIC, &stop_time);
measure = ((stop_time.tv_nsec - start_time.tv_nsec) +
(stop_time.tv_sec - start_time.tv_sec)*1000000000u);
```

L'un des inconvénients de cette méthode est que nous incluons la durée du `clock_gettime()` lui-même dans la mesure. Si la portion de code mesurée est courte, cela conduit à une surestimation drastique. En utilisant cette méthode, le résultat sera en nanosecondes, c'est pourquoi la partie en secondes est multipliée par un milliard mais cela peut entraîner un débordement sur un système 32 bits car la valeur maximale possible est de 4294967295 nanosecondes ou 4,29 secondes.

4.5.3 Fonctions standard C

De nombreuses fonctions définies en langage C sont disponibles pour la mesure, ces fonctions doivent être utilisées comme un chronomètre. La fonction `clock()`[46] détermine le temps du processeur utilisé et la fonction `time()`[46] détermine le temps actuel du calendrier. La norme ne précise pas leur résolution mais indique qu'elles doivent constituer la meilleure approximation de l'implémentation. La fonction `time()` renvoie le nombre de secondes qui se sont écoulées depuis "l'epoch". "L'epoch" est un point de référence arbitraire qui, dans la plupart des cas, est le 1er janvier 1970 (00 :00 :00). Le problème de la fonction `time()` est qu'elle fonctionne avec des unités de temps qui ne possèdent pas une très fine granularité.

La fonction `clock()`, dans son implémentation la plus utilisée, retourne le nombre de ticks du système qui se sont produits depuis le démarrage du processeur. Cette valeur varie d'une plateforme matérielle à l'autre, puisqu'il s'agit d'une valeur spécifique au processeur. Voici un exemple 4.5.3 qui montre comment elle peut être utilisée en pratique. Le temps d'exécution est égal à la différence entre le début et la fin, divisée par CLOCKS_PER_SEC qui est une macro standard [46] utilisée pour convertir la valeur en secondes.

```
#include <time.h>
clock_t debut, fin;
unsigned long temps_d_execution;
debut = clock();
// Code a mesurer
fin = clock();
temps_d_execution = ((double) (debut - fin)) / CLOCKS_PER_SEC;
```

4.5.4 Compteur de cycles d'horloge

La plupart des processeurs intègre des mécanismes et des registres utilisés pour le profiling et le benchmarking. Ces mécanismes sont généralement basés sur des compteurs de cycles, comptant vers le haut, où la valeur du registre sera incrémentée d'une unité à chaque cycle d'horloge, puis sera remise à zéro en cas de dépassement et générera un événement. Ces mécanismes sont des composants physiques sur puce contrôlés par des instructions assembleur, et ont une faible overhead sur le traitement. Sur les architectures Intel, ce compteur est appelé `Timer Stamp Counter (TSC)`, tandis que les architectures ARM utilisent le compteur de cycles d'horloge `CYCCNT`, qui est un registre de 32 bits. Le compteur est généralement utilisé comme un chronomètre : sa valeur est remise à zéro et il doit être démarré avant la mesure.

4.5.5 Timer/Compteur sur puce

Les puces des processeurs contiennent généralement des horloges, qui peuvent être des timers à usage général ou tout autre type d'horloge. Ces timers sont basés sur des compteurs soit ascendants, soit descendants. Dans les deux cas, ils peuvent être utilisés pour mesurer le temps d'exécution d'un segment de code ou d'une fonction en l'utilisant comme un chronomètre. Cette technique nécessite une phase d'initialisation avant d'effectuer la mesure, pendant laquelle le timer doit généralement être configuré en réglant l'horloge source et d'autres réglages mentionnés ultérieurement.

Ces périphériques possèdent de nombreux registres qui fournissent différentes fonctionnalités. Afin de mesurer une portion de code, deux types de registres sont nécessaires : les registres de contrôle et les registres de valeur de compteur. Ce dernier contient la limite maximale du compteur qui est généralement basée sur une taille de registre comprise entre 8 et 32 bits.

Les fonctionnalités de contrôle varient selon les puces et les types, et la possibilité de le configurer comme un compteur ascendant ou descendant est généralement disponible. Une autre fonctionnalité est la possibilité de spécifier les valeurs initiales, de démarrage, d'arrêt et de réinitialisation, ainsi que de définir un pré-scalaire (diviseur d'horloge) et la granularité du compteur. En ce qui concerne la résolution d'un timer, plus la précision est élevée, moins la durée est grande avant un débordement. Supposons que le timer possède des registres de valeurs de comptage de 32 bits, et que la fréquence de l'horloge source est de 200 MHz, ce qui signifie que sa précision peut atteindre 5 nanosecondes. Cela implique qu'il débordera après 21,47 secondes. Avec le pré-scalaire, l'horloge peut être divisée par 2, 4, 6... de sorte que la mesure peut être faite pour une plus longue durée, au prix de la précision. Pour éviter tout type d'imprécision, il est toujours recommandé de commencer avec un pré-scalaire estimé, puis de le réduire pour augmenter la précision, en écartant les valeurs qui montrent qu'un débordement s'est produit. Dans certains cas, il est même recommandé de mesurer un bloc de code avec une durée d'exécution estimée égale à 10% de la valeur maximale que peuvent contenir les registres de valeur.

On peut trouver ces timers à deux niveaux. Soit ils sont disponibles comme périphérique externe à l'intérieur de la puce, soit dans tous les cœurs du même type. D'une part, le fait d'avoir le timer comme périphérique externe présente de nombreux avantages. En plus de fournir différentes fonctionnalités non liées à la mesure, nous pouvons bénéficier d'une vitesse d'horloge plus élevée permettant une plus grande précision, ainsi que de la taille de la valeur des registres. Il est également plus configurable sans effets secondaires sur le système. Ce type est utilisé dans le chapitre 5 suivant, pour mesurer le transfert de messages entre processeurs AMP. Mais ce type est beaucoup plus spécifique à chaque carte, ce qui rend son utilisation moins portable que les autres types. D'autre part, chaque processeur fournit généralement un timer principal qui peut être utilisé à de nombreuses fins. Le plus courant est le générateur de "tick" d'un système d'exploitation. Dans la plupart des cas, il est configuré pour avoir la même vitesse que le processeur, de sorte que la mesure puisse avoir lieu sans aucune initialisation particulière. L'unité de mesure peut être à la fois en

temps ou en cycles. La conversion n'est pas immédiate car certains processeurs voient leur fréquence modifiée dynamiquement en fonction de la charge ou de la température du cœur. Un tel timer doit être utilisé avec précaution car d'autres composants peuvent l'utiliser et modifier sa configuration (par exemple FreeRTOS utilise le timer Systick pour ses besoins). Cela implique également que si un système d'exploitation est utilisé, il y a un risque important de résultats inexacts. Dans ce cas, il est recommandé d'utiliser une autre méthode ou un autre chronomètre et de ne surtout pas modifier sa configuration de n'importe quelle manière, car cela ferait probablement planter le système d'exploitation. Un exemple populaire de timer utilisé pour les mesures est le timer ARM Systick, que l'on trouve dans tous les microcontrôleurs ARM de différents fabricants, ce qui permet au code d'être beaucoup plus portable que le timer externe. Il est également facile à utiliser [122], voici un exemple 4.5.5 qui montre comment configurer et utiliser le timer Systick.

```
#define INIT_MEASUREMENT unsigned int start_time, stop_time, cycle_count;

#define START_MEASUREMENT SysTick->CTRL = 0; \
SysTick->LOAD = 0xFFFFFFFF; \
SysTick->VAL = 0; \
while(SysTick->VAL != 0); \
SysTick->CTRL = 0x5; \
start_time = SysTick->VAL;

#define STOP_MEASUREMENT stop_time = SysTick->VAL; \
if ((SysTick->CTRL & 0x10000) == 0) \
   cycle_count = start_time - stop_time; \
else \
cycle_count = 0xFFFFFF;

INIT_MEASUREMENT

START_MEASUREMENT
// Code to measure
STOP_MEASUREMENT
```

4.5.6 Analyseur logique

L'analyseur logique est une solution à une catégorie particulière de problèmes. C'est un outil polyvalent qui peut aider au débogage du matériel numérique, à la vérification de la conception et au débogage des logiciels embarqués. Les données stockées dans la mémoire d'acquisition en temps réel peuvent être utilisées dans une variété de modes d'affichage et d'analyse. Une fois que les informations sont stockées dans le système, elles peuvent être

visualisées dans de nombreux formats.

Afin d'utiliser un analyseur logique, deux approches sont possibles. La première consiste à connecter les sondes aux broches (pins) du CPU. L'avantage de cette approche est qu'elle ne perturbe pas le code en temps réel. En même temps, c'est une méthode très compliquée car la corrélation entre la mesure de l'analyseur logique et le code source ne peut se faire que par rétro-ingénierie.

La seconde consiste à envoyer des signaux stratégiques à un port de sortie au début et à la fin de chaque segment de code. Ces signaux sont ensuite lus comme des événements par l'analyseur logique. Les instructions de code contenues dans une macro facilitent la redéfinition de la macro sans affecter le code de l'application. Cependant, comme dans le cas logiciel, cela modifie le code mesuré en ajoutant des fonctions pour permettre la mesure. Cette méthode peut être utilisée aussi bien pour les petits systèmes que pour les grandes applications qui utilisent des systèmes d'exploitation commerciaux temps réel.

4.6 Comparaison expérimentale

Nous avons mesuré le temps d'exécution en utilisant différentes méthodes sur trois cartes hétérogènes, la STM32MP157-DK2 de STMicroelectronics, l'architecture détaillée dans la section 3.9.1.1, qui se compose d'un Cortex-A double cœur 650 MHz et d'un Cortex-M simple cœur 209 MHz, et deux cartes de NXP, la i.MX7ULP (Cortex-A 500 MHz et un Cortex-M 200 MHz) et la i.MX8EVK (Cortex-A 1 GHz et un Cortex-M 240 MHz). Nous avons utilisé le benchmark "Adaptive Differential Pulse Code Modulation" qui fait partie de la suite TACLe Benchmark [28], le test a été répété au moins une centaine de fois. Dans la Figure 4.4, nous montrons sous forme de box plots les résultats obtenus en utilisant trois méthodes de mesure : le compteur de cycles ARM (CYCCNT), le timer du cœur ARM (SYSTICK) et un timer périphérique spécifique à chaque carte. Les résultats sont similaires pour les trois méthodes. Cela montre une cohérence qui est rassurante puisque la seule différence réside dans le choix de la méthode la plus facile à installer.

La figure 4.5 montre les résultats du même code de référence mais en utilisant la mesure basée sur l'horloge POSIX à l'aide de la fonction `get_clocktime()` sur un système d'exploitation Linux en condition de charge normale ou dans un environnement stressé à l'aide de l'outil hackbench [123], qui crée de nombreux processus Linux qui envoient des données entre des expéditeurs et des récepteurs via des sockets. Nous pouvons constater que les résultats dans un environnement stressé sont plus étalés qu'en charge normale notamment pour carte i.MX8. Le temps d'exécution mesuré en nanosecondes varie beaucoup en fonction de la fréquence du processeur, par conséquent, le i.MX7 avec une fréquence de 500 MHz est le plus lent et le i.MX8 avec 1 GHz est le plus rapide, bien que si nous calculons le nombre de cycles équivalents, cela donne des valeurs proches en terme de nombres de cycles.

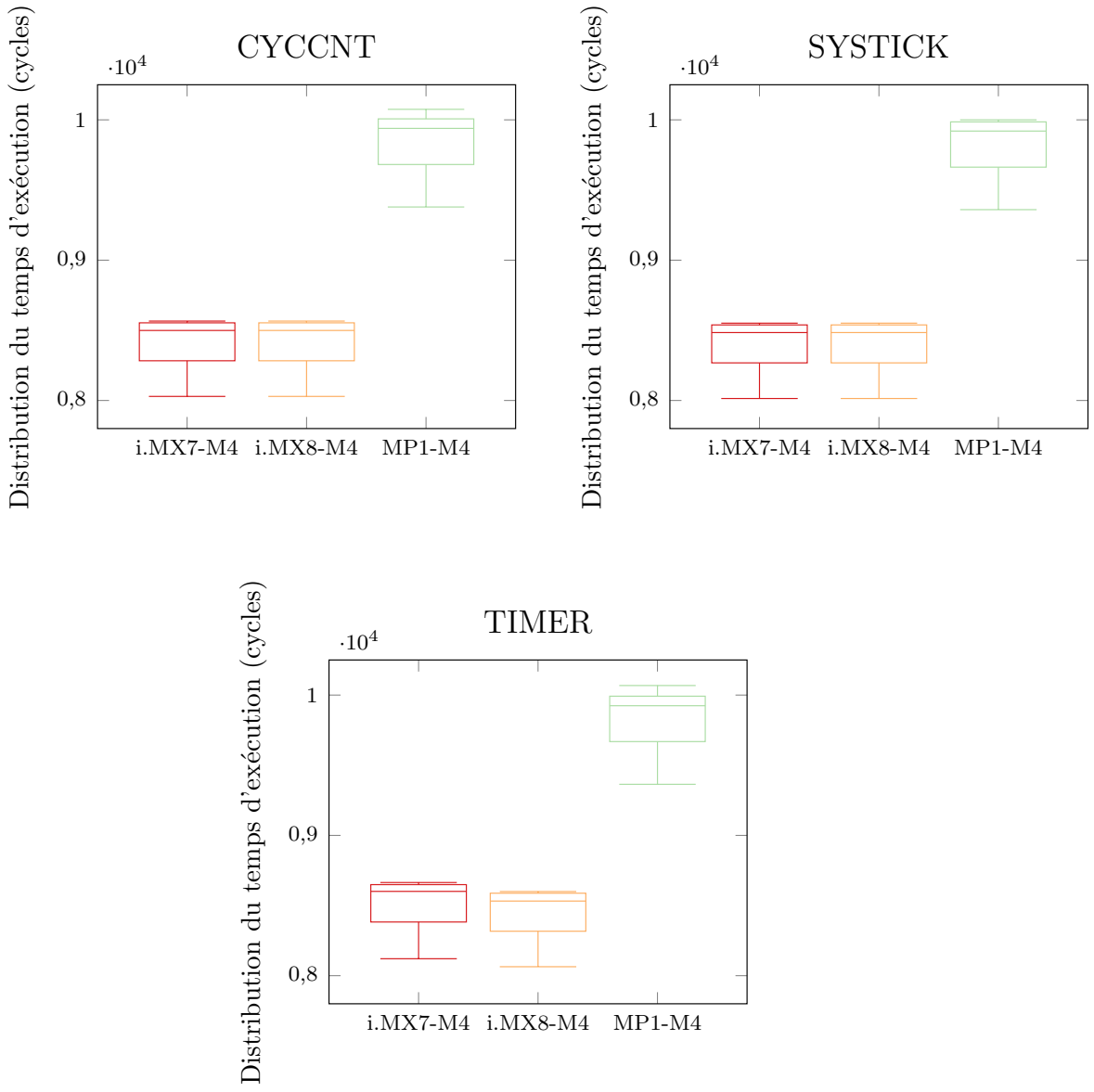

FIGURE 4.4 – Distribution du temps d'exécution selon des différentes méthodes

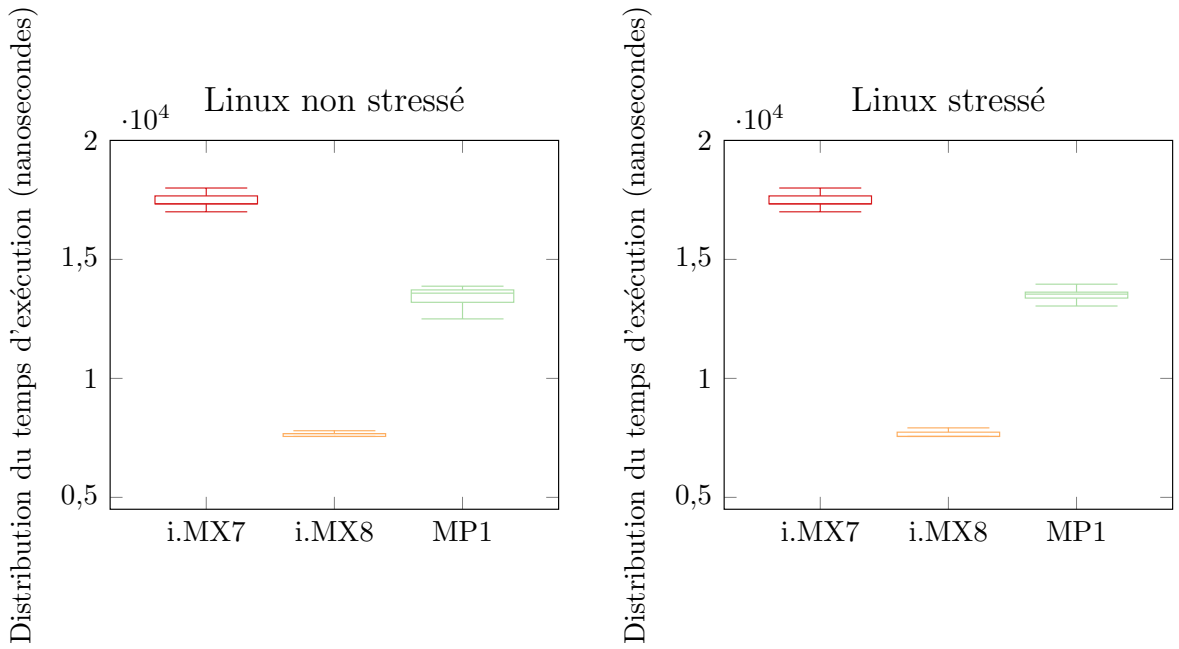

FIGURE 4.5 – Distribution du temps d'exécution sous Linux

Nous avons remarqué que le temps d'exécution sur la carte i.MX8 est plus rapide dans un environnement stressé dans les premiers tests. Cela était dû au fait que Linux modifie la fréquence de façon dynamique en fonction de la charge du processeur. La fréquence de base était de 1 GHz en charge normale, alors qu'elle a été augmentée à 1,5 GHz automatiquement dans un environnement stressé. C'est pourquoi il était important de désactiver le DVFS, et d'utiliser une fréquence fixe lors de la mesure du temps d'exécution avec cette méthode. Cela a permit d'avoir des mesures plus précises sans inclure la latence liée au changement de fréquence.

4.7 Mesure expérimentale de durée d'exécution sur un système multicœur SMP

Les architectures multicœurs peuvent répondre aux besoins temps réel avec un avantage en terme de réduction de la consommation d'énergie. L'utilisation des architectures multicœurs pour le temps réel suscite un intérêt croissant en raison de leur puissance de calcul élevée et leur faible coût.

La plupart des systèmes multicœurs existant sont de type SMP. Un système multicœur effectue toutes les tâches, y compris les fonctions du système d'exploitation et les tâches utilisateur. La figure 4.6 illustre une architecture SMP typique avec quatre cœurs. On remarque que chaque cœur possède ses propres registres, ainsi qu'un cache local. Cependant, tous les cœurs partagent la même mémoire physique via le bus système. L'avantage de ce modèle est que de nombreuses tâches peuvent être exécutées simultanément sans que les performances ne se dégradent significativement. Un autre type de système mul-

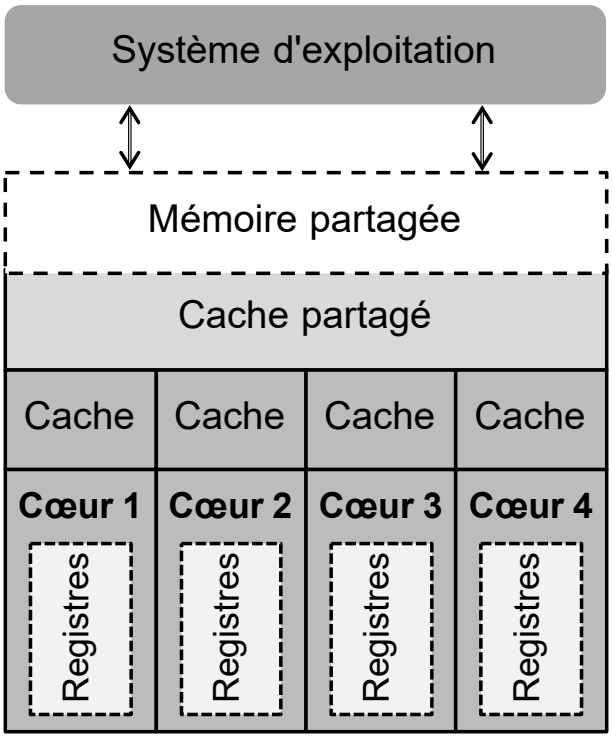

FIGURE 4.6 – Système multicœur symétrique

tiprocesseur est le système en cluster, qui regroupe plusieurs cluster ou chacun contient plusieurs cœurs.

Dans les systèmes multicœurs SMP où la migration est supportée, une tâche préemptée peut reprendre son exécution sur un cœur différent. La migration des tâches est un mécanisme géré par le système d'exploitation qui permet aux tâches d'être transférées vers autre cœur. Il existe plusieurs types de migration, notamment la migration au niveau du cluster, où chaque tâche peut s'exécuter sur un cluster des cœurs disponibles, ou au niveau global, où chaque tâche peut reprendre son exécution sur n'importe quel cœur.

Le problème de l'ordonnancement des tâches temps réel sur le multicœur a reçu beaucoup d'attention dans la littérature [18]. Des stratégies d'ordonnancement optimales (en négligeant les durées de préemption et migration) sur de telles plateformes ont été proposées, comme par exemple RUN [96]. Cependant, les stratégies généralement adoptées supposent que le WCET des tâches ne varie pas en fonction de ce qui est en cours d'exécution sur les autres cœurs au moment de leur exécution. Les outils d'analyse statique pour le multicœur nécessitent des connaissances approfondies sur les parties du logiciel qui peuvent fonctionner en parallèle. Par ailleurs, les propriétés d'utilisation des ressources exportées par les techniques de régulation des ressources pourraient être intégrées aux techniques d'analyse statique afin de limiter l'analyse à une seule application et d'obtenir des résultats. Sur les multicœur, l'analyse statique des caches proposée par [112] a été étendue aux caches partagés [68]. De même, les bus de mémoire partagée ont été analysés dans [54].

En outre, en fonction du modèle d'accès aux ressources partagées des applications

exécutées en parallèle, les accès aux ressources partagées à partir de plusieurs cœurs peuvent entraîner un conflit d'accès ou pas. Cette incertitude augmente le temps d'exécution des applications de manière imprévisible. Il est donc difficile d'estimer le WCET. Ce problème est traité en transformant radicalement les architectures multicœurs pour les rendre plus prédictibles au niveau du timing. La transformation la plus simple est le découpage statique du temps pour accéder aux ressources partagées. [101] propose une technique permettant de mesurer le WCET d'applications exécutées sur des architectures multicœurs à l'aide d'outils d'analyse temporelle existants pour les architectures monocœurs. La technique consiste à insérer un module d'observation du cache et un module de suivi temporel basé sur un compteur.

4.8 Contribution industrielle

Nous avons développé un nouveau module pour l'outil *System Workbench for Linux* (SW4Linux) qui permet aux utilisateurs de réaliser des mesures de temps d'exécution des fonctions C/C++ en se basant sur les méthodes présenté dans ce chapitre. Pour cet outil, nous avons essayé de rendre la technique facile à utiliser, ne nécessitant que le matériel cible et l'ordinateur de développement, donnant la plus haute résolution possible en fonction des horloges ou du système d'exploitation disponibles. Le module développé récupère les données de mesure automatiquement à travers le débogueur et les exporte vers une fiche de données. La limitation, comme pour les autres méthodes dynamiques, est l'impossibilité de connaître la précision. Une présentation détaillée de cet outil est faite dans le chapitre 6.

4.9 Conclusion

Ce chapitre a permis de souligner l'importance de l'analyse de temps d'exécution en comparant les différents types d'analyse. L'analyse statique est la méthode la plus sûre applicable dans le cadre d'un système temps réel dur ; mais sa complexité et son caractère très spécifique rend son application très coûteuse et très difficile dans certains cas.

Contrairement à ce premier type d'analyse, l'approche dynamique est plus accessible mais non applicable sur des systèmes temps réel dur. En industrie, l'approche dynamique pour les applications nécessitant des systèmes temps réel *mous* peut suffire, ce qui peut lui permettre de se substituer, tout en perdant en sûreté, à l'analyse statique. Il faut noter que si l'environnement nécessite des contraintes temporelles plus strictes, une analyse hybride, qui représente un mixe entre l'analyse statique et dynamique ou une analyse probabiliste basée sur des estimations statistiques, peuvent être utilisées.

La complexité de différentes plateformes récentes rend l'approche basée sur la mesure plus adaptée. Une comparaison entre différentes méthodes de mesure de temps d'exécution, basée sur plusieurs attributs (pour rappel, la précision, la difficulté, la granularité et

la résolution) a été effectué entre elles sur plusieurs cibles de différentes caractéristiques. Il a été constaté expérimentalement que la plupart des méthodes de mesure basées sur des compteurs donnent des résultats similaires, ce qui implique que le choix des méthodes doit reposer sur la simplicité et l'accessibilité.

Une nouvelle fonctionnalité a été développée pour l'outil SW4Linux qui permet d'effectuer des mesures, en utilisant les techniques présentées dans ce chapitre, ainsi que de récupérer les résultats facilement.

Chapitre 5

Mesure de durée d'exécution et migration sur les systèmes AMP

Sommaire

5.1 Introduction 113
5.2 Application AMP 113
5.3 Mesure de durée d'exécution sur les systèmes AMP 115
5.4 Méthodes de mesure 116
5.4.1 Résolution du timer et débordement 116
5.4.2 Remontée des mesures vers la station de développement 117
5.4.3 Méthodes de mesure possibles 118
5.4.4 Validation expérimentale 122
5.5 Migration hétérogène 124
5.5.1 Introduction 124
5.5.2 Méthode de migration hétérogène proposée 124
5.6 Étude de cas ROSACE 132
5.6.1 Charge normale 133
5.6.2 ROSACE stressé 133
5.7 Discussion et perspective 136
5.8 Conclusion 137

5.1 Introduction

Les architectures hétérogènes asymétriques sont aujourd'hui très populaires. La plupart des SoCs récents contiennent différents types de cœurs, il a été prouvé que cela permet d'obtenir le matériel le plus efficace pour le logiciel [116]. Les cœurs hétérogènes sont implémentés en fonction de l'application ciblée par la carte embarquée. Par exemple, sur le marché des téléphones portables, big.LITTLE est un HMPSoC très répandu, offrant une haute performance avec une faible consommation d'énergie. Récemment, les cœurs dédiés à l'intelligence artificielle (NPU) ont également fait leur apparition sur ce marché. D'autre part, pour les systèmes embarqués temps réel, nous trouvons des cœurs plus optimisés pour le déterminisme du système, comme le Cortex-M ou le Cortex-R, à côté d'autres cœurs optimisés pour de hautes performances mais moins de déterminisme comme le Cortex-A.

Plusieurs défis existent pour cette technologie, en particulier pour un système multitâche temps réel. Le premier défi est d'utiliser efficacement les ressources disponibles, il est inutile d'avoir des cœurs disponibles qui ne sont pas utilisés car le logiciel ou le système d'exploitation ne peut pas les exploiter. En ce qui concerne l'ordonnancement, ce n'est pas une décision simple de déterminer quelle tâche doit être exécutée sur quel cœur (outre le fait qu'ils ont des ISA différentes). Cela implique qu'il n'est pas possible de migrer les tâches entre les cœurs de la même manière que sur les systèmes SMP. Dans ce chapitre, nous abordons une méthode de migration des tâches sur des cœurs AMP hétérogènes, basée sur des points logiciels de migration.

Le deuxième défi est l'analyse du temps d'exécution. Comme ces cartes sont destinées à des applications temps réel, l'analyse du temps est essentielle. Dans ce chapitre, nous présentons la méthode que nous avons utilisée pour mesurer la latence de communication inter-processeur entre les cœurs hétérogènes.

Enfin, nous avons utilisé l'étude de cas ROSACE [86] afin de comparer le système avec ou sans migration hétérogène et avons conclu en indiquant les cas où il est avantageux d'effectuer la migration vers un cœur hétérogène.

5.2 Application AMP

Pour les HMPSoCs, il est possible d'utiliser chaque partie seule pour l'application voulue sans avoir besoin d'accéder à des ressources partagées, à des périphériques communs, ou de communiquer entre les différentes parties. Par exemple, pour créer une interface graphique qui utilise le réseau et un écran, le HMPSoC STM32MP1 peut être utilisé, mais dans ce cas le Cortex-M4 est ignoré comme s'il n'existait pas. Cela ne permet pas une utilisation totale des ressources et, dans certaines applications, le Cortex-M est nécessaire pour réduire la consommation d'énergie ou pour exécuter des tâches temps réel. Donc il peut exister un besoin de communication et d'éléments partagés.

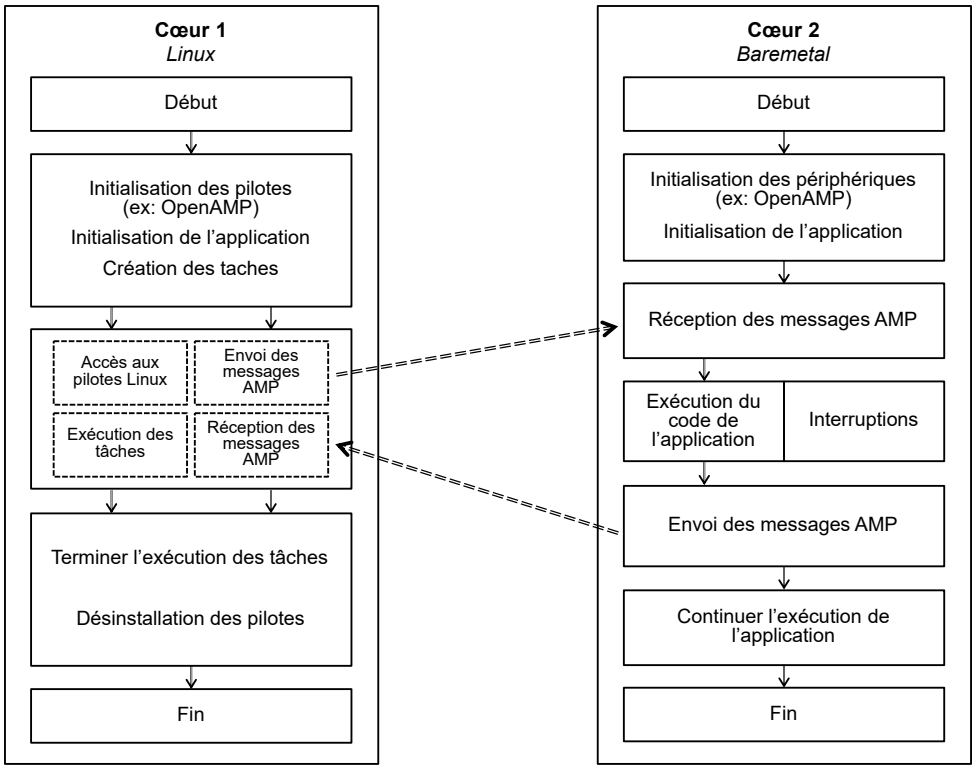

FIGURE 5.1 – Application AMP utilisant des cœurs hétérogènes

Une application hétérogène typique est illustrée sur la figure 5.1. Dans ce cas, il y a deux cœurs hétérogènes, le premier sous Linux et le second sans système d'exploitation. Les deux cœurs tournent en parallèle, en exécutant des codes distincts qui sont compilés à l'aide de compilateurs différents. Sur le premier cœur qui utilise Linux, le code typiquement trouvé est celui qui exige de hautes performances de calcul et qui a besoin d'accéder à des pilotes complexes comme par exemple pour l'interface graphique, l'Ethernet ou l'USB. Il initialise tout d'abord les pilotes, qui respectent les exigences du modèle des pilotes Linux, et utilisés par l'application, y compris le pilote de communication inter-cœurs comme OpenAMP, pour lequel il y a au moins le besoin d'utiliser l'appel système `open()` et de vérifier si le canal de communication avec l'autre cœur est établi. Ensuite, il initialise la partie spécifique à l'application, et, à la fin de la phase d'initialisation, il crée les différents processus et tâches qui représentent le code de l'application qui s'exécute du côté Linux.

D'autre part, comme le deuxième cœur est utilisé sans système d'exploitation, le code ne peut être exécuté que séquentiellement, à l'exception des routines de service d'interruptions qui utilisent le séquenceur d'interruptions codé en dur. La configuration des périphériques doit être faite manuellement en se basant sur le firmware, idem pour la configuration des interruptions. Par exemple, le périphérique de communication avec l'autre cœur nécessite une configuration en utilisant le firmware OpenAMP. Une fois tous les périphériques initialisés, le code d'application spécifique doit à son tour être initialisé. Ensuite, l'application se met en attente de messages. Une fois que le message est reçu, le cœur exécute les fonctions suivantes en se basant sur ce message comme valeur d'entrée. Lorsqu'il termine l'exécution, un message sera renvoyé à Linux via OpenAMP avec les

éventuels résultats calculés.

Cette partie de communication AMP peut être répétée plusieurs fois, pour assurer la communication sur plusieurs parties de l'application. Dans le cas où une ressource partagée (comme un périphérique GPIO) est utilisée par les deux cœurs, il est important d'éviter tout conflit d'accès, dans ce cas un périphérique dédié comme le sémaphore matériel (ou spinlock matériel) détaillé dans la section 3.10.3 peut être utilisé.

À la fin, les deux processeurs poursuivent l'exécution des codes spécifiques à l'application, jusqu'à ce que toutes les tâches soient terminées. Ensuite, les pilotes doivent être fermés correctement.

Les systèmes AMP peuvent être utilisés dans de nombreux domaines : des systèmes critiques de sécurité, objets connectés ou drones. Par exemple, dans un drone autonome, les cœurs à haute performance peuvent être utilisés pour effectuer un traitement d'image, nécessaire pour la classification d'image, puis les messages AMP sont envoyés au microcontrôleur qui sert à contrôler le système de vol. Un autre exemple est celui où une très faible consommation d'énergie est requise (comme dans les objets connectés), le microcontrôleur restera en veille pour effectuer le traitement à faible consommation d'énergie, puis il réveillera l'autre cœur lorsque le traitement à haute performance sera nécessaire.

5.3 Mesure de durée d'exécution sur les systèmes AMP

Comme discuté dans le chapitre 4, l'analyse de WCET sur les architectures modernes est confrontée à de nombreux défis. L'analyse statique est la méthode la plus sûre, mais elle nécessite des outils particuliers et un support pour les processeurs modernes qui sont généralement très complexes, ce qui rend la tâche très ardue dans certains cas. Comme cela nécessite un modèle de matériel spécifique, effectuer cette opération pour les HMPSoCs est très difficile, en particulier sur les processeurs d'applications qui déploient de nombreux éléments d'optimisation en moyenne. Ainsi, les méthodes d'analyse de WCET basées sur la mesure peuvent être les seules applicables sur ce type d'architecture.

Notons cependant que sur les cœurs de type microcontrôleurs (généralement Cortex-M ou Cortex-R), il sera plus aisé de définir un modèle pour l'analyse statique de WCET, ce qui permettrait d'obtenir un WCET totalement sûr. Nous pourrions alors se retrouver avec une analyse basée mesure pour les cœurs d'application, et une analyse soit basée mesure, soit statique, sur les microcontrôleurs. Il manque cependant un point important pour l'exécution d'une application qui utiliserait plusieurs types de cœurs : la mesure des durées de communication d'un cluster à un autre. Notre proposition de méthode de mesure est donc présentée dans cette section.

Dans ce chapitre, nous mesurons des durées de communication entre un processeur émetteur et un processeur récepteur faisant partie de la durée d'exécution sur les systèmes AMP. Cependant, dans certains cas où il y a un retour d'information du récepteur vers l'émetteur, le récepteur originel émet un message vers l'émetteur originel qui est donc

récepteur dans cette seconde phase de communication. Afin de bien distinguer les rôles, nous supposons que nous représentons à gauche le processeur (initialement) émetteur, et à droite le processeur (initialement) récepteur. Le processeur gauche est donc le processeur dont nous voulons mesurer la vitesse d'émission vers le processeur droit.

5.4 Méthodes de mesure

Pour mesurer le temps d'exécution d'un programme qui s'exécute sur des multicœurs asymétriques, il faut utiliser des techniques indépendantes de tout cœur. En outre, le cycle n'a pas de sens pour la mesure de temps dans ce cas, tant que les cœurs ne sont pas synchronisés et que chacun tourne à sa propre fréquence. C'est la raison pour laquelle il est nécessaire d'utiliser une unité de temps comme la nanoseconde ou la microseconde μs où les références temporelles sont les mêmes.

Un timer externe sur puce est un bon choix dans ce cas, mais cela présente de nombreuses conditions et limitations qui dépendent de la plateforme utilisée et de la disponibilité des composants. Toute mesure effectuée d'un cœur à l'autre nécessite que ce timer puisse être contrôlé par tous les cœurs AMP. Nous pouvons citer d'autres conditions comme la possibilité de réinitialiser ou de reconfigurer le timer, l'existence de certains périphériques de synchronisation comme le sémaphore matériel (voir section 3.10.3) ou la possibilité d'établir la connexion entre le processeur et le débogueur, car sur certaines plateformes, la fonctionnalité de débogage est limitée à certains cœurs ou il est très complexe d'accéder aux autres et nécessite un outil spécifique.

5.4.1 Résolution du timer et débordement

La résolution du timer choisi doit permettre la mesure d'une durée de l'ordre de quelques dizaines de microsecondes au moins. Il existe deux familles de méthodes : l'une démarrant par une remise à zéro (reset) du timer, l'autre prenant sa valeur courante au départ. Dans la seconde famille, comme il est possible que le timer ait été réinitialisé durant la mesure, il convient de détecter ce débordement. Il y a débordement si la nouvelle valeur (ou finale) est inférieur de l'ancienne valeur (ou valeur initiale) pour un timer configuré en mode incrémental, alors dans ce cas il faut ajouter sa valeur maximale à la valeur absolue de la différence entre les deux valeurs (finale et initiale), afin de la calculer, l'opérateur modulo a été utilisé, pour les langages C/C++, il est défini dans les standards [46] et [23] et il est basé sur la méthode de troncature de la partie décimale :

$$x\%y = x - y * partie_entiere(x/y)$$

Dans les formules suivantes `x` peut être inférieur à `y` dans ce cas la `partie_entiere(x / y)` est égale à 0 alors `x % y = x - y * 0 = x`. Si `x` et `y` sont positifs le modulo retourne une valeur positive et si `x` est négatif et `y` est positif, le modulo retourne une valeur

négative.

Voici la formule utilisée pour calculer la mesure :

$$mesure = (((tf - ti)\%max) + max)\%max$$

`tf` représente la valeur finale du timer, `ti` la valeur initiale et `max` la valeur maximale.

Nous pouvons aussi utiliser :

$$mesure = ((diff\%max) + max)\%max$$

Avec `diff` est la différence signée entre `tf` et `ti`.

Si `diff` est supérieur à 0 (le timer n'a pas débordé) :

$$diff\%max = diff \qquad \text{Car } 0 < \texttt{diff} < \texttt{max}$$
$$(diff\%max) + max = diff + max$$
$$((diff\%max) + max)\%max = (diff + max)\%max$$
$$(diff+max)\%max = ((diff\%max)+(max\%max)))\%max \qquad \text{Formule de distributivité}$$
$$((diff\%max) + (max\%max)))\%max = (diff + 0)\%max) = diff$$
$$((diff\%max) + max)\%max = diff$$

Alors Si le timer n'a pas débordé (`tf` est supérieur à `ti`) la formule retourne la différence entre les deux valeurs.

Si `diff` est inférieur ou égale à 0 (le timer a débordé) :

$$diff\%max = -|diff| \qquad \text{Car } \texttt{diff} < 0 \text{ et } \texttt{|diff|} < \max$$
$$(diff\%max) + max = max - |diff|$$
$$((diff\%max) + max)\%max = (max - |diff|)\%max$$
$$(max - |diff|)\%max = (max - |diff|) \qquad \text{Car } \texttt{max - |diff|} < \texttt{max}$$

En conclusion, si le timer a débordé (`tf` est inférieur à `ti`) la formule retourne la valeur maximale du timer moins la différence entre la valeur finale et initiale.

On pourrait penser qu'il est plus simple, dans tous les cas, de mettre le timer à zéro au début de chaque mesure, cependant, dans le cas où on voudrait mesurer à la suite plusieurs durées d'envoi de message, sans aucun contrôle du rythme d'envoi, il est possible que le second envoi remette le timer à zéro avant sa lecture par le processeur droit pour l'envoi précédent.

5.4.2 Remontée des mesures vers la station de développement

Suite à la fin de la compagne de mesure, il faut savoir quel processeur permet de remonter les données vers la station de développement utilisée pour effectuer les mesures.

Celui-ci peut être le processeur gauche ou bien le processeur droit. Afin de ne pas impacter la mémoire et ainsi potentiellement altérer l'état des caches si le processeur collectant les mesures est de type application, ou bien ne pas avoir de problème lié à la taille de la mémoire lorsque la mesure est collectée sur un microcontrôleur, nous utilisons la connexion débogueur pour extraire les valeurs mesurées vers la station de développement. Il convient donc de fournir des méthodes de mesure permettant de mesurer une durée de communication gauche-droite permettant d'extraire via le déboguer les données soit du processeur gauche, soit du processeur droit, car cette extraction peut être imposée par le HMPSoC cible.

5.4.3 Méthodes de mesure possibles

Nous présentons ci-dessous toutes les méthodes de mesure possibles :

- la méthode chronométrique `gauche-droite` illustrée dans la figure 5.2, utilisable si le processeur droit remonte les mesures, consiste à (1) le cœur gauche lit le timer au moment de l'envoi de message, et transmet sa valeur parmi les données passées au cœur droit. (2) le cœur droit, à réception, lit le timer, retrouve la valeur de timer passée dans les données et en déduit la durée de communication. Ne peut s'appliquer que si on mesure une communication incluant des données. Doit prendre en compte le débordement éventuel du timer. La campagne de mesure est constituée de plusieurs milliers de mesures. Cela ajoute une nouvelle contrainte car nous devons éviter de répéter la mesure lorsque nous envoyons un message du cœur gauche vers le cœur droit avant de le lire, et comme le côté gauche ne sait pas combien de temps dure cette opération, cela pourrait créer un conflit où nous envoyons plusieurs messages avant de les recevoir de l'autre côté. En utilisant cette méthode, nous pouvons surestimer le temps d'attente entre deux mesures, afin d'attendre un délai entre deux envois successifs, et éviter ainsi que les messages s'accumulent et interfèrent les uns avec les autres. Cependant, cela peut conduire à une campagne de test plus longue.

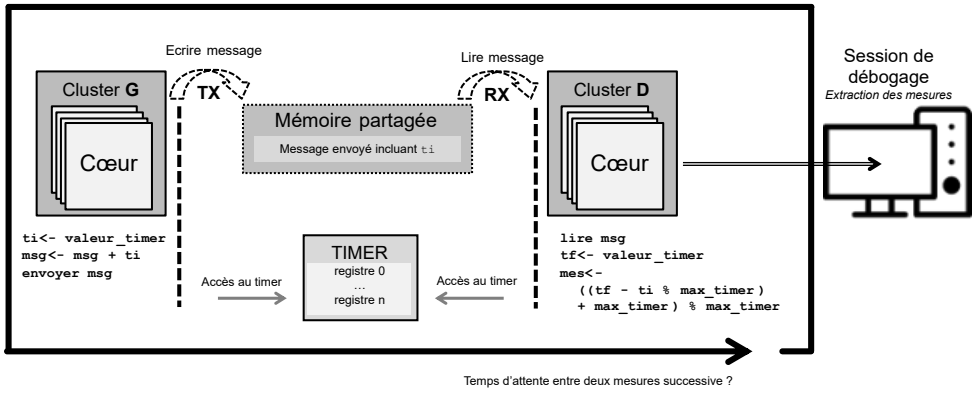

FIGURE 5.2 – Méthode de mesure : gauche-droite

- la méthode chronométrique `gauche-droite-acknowledge` illustrée dans la figure 5.3 est identique à la méthode précédente sauf qu'ici le cœur droit répond par une

message vide une fois que le temps d'exécution est mesuré, ce message représente donc un accusé de réception et signifie qu'un nouveau message peut être envoyé. La limitation de cette méthode est que, sur certaine plateforme, il n'est pas possible d'envoyer des messages dans les deux directions.

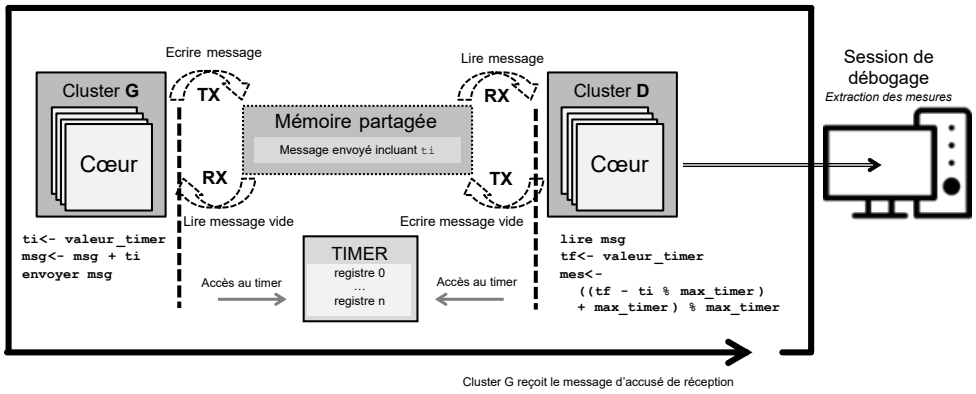

FIGURE 5.3 – Méthode de mesure : gauche-droite-acknowledge

- la méthode chronométrique `gauche-droite-hsem` avec sémaphore matériel illustrée dans la figure 5.4 est identique à la méthode 5.2 mais nous utilisons un sémaphore matériel pour la synchronisation entre les tests successifs. Le cœur gauche verrouille le sémaphore, envoie le message puis il se bloque sur verrouillage avant le message suivant. Le cœur droit libère le sémaphore quand il termine son travail, ce qui débloque le cœur gauche. Cette méthode est la méthode recommandée pour la synchronisation car elle est la plus rapide et elle est basée sur un périphérique adapté à la synchronisation entre les cœurs hétérogène.

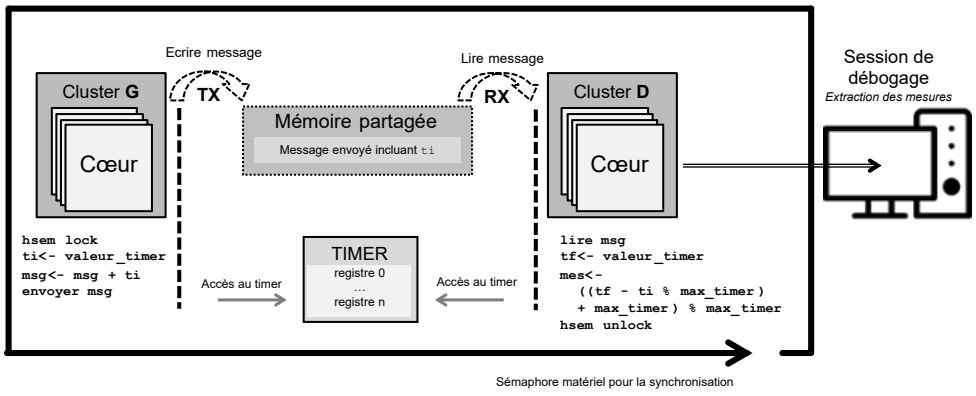

FIGURE 5.4 – Méthode de mesure : gauche-droite-hsem avec sémaphore matériel

- la méthode chronométrique `zero-gauche-droite`, illustrée dans la figure 5.5, se distingue de la gauche-droite par le fait que le timer est mis à 0 par le processeur gauche juste avant envoi. Elle a l'avantage de ne pas avoir à gérer les débordements de timer. Un sémaphore matériel est utilisé pour synchroniser les mesures successives nécessaires pour la campagne de test, mais la méthode avec le message d'accusé de réception pourrait être utilisée également. La seule limitation est que pour certaines plateformes, il n'est pas possible de remettre le timer à zéro. Comme la mesure est

égale à la valeur du compteur, il est possible d'extraire les valeurs à partir un des deux processeurs, le choix devrait être sur celui qui est le plus simple pour lancer une session de débogage.

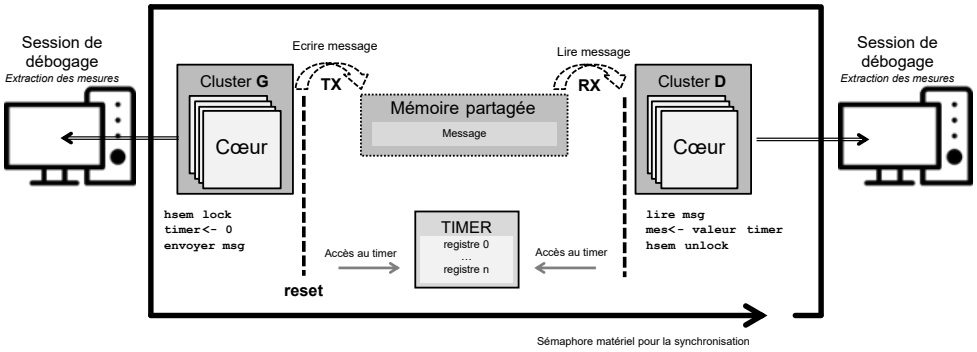

FIGURE 5.5 – Méthode de mesure : zero-gauche-droite

- la méthode chronométrique `gauche-droite-gauche` illustrée dans la figure 5.6, utilisable si le processeur gauche remonte les mesures, est applicable même si aucune donnée n'est passée d'un cœur à l'autre, dans ce cas on parle de transfert de contrôle, ou d'événement. Le cœur gauche lit le timer juste avant envoi, le cœur droit à réception lit le timer, et renvoie la valeur lue vers le cœur gauche. Ceci permet de mesurer des durées de communication gauche-droite même sans transfert de données. Les débordements éventuels de timer doivent être compensés. De plus, cette méthode nécessite la mise en place de communications dans les deux sens, ce qui va rallonger la durée des mesures et complexifier l'implémentation de la mesure.

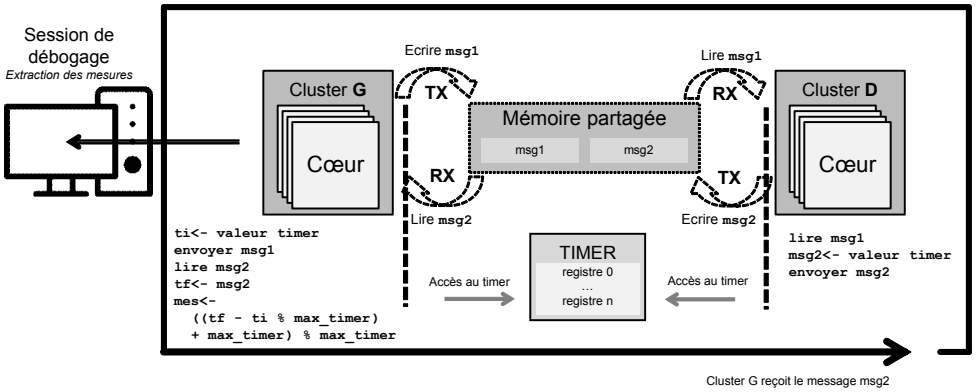

FIGURE 5.6 – Méthode de mesure : gauche-droite-gauche

- la méthode chronométrique `zero-gauche-droite-gauche`, illustrée dans la figure 5.7, est similaire à la méthode précédente sauf que le timer est mis à zéro au moment de l'envoi par le processeur gauche. Cela permet d'éviter la gestion des débordements, mais présente les mêmes désavantages que la méthode précédente. Il est possible d'extraire les valeurs à partir un des deux processeurs gauche ou droit.
- la méthode chronométrique `start-stop` est de type `zero-gauche-droite` illustrée dans la figure 5.8, mais utilisable lorsque la mesure est transmise par le processeur gauche. Elle permet surtout de savoir à quel moment une seconde mesure peut être

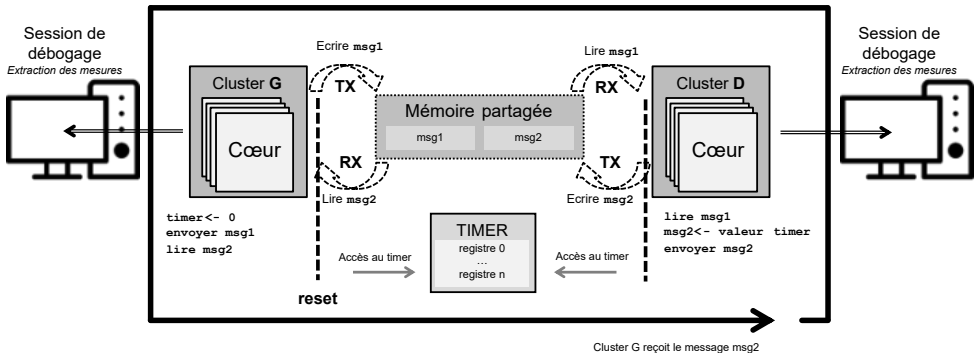

FIGURE 5.7 – Méthode de mesure : zero-gauche-droite-gauche

démarrée. Juste avant envoi, le cœur gauche met le timer à zéro et le démarre, avant d'envoyer un message au cœur droit. Le cœur gauche se met alors en scrutation périodique du timer, jusqu'à ce que celui-ci s'arrête (arrêt détecté par deux lectures consécutives de la même valeur). Le cœur droit, à réception du message, a juste à stopper le timer. D'un point de vue programmation, cette méthode est la plus simple. De plus, elle permet de facilement enchaîner des mesures successives.

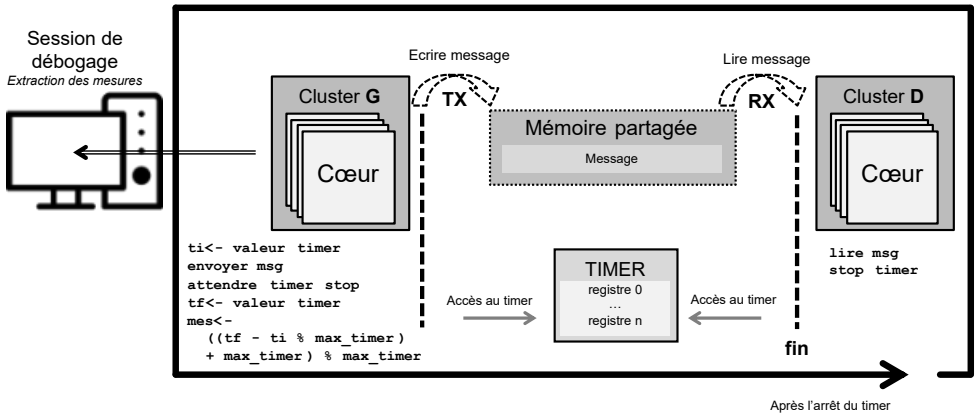

FIGURE 5.8 – Méthode de mesure : start-stop

- la méthode chronométrique `zero-gauche-droite-stop`, illustrée dans la figure 5.9, s'inspire de la méthode précédente. Dans cette méthode, le processeur gauche met le timer à zéro, envoie un message au processeur droit, puis scrute la valeur du timer jusqu'à son arrêt. Le processeur droit à réception du message stoppe le timer et le lit. Comme pour la méthode `zero-gauche-droite`, il est possible d'extraire la mesure soit du processeur droit, soit du processeur gauche.

Les deux dernières méthodes cumulent flexibilité et simplicité, la méthode start-stop permet des campagnes de mesure, avec une remontée des données par le processeur gauche. La méthode zéro-gauche-droite-stop permet aussi des campagnes de mesures, avec remontée des données de mesures par le processeur droit. Le tableau 5.1 récapitule des différentes méthodes de mesures présentées dans cette section. La colonne `msg=t` indique qu'il est nécessaire d'émettre la valeur du compteur dans le message dont on veut mesurer la durée, influençant potentiellement celle-ci. La colonne `ACK` signifie qu'il est nécessaire d'envoyer un accusé de réception, non mesuré, qui contient potentiellement la valeur du timer. La

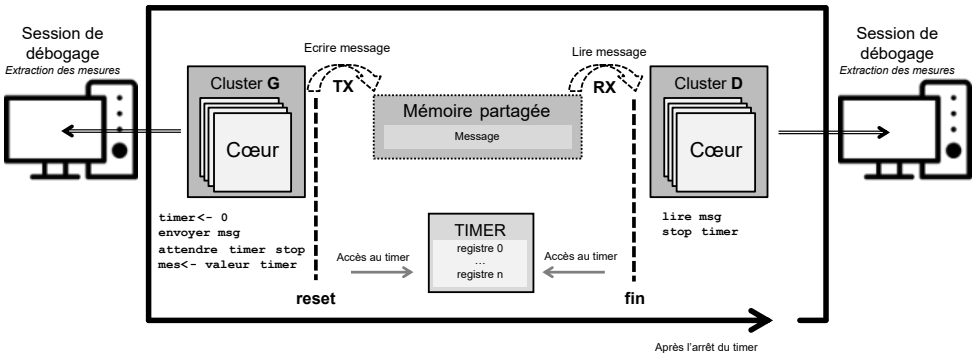

FIGURE 5.9 – Méthode de mesure : zero-gauche-droite-stop

Méthode	msg=t	ACK	HSEM	DebugG	DebugD
gauche-droite	X	-	-	-	X
gauche-droite-acknowledge	X	X	-	-	X
gauche-droite-hsem	X	-	X	-	X
zero-gauche-droite	-	-	X	X	X
gauche-droite-gauche	X	X	-	X	-
start-stop	-	X	-	X	-
zero-gauche-droite-stop	-	-	-	X	X

TABLEAU 5.1 – Comparaison des méthodes de mesure proposées

colonne `HSEM` indique qu'il est nécessaire d'utiliser un mécanisme de sémaphore matériel pour synchroniser les mesures successives. Les colonnes `DebugG` et `DebugD` sont les seules souhaitables, elles indiquent s'il est possible de collecter les mesures par la liaison de débogage à gauche (émetteur) et/ou à droite (récepteur). Rappelons que la difficulté ou possibilité d'effectuer ce débogage n'est pas symétrique et que l'une peut être simple à mettre en oeuvre alors que l'autre est complexe, voire impossible. Il devient clair au vu de ce tableau que la méthode `zero-gauche-droite-stop` est la plus simple et la plus flexible, étant donné que la synchronisation entre les mesures successives est assurée avec les messages, un mécanisme déjà implémenté, et la valeur de la mesure est simplement la valeur récupérée du timer. C'est celle que nous utiliserons pour toutes les mesures par la suite.

5.4.4 Validation expérimentale

Nous avons effectué 50.000 mesures, en utilisant la méthode `zero-gauche-droite-stop`, représentées dans quatre boîtes à moustaches, sur deux cartes HMPSoCs, la carte STM32MP1-DK2 détaillé dans la section 3.9.1.1 et la carte i.MX8 qui contient un processeur quadricœur Cortex-A (1 GHz) et un microcontrôleur Cortex-M (240 MHz). Chacun inclut un sémaphore matériel ayant une architecture physique différente mais fournissant la même fonctionnalité. Ce sémaphore n'a pas été utilisé car par défaut le driver du coté

Linux est conçu pour une utilisation au niveau noyau. Il est plus simple d'utiliser le message d'accusé de réception pour assurer la synchronisation entre les tests successifs. Les mesures ont été extraits du côté de Linux, car pour lancer une session de débogage sur la carte i.MX8 nécessite une sonde JTAG (voir section 3.12) avec d'autres logiciels.

Afin de préparer l'environnement de test, nous avons développé deux applications exécutées chacune sur un cœur différent. La première envoie un message de 512 octets qui représentent la taille maximale du buffer AMP dans le pilote du protocole, celui-ci pourrait lors d'une migration de tâche correspondre au contexte de la tâche (variables locales et globales manipulées), à la deuxième partie qui continue l'exécution. Le timer externe utilisé pour la mesure est initialisé du côté du microcontrôleur, et il est possible de démarrer, arrêter ou lire la valeur de son compteur sur les deux processeurs et les mesures ont été répétés 50.000 fois. Le cluster à haute performance fonctionne sous Linux, patché avec PREEMPT_RT, avec une fréquence fixe, tandis que le microcontrôleur est utilisé en bare metal.

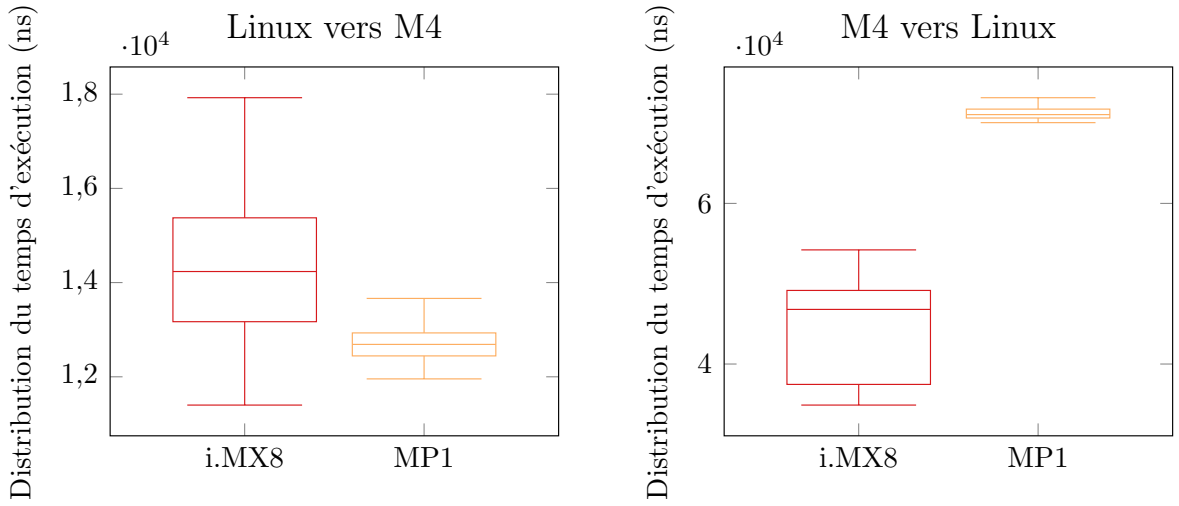

FIGURE 5.10 – Mesure du temps de communication d'une application AMP

Les résultats présentés sur la figure 5.10 montrent le temps d'exécution en nanosecondes d'envoi de messages utilisé par cette application. Nous pouvons remarquer, sur toutes les cartes, que l'envoi d'un message du microcontrôleur au microprocesseur (avec Linux installé) a une latence plus élevée. Le fait qu'il y ait plus de variation lorsque Linux est utilisé comme processeur droit peut s'expliquer par le fait que la tâche qui attend le message, bien qu'ayant la plus haute priorité, est dans un état d'attente. Lorsque le message arrive, il déclenche une interruption, qui déclenche les opérations du noyau Linux, qui vont préempter la tâche en cours d'exécution, puis appeler l'ordonnanceur. C'est pourquoi cette tâche est affectée par la variation de la latence du noyau. Lorsque Linux agit en tant qu'expéditeur, nous observons très peu de variation, car lorsque le compteur est activé par la tâche la plus prioritaire, qui alimente ensuite le pilote du mécanisme de communication inter-processeurs avec les données à envoyer, il n'y a pas de décision d'ordonnancement à prendre.

5.5 Migration hétérogène

5.5.1 Introduction

Actuellement, les ordonnanceurs ne supportent la migration que pour les systèmes symétriques SMP. Dans les systèmes AMP, ces ordonnanceurs sont confrontés à de nombreux défis, car ils doivent gérer des plateformes multiprocesseurs hétérogènes, qui ont généralement une ISA différente et parfois des mémoires centrales physiquement différentes. Néanmoins, permettre la migration pourrait conduire à une utilisation maximale des ressources, vu que des ordonnanceurs globaux optimaux pour les plateformes hétérogènes ont été proposés dans la littérature [12, 11]. Cependant, ces ordonnanceurs globaux considèrent des migrations et des préemptions instantanées. La solution d'implémentation de migration que nous avons retenue s'appuie donc sur un mécanisme de passage de contexte et de contrôle totalement logiciel, et doit être prévue dans le code. Pour effectuer une migration d'un processeur (que nous appellerons gauche) au processeur droit, un mécanisme proche du principe de mesure gauche-droite est employé. La tâche s'exécutant sur le processeur gauche collecte sous forme d'un tableau d'octets les états de toutes ses variables locales, avant de les transférer via OpenAMP au processeur droit. Celui-ci exécute une tâche qui est en attente sur l'arrivée de données OpenAMP. Celle-ci est réveillée par l'arrivée du message, peut dé-sérialiser les données reçues pour mettre à jour ses variables locales, avant de reprendre l'exécution de la tâche sur le processeur droit. Notons que l'on ne peut pas facilement transférer l'état d'une variable globale. Si une variable globale est utilisée par la tâche qui doit migrer, alors celle-ci doit être hébergée dans un espace mémoire partagé, et protégée par sémaphore matériel. Si plusieurs tâches sont appelées à migrer, alors il faut ajouter dans les données transférées un identifiant de tâche à activer sur arrivée du message, et dans ce cas, une fonction de distribution ira choisir en fonction de l'identifiant de la tâche à activer laquelle activer. Dans le cadre de cette thèse, nous nous sommes limités au cas ou une seule tâche est appelée à migrer dans un sens et dans l'autre.

Dans les sous-sections suivantes, nous étudions le coût de migration des tâches intra-cluster (SMP) et inter-cluster (AMP). Tout d'abord, nous étudions comment les migrations inter-clusters peuvent être effectuées en pratique et nous estimons le coût possible de ces migrations en mesurant le temps de communication requis pour partager les données. Ensuite, nous comparons les résultats avec les migrations intra-cluster dans un HMPSoC. Enfin, nous mesurons le temps d'exécution d'une tâche sur différents clusters de CPU.

Il faut noter que les méthodes présentées sont déployées pour donner un ordre de grandeur de l'impact des migrations dans une plateforme hétérogène. Elles ne visent pas à fournir un WCET sûr.

5.5.2 Méthode de migration hétérogène proposée

La figure 5.11 montre un exemple d'exécution d'une tâche sur deux cœurs asymétriques, où le cœur B est trois fois plus lent que le cœur A. La tâche a une période égale

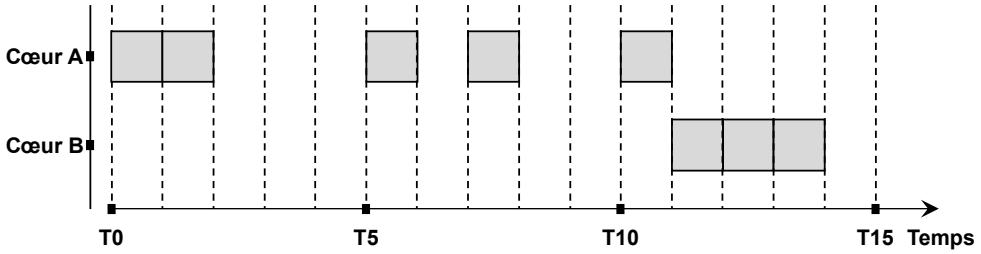

FIGURE 5.11 – Exécution d'une tâche sur des cœurs hétérogènes

à 5 unités de temps, elle a un temps d'exécution égal à 2 unités de temps sur le cœur A. Cependant, dans la troisième partie de la séquence d'exécution, la tâche commence son exécution sur le cœur A, est à moitié exécutée, puis migre jusqu'à terminer son exécution sur le cœur B. En effet, le cœur B est trois fois plus lent pour exécuter la tâche que le cœur A. Par conséquent, nous voyons que le temps de calcul restant sur B est de 3 unités de temps, alors que le temps de calcul sur A est égale à 1 unité de temps. De plus, dans cet exemple, nous n'avons pas tenu compte du temps de migration, qui n'est pas négligeable sur une plateforme réelle. Ensuite, nous supposons ici que le cycle se répète à T15 qui ramène à l'état dans lequel on se trouvait à T0. La tâche doit donc migrer sur le cœur A à la fin de son exécution sur B.

A l'intérieur d'un cluster, un système d'exploitation multicœur gère de manière transparente les migrations intra-clusters des tâches. Pourtant, la migration inter-clusters n'est pas supportée, du moins avec la technologie actuelle présente dans les HMPSoCs, et nécessiterait un effort de la part du développeur, à l'exception des plateformes hétérogènes partageant la même ISA comme ARM big.LITTLE présenté dans la section 3.8.

Ainsi, le compilateur devrait générer des exécutables différents en fonction des cœurs, non seulement en ce qui concerne les jeux d'instructions, mais aussi en ce qui concerne la communication et le contrôle. Ceci est présenté, en utilisant une API inspirée de POSIX, avec les pseudo codes 5.1 et 5.2. L'implémentation à générer suppose un ordonnancement statique calculé à l'avance, ici nous considérons l'ordonnancement de la tâche tel que sur la figure 5.11.

Sur le listing 5.2, la migration est effectuée comme suit : les variables locales des tâches sont sérialisées dans un tableau d'octets. Dans notre contexte le terme sérialisation représente la mise sous forme d'une suite d'octets (propices donc à être transférés) de données tout en assurant la gestion des éventuelles différences d'endianness, de l'alignement de la mémoire en fonction de l'architecture sous-jacente et des options de compilation, ou des différences de représentation des types. Ensuite, les données ainsi mises en forme sont transférées d'un cluster à un autre, en utilisant le protocole de communication inter-clusters disponible.

Afin d'estimer le temps de migration inter-clusters sur des plateformes réelles, nous avons donc mesuré le temps nécessaire à un cluster pour transférer 512 octets de données, ce qui pourrait correspondre à la sérialisation des variables locales d'une tâche de taille moyenne. Les tests sont effectués dans les deux sens, car comme nous pouvons le voir sur

les résultats, ils sont loin d'être identiques.

Les résultats affichés sur la figure 5.10 dans des boîtes à moustaches dont chacune représente les mesures sur une plateforme, soit du cluster rapide au cluster lent (graphique de gauche), soit du cluster lent au cluster rapide. Par exemple, le graphique de gauche indique que pour transférer 512 octets du cluster rapide au cluster lent sur un i.MX8, il faut entre 11,5 et 18 μs, avec une médiane de 14,1 μs, avec un quartile inférieur de 13,3 et un quartile supérieur de 15,5 μs. L'autre plateforme a montré une durée moins variable entre 12 et 14 μs. Ce résultat ne doit pas être utilisé pour comparer les deux plateformes, car il repose sur l'implémentation de la bibliothèque AMP. La communication entre le microcontrôleur et le cluster fonctionnant sous Linux est beaucoup plus longue, puisque sur une plateforme, elle prend de 30 à 55 μs, et environ 70 sur l'autre.

Listing 5.1 – Implémentation des points de migration logiciels partie 1

```
Task Original_tau_1 {
    Local variables declarations
    Initialization code
    for(;;) {
        wait activation
        First&Second half of the code
    }
}
```

Listing 5.2 – Implémentation des points de migration logiciels partie 2

```
Task tau_1_on_cluster1 {
    static unsigned count=0;
    static bool first=true;
    Local variables declarations
    Initialization code
    for(;;) {
        wait activation
        if (!first && count==0) {
            // migration from cluster2 to cluster1
            deserialize(wait_AMP_msg(),&local vars);
        }
        first=false;
        First half of the code
        if (count==2) {
            // migration from cluster1 to cluster2
            send_AMP_msg(serialize(local vars));
        } else {
            Second half of the code
            count=(count+1)%3;
        }
    }
}
Task tau_1_on_cluster2 {
    static unsigned count;
    Local variables declarations
    for(;;) {
        // migration from cluster1 to cluster2
        deserialize(wait_AMP_msg(),&local vars);
        Second half of the code
        count=(count+1)%3;
        // migration from cluster2 to cluster1
        send_AMP_msg(serialize(local vars));
    }
}
```

5.5.2.1 Coût de la migration inter- versus intra-cluster

Pour pouvoir comparer la migration inter-cluster et intra-cluster, nous avons réalisé quelques expériences sur les mêmes plateformes pour mesurer le temps de migration (intra-cluster) au sein d'un système Linux.

Du côté Linux, nous avons défini une fréquence CPU fixe, ce qui évite toute variation de fréquence due à la charge du processeur, et nous avons utilisé la priorité la plus élevée avec la politique d'ordonnancement sched_fifo.

La méthode de mesure utilisée est celle détaillée dans la section 4.5.2, elle est basée sur l'horloge POSIX CLOCK_MONOTONIC, qui est une horloge au niveau du système. Le temps d'exécution est égale à la différence entre la valeur de la fonction clock_gettime() avant et après la migration.

Nous avons mesuré 10.000 fois le temps de migration entre deux cœurs, dans un environnement stressé ou pas. Stresser l'environnement consiste à créer 50 processus de faible priorité qui communiquent à travers des sockets dans l'objectif de garder les cœurs occupés. Sur la partie gauche de la figure 5.12, nous pouvons voir que sur la carte i.MX 8, le temps de migration intra-cluster est inférieur à 1 μs, alors que pour le STM32MP1, il est toujours inférieur à 2 μs.

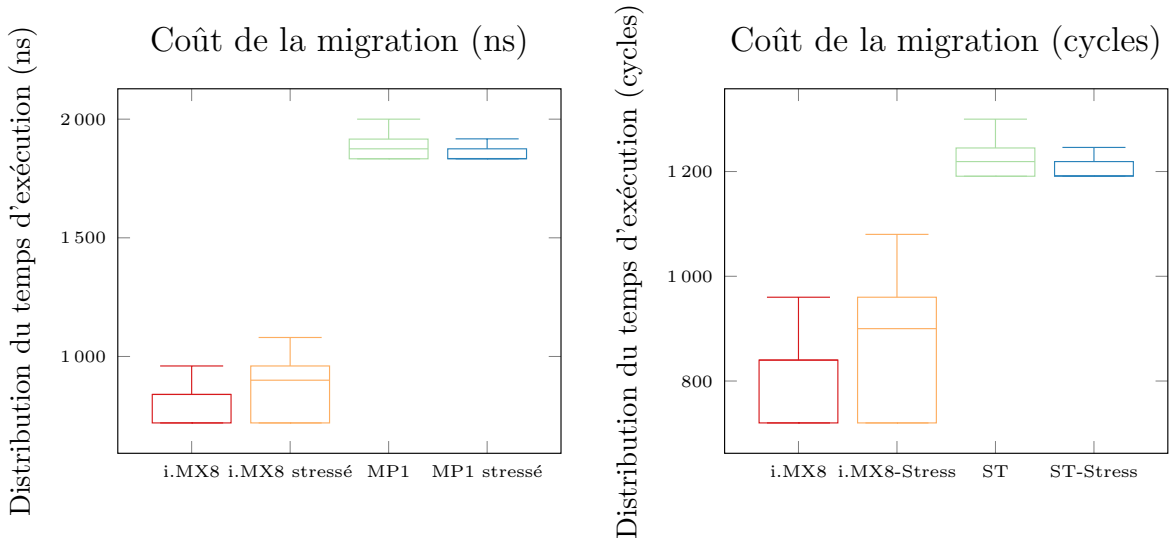

FIGURE 5.12 – Temps de migration intra-cluster

Cela montre que le coût d'une migration inter-clusters est, au minimum, entre 10 et 55 fois plus long sur l'i.MX 8, et entre 6 et 35 fois plus long sur le STM32MP1 que le coût d'une migration intra-clusters. Ces résultats montrent à quel point il peut être utile de différencier les deux types de migrations sur le modèle de plateforme HMPSoC. Cela justifie donc le modèle hiérarchique discuté dans le travail mené par Bertout, et al.[12] dans lequel nous distinguons deux modèles de plateformes : `à plat` et `multicœurs non liées`, illustrées dans la figure 5.13 partie gauche. Dans la littérature, une plateforme est souvent considérée comme `à plat`, ce qui signifie qu'elle ne distingue pas les différents clusters de cœurs hétérogènes. En outre, il n'y a pas de hiérarchie entre les cœurs et toutes les migrations sont considérées comme ayant le même coût. C'est une approche abstraite puisque la plupart des plateformes modernes sont composées d'un ou plusieurs clusters de cœurs, comme le montre la partie droite de la figure 5.13. Les cœurs des clusters sont identiques, mais peuvent différer d'un cluster à l'autre dans le cas de plateformes `multicœurs non liées`.

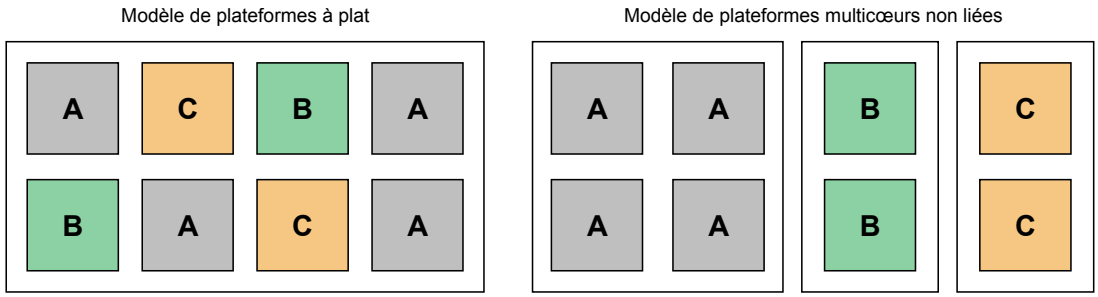

FIGURE 5.13 – Modèle de plateforme plate versus modèle de plateformes multicœurs non liées

5.5.2.2 La variation du temps d'exécution d'une tâche en fonction du type du cœur

Dans la littérature, les systèmes multiprocesseurs sont généralement classés en trois catégories [6] :

1. **Identiques :** tous les processeurs sont identiques et exécutent les tâches avec la même vitesse.
2. **Uniformes :** chaque processeur est caractérisé par une vitesse, par exemple, le processeur A exécute toute tâche deux fois plus vite que le processeur B.
3. **Non liées :** la vitesse de traitement dépend à la fois du processeur et de la tâche.

Nous avons mesuré le temps d'exécution de six programmes de test différents pour observer les caractéristiques d'une plateforme HMPSoC, la carte STM32MP1, sur les différents types de cœur disponibles. Nous considérons ici l'ordre de grandeur des temps d'exécution mesurés pour comparer les différents résultats et pas trouver le WCET. Nous avons mesuré 10000 fois sur un monocœur Cortex-A et 1000 fois sur le Cortex-M qui a montré beaucoup moins de variation de temps d'exécution.

La description des programmes de test suivants et la distribution de leurs temps d'exécution sont illustrées sur la figure 5.14.

- **BigNum** : Ce programme utilise la bibliothèque BigNum [60], qui implémente des opérations sur des entiers de grande taille représentés sous forme de tableaux. En fonction de ses paramètres, il peut nécessiter une puissance de calcul importante. Notre test `BigNum1` exécute 5 additions, `BigNum2` exécute 8 divisions, et `BigNum3` est plus intensif en calcul car il exécute 12 additions et multiplications. Lorsque le système Linux est stressé, nous observons des variations importantes des temps d'exécution. Par exemple, la boite à moustache pour `BigNum1` varie d'environ 37,2 μs à 37,4 μs sans stress, alors qu'il peut aller jusqu'à 38,6 μs sous stress. Le comportement montre beaucoup moins de variation sur le cœur Cortex-M baremetal, mais il est presque 4 fois plus lent que le cœur Cortex-A. Le même rapport, environ 4, peut être observé pour les trois tests BigNum.
- **N-body** : Il calcule le mouvement des planètes en utilisant un intégrateur symplectique [61]. Il commence par une courte phase d'initialisation, puis effectue des opérations intensives en virgule flottante pour déterminer les mouvements des planètes. Pour ce programme, nous observons un rapport supérieur à 10 entre les temps

d'exécution sur le cœur Cortex-A par rapport à l'exécution sur le cœur Cortex-M. Ceci est probablement dû à la présence d'un co-processeur ARM Neon sur les cœurs rapides. Ce rapport est proche de 15 fois pour les 50 itérations de ce programme, qui contient davantage d'opérations de calcul en virgule flottante.

- **FFT** : Transformée de Fourier rapide, qui nécessite des calculs intensifs avec des opérations en virgule flottante. De la même manière que pour `Nbody-50`, cette tâche est exécutée environ 15 fois plus vite sur un Cortex-A que sur un Cortex-M.

Ces tests ont montré que les cœurs les plus rapides ont toujours été capables d'exécuter tous les tests plus rapidement que les plus lents. Ils illustrent le fait que les composants d'accélération d'un processeur, qui peuvent avoir un impact sur le temps d'exécution, ne sont pas globaux sur les plateformes HMPSoCs, mais plutôt basés sur l'utilisation des tâches de certaines parties matérielles du processeur. Habituellement, un processeur plus rapide a tendance à être mieux équipé en unités matérielles supplémentaires qu'un processeur plus lent. Par exemple, l'ARM Neon n'était utilisé que par le Cortex-A, sans que le Cortex-M soit capable de l'utiliser pour effectuer des calculs en virgule flottante de manière plus rapide, mais il utilise son propre FPU qui est beaucoup plus lent. Ces résultats indiquent qu'une quatrième catégorie de systèmes multiprocesseurs devrait exister : l'`architecture consistante`. Il s'agit d'un cas particulier d'architecture non liée où l'hétérogénéité est consistante. Lorsqu'un processeur exécute une tâche plus rapidement que les autres processeurs, il est également plus rapide pour exécuter les autres tâches.

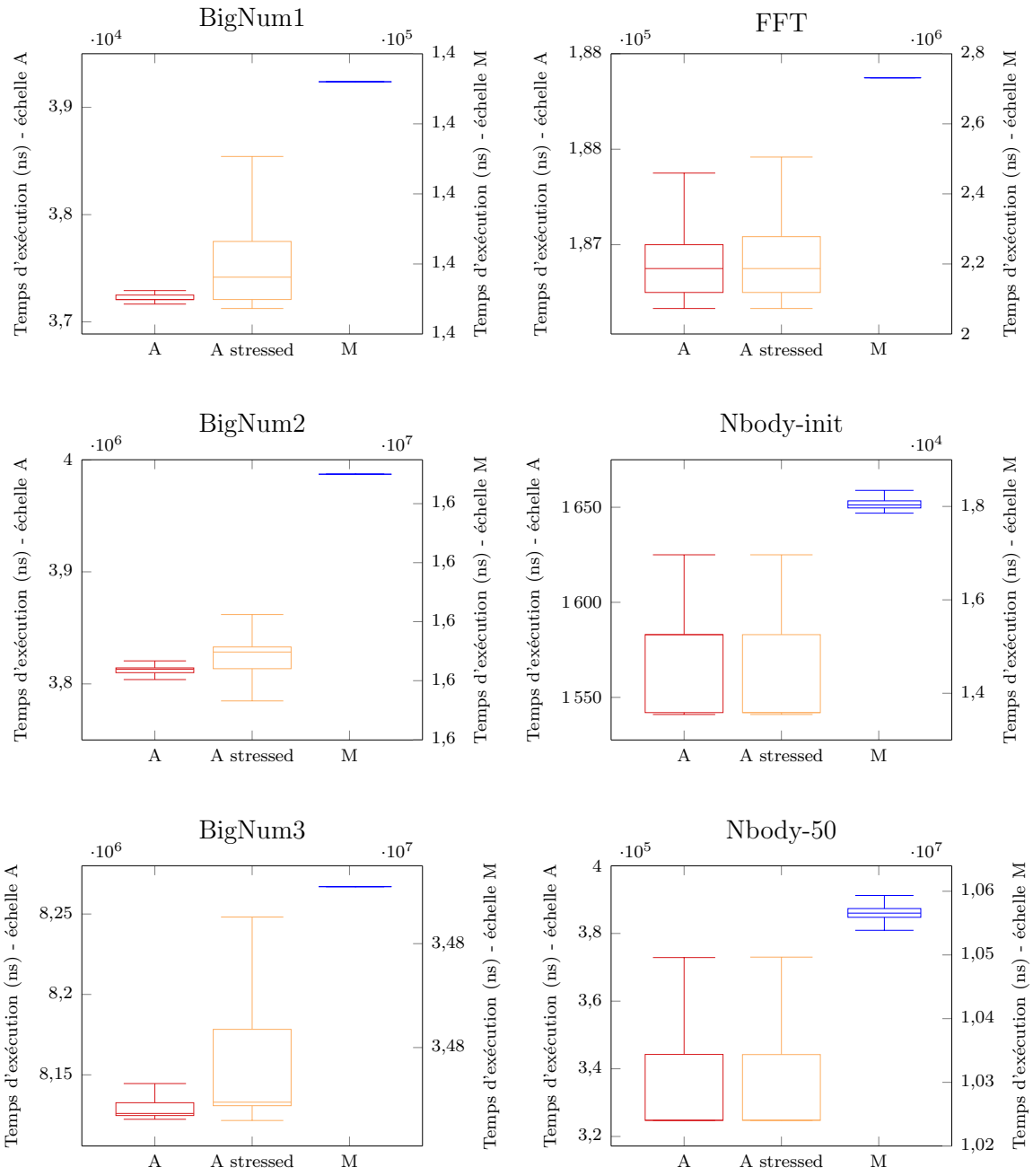

FIGURE 5.14 – Distribution du temps d'exécution de différents programmes

5.6 Étude de cas ROSACE

Cette section présente l'étude de cas aéronautique appelée ROSACE (Research Open-Source Avionics and Control Engineering) [86] qui met en œuvre un système de gestion de vol (FMS) pour drone, ainsi que la simulation de la dynamique de vol mettant à jour l'état physique du drone. Il sera utilisé pour évaluer le code multitâches sur des systèmes multiprocesseurs symétriques et asymétriques. Ce contrôleur de vol parallèle représente un FMS typique de drone. Nous avons utilisé l'implémentation C POSIX qui est composée de 5 tâches (pthreads) ayant la priorité la plus élevée avec la politique d'ordonnancement SCHED_RR, et les primitives pthread_barrier ont été utilisées pour les synchroniser. ROSACE peut être exécuté dans deux modes différents : l'un est l'exécution en mode temps réel, où les tâches ont une période et attendent la période suivante. L'autre est le mode simulation, où la tâche et le simulateur d'attitude sont synchronisés, mais exécutés au plus tôt, ce qui permet de simuler, en quelques secondes, le comportement du drone sur plusieurs minutes, voire plusieurs heures. Ce mode est bien sûr très intensif en termes de calcul est c'est celui qui a été utilisé pour les tests avec vingt mille itérations.

La carte STM32MP1-DK2, détaillée dans la section 3.9.1.1, est utilisée pour les tests, dans le cas SMP, seul le processeur d'application double cœur (Cortex-A7) est utilisé tandis que pour le cas AMP, le microcontrôleur (Cortex-M4) est aussi utilisé. Nous avons mesuré le temps d'exécution de toutes les tâches en utilisant la méthode Linux présentée précédemment dans la section 4.5.2 basée sur l'horloge POSIX CLOCK_MONOTONIC. La mesure commence lorsque la première tâche est créée et se termine lorsque toutes les tâches sont terminées. Afin d'éviter toute interférence des entrées/sorties avec la mesure, nous redirigeons également toutes les sorties vers le périphérique null. Nous utilisons `stress-ng` [57], une application utilisée pour stresser l'environnement Linux en créant une forte charge sur le CPU. Chaque tour de mesure consiste à exécuter en parallèle, l'outil de stress pendant 60 secondes et le programme ROSACE. Ce processus est répété des centaines de fois. Deux versions de ROSACE ont été utilisées :

- `ROSACE SMP` représente la version par défaut de ROSACE, où cinq tâches sont exécutées sur les deux processeurs d'application sous Linux. Les tâches migrent régulièrement entre les cœurs ce qui est géré par le noyau Linux.
- `ROSACE AMP` est une version modifiée où une des tâches s'exécute sur le microcontrôleur en utilisant la méthode de migration hétérogène présentée dans la section précédente 5.5. Le point de migration se produit après le mécanisme de synchronisation de Linux : un message est envoyé du système Linux au microcontrôleur, contenant l'index de la fonction qui doit être exécutée et représentant le code principal de cette tâche. Pendant ce temps, le processeur peut sélectionner une autre tâche et l'exécuter. Le microcontrôleur est utilisé en baremetal.

Dans les sous-sections suivantes, nous discutons plus en détail l'exécution des tâches sur SMP et AMP, afin de les comparer. L'idée est de montrer quand l'utilisation d'AMP a un avantage en termes de temps d'exécution par rapport à l'utilisation de SMP. Ces

tests sont effectués sous deux environnements différents, une charge normale (c'est-à-dire que la plateforme n'exécute que le programme ROSACE) et un environnement stressé.

5.6.1 Charge normale

Dans cet environnement, nous testons les deux versions de ROSACE sous la charge normale de Linux où seuls les tâches d'arrière-plan sont exécutées avec un impact négligeable sur la plateforme. La figure 5.15 représente une exécution typique d'une instance des cinq tâches dans les deux cas. Dans le cas SMP, les cinq tâches sont exécutées sur les cœurs disponibles. Dans les cas SMP et AMP, une barrière est utilisée pour que les tâches attendent les unes les autres, avant de commencer l'instance suivante. Nous utilisons CLOCK_MONOTONIC pour mesurer le temps de réponse de l'ensemble de la simulation de vingt mille de ces instances. Dans le cas AMP, l'exécution d'une tâche sur le microcontrôleur nécessite deux petites portions de code à exécuter sur le programme Linux. La première contient la synchronisation avec les autres tâches (pthread_barrier), en plus de l'envoi du message. La seconde consiste à lire le message traité et à donner l'information que l'exécution de cette tâche est terminée sur le microcontrôleur. Par rapport à la version SMP, le temps d'exécution de cette tâche est beaucoup plus long, ce qui est dû au fait que le microcontrôleur est beaucoup plus lent car il fonctionne à une fréquence plus basse et ne possède pas de FPU. La latence de migration associée à un cœur plus lent a entraîné un temps de réponse global plus long de ROSACE.

En regardant la figure 5.16, nous voyons que l'application SMP est plus rapide que l'application AMP d'une seconde, le temps d'exécution médian étant respectivement d'environ 4 secondes contre 5 secondes. Même le pire temps de réponse observé en SMP (4,2 secondes) est plus rapide que le meilleur temps de réponse observé en AMP (4,4 secondes).

5.6.2 ROSACE stressé

Nous avons stressé la plateforme Linux en générant une charge CPU constituée de 20 tâches qui partagent l'exécution avec le programme ROSACE Linux selon la politique d'ordonnancement round robin. Cet environnement stressé a un impact important sur le temps de réponse des deux applications, n'affectant que le côté Linux, laissant le microcontrôleur intact, ce qui est le principal avantage dans la situation AMP où nous pouvons noter une performance globale légèrement meilleure. Ceci peut être illustré sur la figure 5.17, où nous voyons que dans ce cas, l'interférence générée par le stress augmente le temps de réponse sur la plateforme SMP.

Sur la figure 5.18, l'application AMP est légèrement plus rapide que l'application SMP lorsqu'on regarde la médiane (33 secondes contre 36 secondes). Pourtant, le pire temps de réponse observé pour la version SMP (37 secondes) est inférieur au pire temps de réponse observé pour la version AMP (40 secondes). Ceci est dû à la nature du noyau Linux ainsi qu'au mécanisme de transmission de messages qui est très affecté par l'environnement

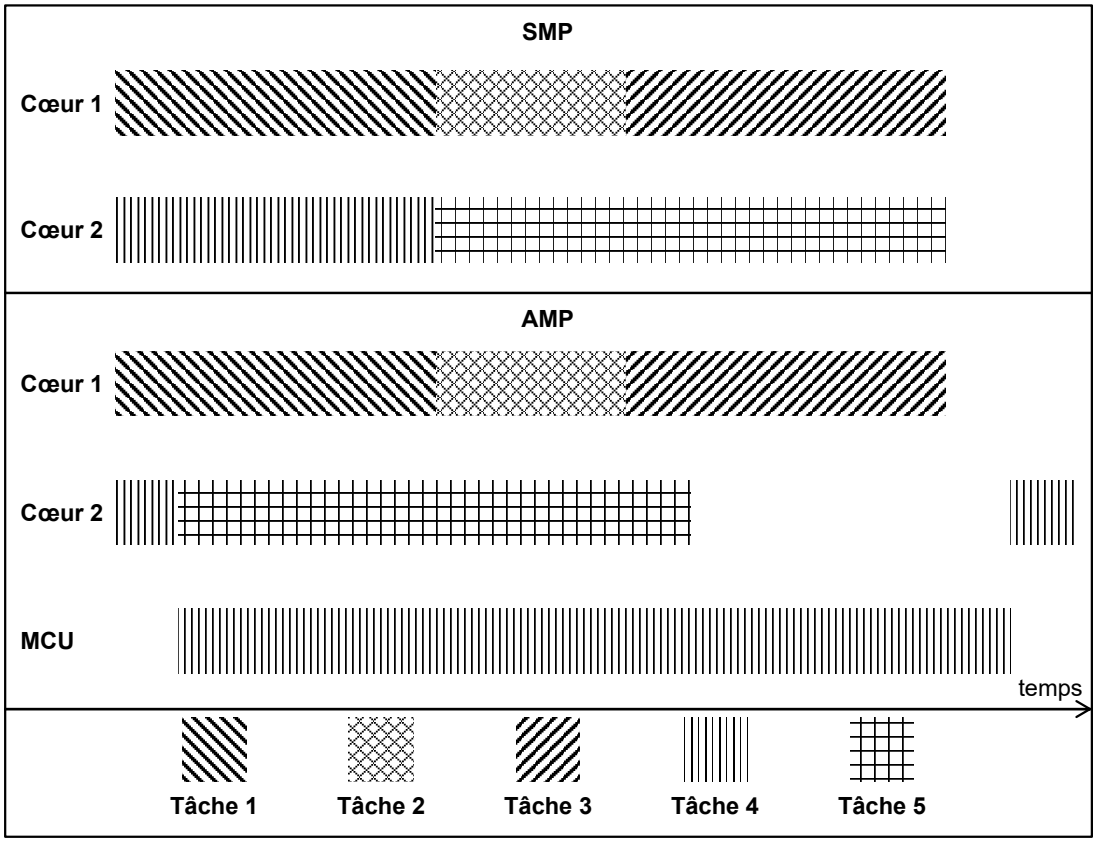

FIGURE 5.15 – Temps d'exécution du ROSACE en charge normale

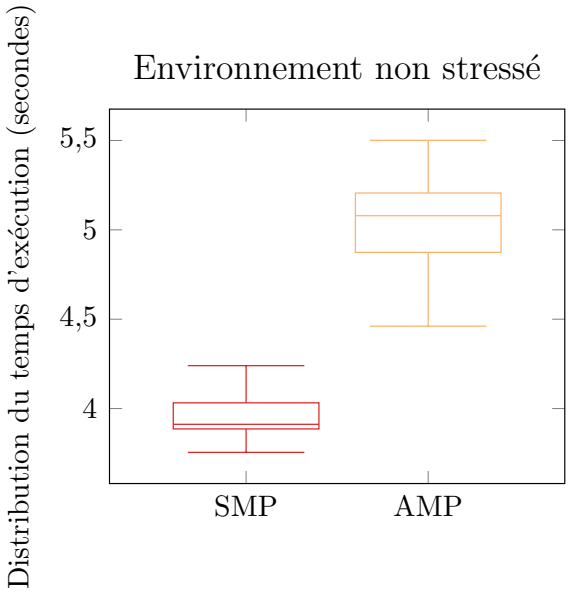

FIGURE 5.16 – Mesure du temps d'exécution de ROSACE en charge normale

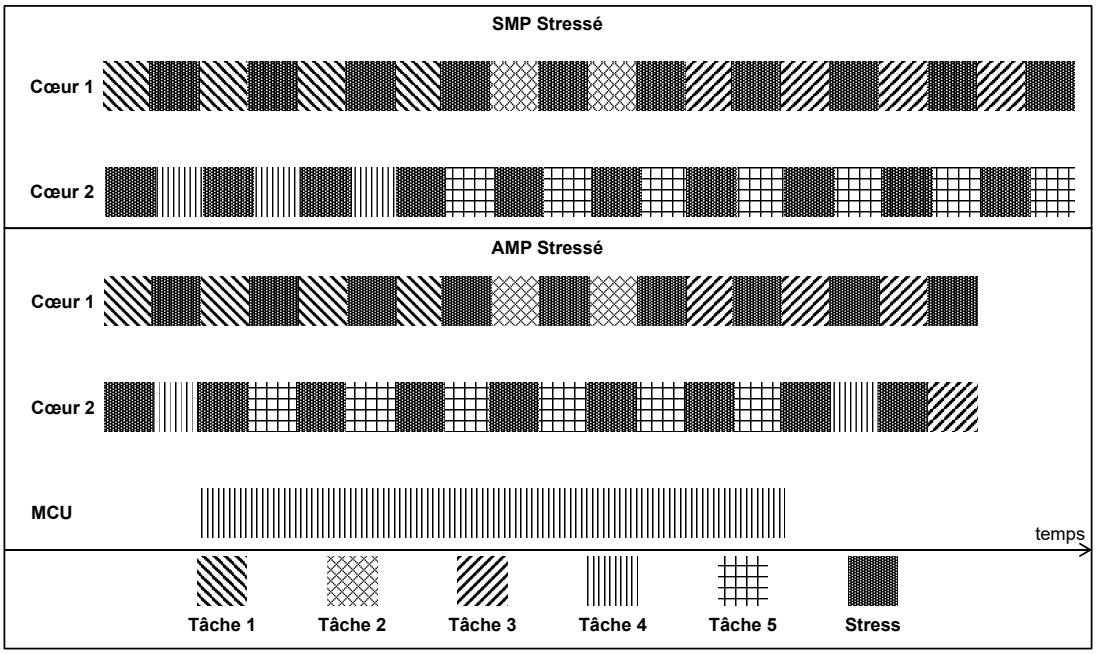

FIGURE 5.17 – Temps d'exécution de ROSACE dans un environnement stressé

stressé.

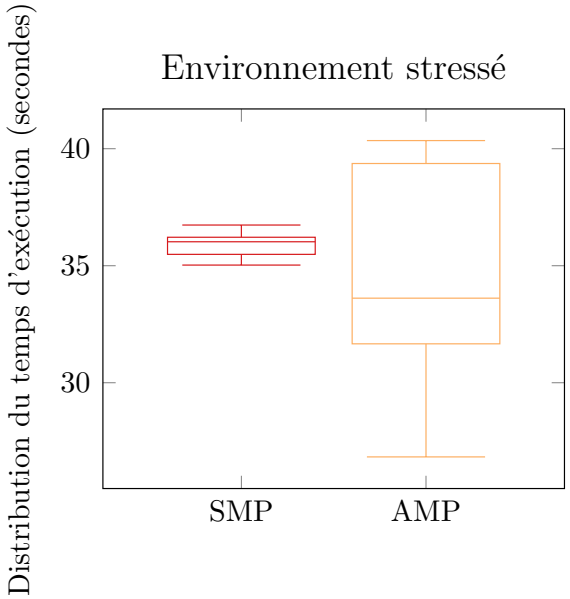

FIGURE 5.18 – Mesure du temps d'exécution de ROSACE dans un environnement stressé

En effet, l'une des difficultés que nous avons rencontrées est que Linux implémente la communication AMP en tant que pilote du noyau, et les opérations d'entrée/sortie peuvent être perturbées par une utilisation intensive du processeur par des tâches temps réel. Donc, lorsque les tâches temps réel occupent un cœur, les opérations d'entrée/sortie émises, y compris par les tâches temps réel, sont retardées.

5.7 Discussion et perspective

Les ordonnanceurs actuels ne profitent pas des microcontrôleurs disponibles dans les systèmes hétérogènes asymétriques et ne les utilisent jamais pour supporter le traitement hétérogène des tâches. En se basant sur ce que nous avons vu dans ce chapitre ceci est logique, car la migration des tâches n'est pas simple, principalement à cause des différentes ISA. La migration de tâches hétérogènes est donc nécessaire. La méthode de migration suggérée dans la section 5.5 a montré qu'elle n'est pas toujours avantageuse. La raison principale étant le coût élevé de la migration. En effet, en plus de l'installation des points de migration, il est nécessaire de configurer les mécanismes de communication et de réécrire le code de manière à ce que l'application s'attende à envoyer et recevoir des messages dans les tâches pour permettre la migration, il est aussi nécessaire de se baser sur un ordonnancement statique afin de savoir à quels endroits la tâche peut être amenée à migrer.

Il serait intéressant de créer un ordonnanceur qui permette la migration hétérogène, ceci simplifierait au moins le travail du développeur. Il faudrait mettre en place au préalable les points de migration logiciels dans le code, par conséquence c'est l'ordonnanceur qui déciderait d'effectuer la migration de cette tâche selon les points instrumentés en utilisant le protocole de communication sur lequel la migration hétérogène est basée. L'ordonnanceur principal serait implémenté sur un cluster maître qui prendrait les décisions de migration des tâches vers d'autres clusters ou cœurs esclaves. Ces derniers devraient inclure un logiciel ou une bibliothèque permettant la migration et capable de communiquer avec l'ordonnanceur.

Comme nous l'avons montré expérimentalement, si le cluster principal n'est pas très chargé, le coût de la migration et l'exécution du code sur un processeur plus lent (surtout si le code inclut beaucoup de calculs flottants) rendra le système moins efficace et moins performant, tout en ayant un code et un ordonnanceur très sophistiqués.

La migration hétérogène des tâches ne devrait donc être autorisée que si ces trois conditions sont remplies :

1. Le microcontrôleur n'est pas utilisé pour exécuter une tâche spécifique.
2. La tâche à migrer comprend des points de migration AMP et le système respecte la solution présentée dans la section 5.5.
3. Ne migre que lorsque le système est très chargé.

En fonction de ces conditions, l'ordonnanceur peut utiliser le système de manière plus efficace. Une autre option intéressante à ajouter serait d'attacher une tâche à un cœur spécifique, et lorsque cette tâche n'est pas sélectionnée, il serait possible d'utiliser le cœur pour ordonnancer les tâches migrées. Cette option a l'avantage de simplifier l'utilisation du système dans certaines applications, principalement celles exigeant un système sécurisé. Dans ce cas, ce cœur sera un système de secours, en cas de défaillance des autres clusters et cœurs. Il restera disponible pour assurer les tâches minimales requises afin d'éviter des conséquences catastrophiques. Ce cœur de secours sera également capable d'essayer de

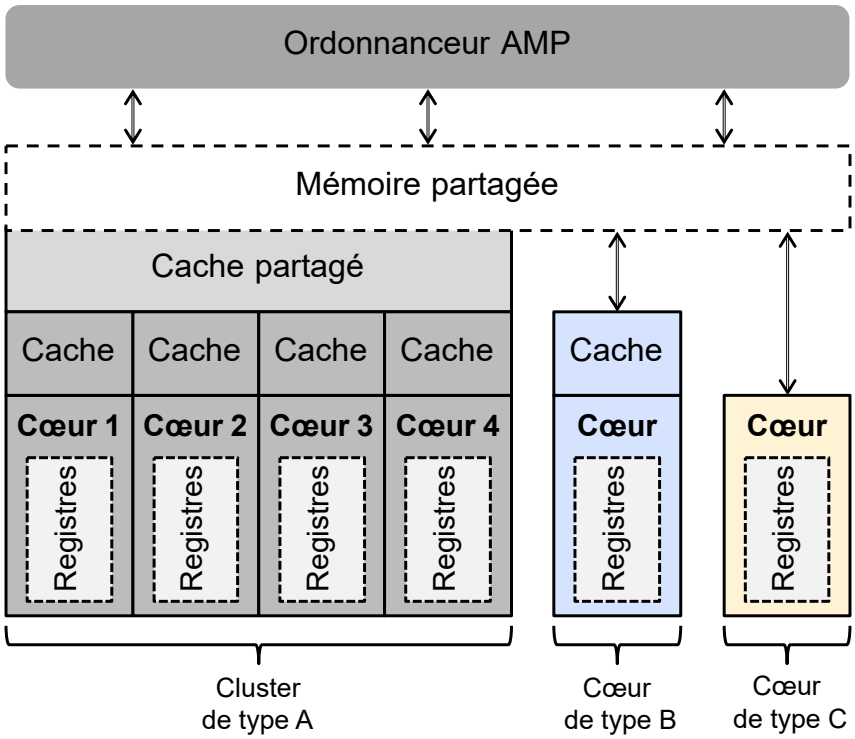

FIGURE 5.19 – Système HMPSoC avec un ordonnanceur AMP

récupérer le système en essayant par exemple de redémarrer les autres clusters.

La figure 5.19 montre un système HMPSoC qui a trois différents types de cœurs, les premiers du même type sont regroupés dans un cluster et les autres chacun seul. Dans cet exemple l'ordonnanceur AMP devrait décider quand migrer une tâche inter ou intra cluster. La décision de migration inter cluster devrait être basée soit sur une indication explicite soit sur les trois conditions citées ci-dessus.

5.8 Conclusion

Dans ce chapitre, nous avons vu la structure d'une application AMP qui utilise tous les processeurs disponible dans les HMPSoCs, ainsi que le mécanisme de communication AMP et les ressources partagées.

Ce type de systèmes comporte de nombreux défis pour l'analyse du temps d'exécution, en raison de l'architecture complexe du matériel. Les méthodes basées sur la mesure sont plus simples à mettre en œuvre que les méthodes d'analyse statique. Les méthodes dépendantes du cœur ne peuvent pas être utilisées à cette fin car elles ne sont pas accessibles par d'autres parties du système. Il a été observé que les cœurs hétérogènes sur le même HMPSoC ne partagent pas les accélérateurs, ce qui indique la nécessité de les placer dans une nouvelle catégorie de multicœurs : `architecture consistante`, dans laquelle le processeur rapide peut généralement exécuter toutes les tâches plus rapidement que le processeur le plus lent. Plusieurs méthodes de mesure ont été comparées, et deux retenues, de façon à obtenir un ordre de grandeur du temps d'exécution de la communication AMP.

Elle est basée sur le timer à usage général accessible par les différents cœurs et qui est souvent disponible sur tous les SoCs récents.

Une méthode de migration hétérogène a été présentée, qui permet à une tâche de migrer entre des cœurs ayant des ISA et des fréquences différentes. Cette méthode est basée sur des points de migration dans l'application, où les données locales sont transférées en utilisant le protocole de communication AMP. Le temps de latence de communication a été mesuré à l'aide de la méthode de mesure proposée, et il a été constaté qu'il n'est pas négligeable et varie beaucoup d'un cœur à l'autre. De plus, cette migration a également été comparée à la migration intra-cluster et il a été observé qu'elle a une latence beaucoup plus élevée, d'où la nécessité différencier les deux types de migrations sur le modèle de plateforme HMPSoC, ainsi que d'adopter un modèle hiérarchique dans lequel les cœurs de même type seront groupés dans des clusters.

Enfin, une étude de cas basée sur ROSACE a été réalisée, dans laquelle nous avons comparé le temps d'exécution en utilisant uniquement le cluster SMP à celui utilisant le cœur hétérogène (plus lent). Il a été observé que le coût de la migration entraînera un temps d'exécution global plus long, ce qui signifie un système moins efficace. Ensuite, nous l'avons comparé avec un scénario où le cluster de processeurs rapides étaient stressé et très chargé, et dans ce cas, un léger avantage a été observé.

En conclusion, bien que, en théorie, l'ordonnancement temps réel global permette une totale utilisation de la plateforme, en pratique, celui-ci est complexe à mettre en œuvre. Dans le cas où il n'y a pas besoin d'un processeur dédié pour une application spécifique, ce processeur peut être utilisé grâce à la migration hétérogène, mais il doit respecter certaines conditions.

Chapitre 6

Environnement de développement unifié

Sommaire

6.1 Introduction 141
6.1.1 Les éléments d'une distribution Linux embarqué 141
6.1.2 Processus de démarrage d'un système basé sur Linux 143
6.1.3 Construction automatisée d'une distribution Linux 145
6.2 SW4Linux 148
6.2.1 Aperçu 150
6.2.2 Sécurité et démarrage sécurisé 151
6.2.3 Développement des pilotes Linux 152
6.2.4 Plugin de mesure de temps d'exécution 154
6.3 Outils de développement des microcontrôleurs 157
6.4 Environnement de développement unifié 158
6.4.1 Développement d'applications AMP 159
6.4.2 Débogage des applications AMP 160
6.5 Conclusion 161

6.1 Introduction

Suite aux récentes avancées technologiques concernant les SoCs, la manière de développer les applications embarquées a changé. On distingue trois types de développement :

- `Le développement du système` qui consiste à créer, personnaliser, configurer et maintenir la distribution du système d'exploitation.
- `Le développement de l'application` qui consiste à créer l'application ciblée.
- `Le développement de firmware ou de noyau` qui est le développement de la couche la plus proche du matériel. Il est nécessaire de s'assurer que le code écrit est sûr et fiable car toute erreur à ce niveau peut planter tout le système. Sous Linux, cela signifie généralement le développement de pilotes ou de modules du noyau.

Les distributions basées Linux sont largement utilisées sur les processeurs à haute performance. Un système de *build*, constitué d'un ensemble d'outils permettant de configurer, compiler et télécharger un ensemble de logiciels, est nécessaire afin de construire une distribution personnalisée adaptée à une application spécifique, un exemple de ce système de *build* est Yocto [98]. En général, chaque type de développement nécessite un outil différent, et même parfois plusieurs outils pour le même type.

Sur les petits systèmes embarqués comme ceux qui sont basés sur un microcontrôleur, ces trois types de développement sont parfois plus simples à gérer ; un ou quelques outils sont généralement utilisés. Dans un environnement plus complexe, en particulier lorsque Linux est utilisé sur les microprocesseurs, de nombreux outils sont nécessaires, le cas est plus complexe lorsqu'il s'agit des HMPSoCs, étant donné que Linux est généralement utilisé et que ces systèmes sont confrontés à de nombreux défis à tous les niveaux du développement. Ce chapitre les présente ainsi que la solution SW4Linux à laquelle cette thèse contribue.

Ac6, société spécialisée dans le développement des *Integrated Development Environment* (IDE)s, a créé SW4Linux qui vise à unifier dans un seul environnement ces trois types de développement. Il peut être utilisé pour créer une distribution Linux, pour développer des applications compatibles avec celle-ci, ainsi que le développement de pilotes. Ma mission était d'ajouter plus de fonctionnalités à cet outil afin de le rendre compatible avec les HMPSoCs et en particulier ceux qui sont basés sur l'architecture ARM. Nous présenterons dans ce chapitre comment SW4Linux permet de créer un environnement de développement unifié pour les puces asymétriques. De plus, j'ai ajouté une fonctionnalité, qui permet de mesurer facilement le temps d'exécution d'une fonction en C ou C++ et d'extraire automatiquement les données mesurées, en créant un nouveau plugin. Ce dernier a été utilisé pour effectuer les mesures dans les chapitres précédents.

6.1.1 Les éléments d'une distribution Linux embarqué

Une distribution Linux est une collection de packages (des logiciels, aussi appelés paquets ou paquetages) et de règles. Il existe des centaines de distributions Linux disponibles,

la plupart d'entre elles ne sont pas conçues pour les systèmes embarqués. Elles ne sont pas adaptées aux systèmes à ressources limitées et manquent de certaines fonctionnalités telles que le contrôle du matériel. Un système Linux comporte plusieurs types d'éléments logiciels. Ils sont utilisés dès que le processeur est mis sous tension, jusqu'à la fin du processus de démarrage. Les principaux éléments sont :

- `Chaîne de compilation` : Aussi appelé toolchain, elle est formée du compilateur et les autres outils nécessaires pour créer des programmes pour la cible comme l'assembleur et l'éditeur de liens. Tous les autres éléments du système Linux dépendent d'elle. Elle doit être compatible à la fois avec le matériel et le système d'exploitation utilisé. De nombreuses chaînes de compilation sont disponibles, mais pour un système basé sur Linux, il est préférable d'utiliser celle basée sur GNU, car elle est la seule à pouvoir compiler un noyau Linux. La chaîne de compilation peut être native ou croisée comme détaillé dans la section 3.11. Lors de la création du système, le compilateur utilisé peut être pre-compilé en format exécutable (comme celui fourni par Linaro) ou peut être compilé à partir du code source ; ceci peut être fait manuellement en utilisant crosstool-ng [32] ou automatiquement en utilisant un des systèmes de *build* que nous allons voir dans ce chapitre.
- `Chargeur d'amorçage` : Aussi appelé *bootloader*, c'est le tout premier logiciel qui s'exécute au démarrage du système, il est généralement constitué de plusieurs éléments : "Boot ROM", le chargeur d'amorçage de première étape et le chargeur d'amorçage de deuxième étape. Son rôle principal est d'initialiser la carte, y compris la RAM et certains périphériques, et à la fin de charger et de démarrer le noyau Linux. Le chargeur d'amorçage (ou plus précisément, dans certains cas, le chargeur d'amorçage de deuxième étape) est un petit noyau qui est capable d'exécuter certaines commandes, lire la mémoire, accéder au système de fichiers, et même se connecter au réseau. Le chargeur d'amorçage le plus utilisé pour les systèmes embarqués basés sur Linux est u-boot [121].
- `Noyau` : Aussi appelé *kernel*, il représente le cœur du système, il a plusieurs rôles comme la gestion des ressources et l'interface avec le matériel. Ses principaux éléments sont l'ordonnanceur, le gestionnaire du système de fichiers, les pilotes des différents périphériques, la gestion de la mémoire et le support réseau. Linux fournit un code source libre très configurable, qui doit être compilé pour créer un fichier exécutable en format binaire (le noyau). Il fournit également le "device tree" qui sert à configurer les périphériques, ainsi que les modules du noyau, qui représentent des fichiers qui peuvent être chargés à partir du système de fichiers et exécutés au même niveau que le noyau.
- `Système de fichiers` : aussi appelé *root filesystem* ou *rootfs*. C'est un espace qui contient les bibliothèques et les programmes qui sont exécutés une fois que le noyau a terminé son initialisation. Il possède tout d'abord un format de système de fichiers spécifique, généralement `ext4`. Il doit respecter l'arborescence des dossiers Linux et contenir au moins quelques programmes requis, le plus important étant le point d'entrée, en règle générale appelé "init", qui peut exister sous différents formats

FIGURE 6.1 – Séquence de démarrage des systèmes embarqués

comme systemd (le plus utilisé), system V ou busybox [103]. Le reste des programmes indispensables peuvent être fourni grâce à l'outil busybox qui peut intégralement les créer. Le système de fichiers contient également les modules du noyau et les fichiers de configuration les plus importants.

Un système Linux complet doit également comporter un autre élément : la collection de programmes spécifique à l'application embarquée qui permet au système de faire ce qu'il est censé faire, comme par exemple pour un drone, le programme du traitement d'images et l'autopilote. Cette collection est formée d'au moins des dizaines de logiciels et de bibliothèques qui ont une relation de dépendance très sophistiquée. Ces applications (dont la plupart en open source) doivent être compilées à l'aide d'une chaîne de compilation compatible avec la cible, où chacune a une méthode de configuration et de paramétrage spécifiques. En outre, elles ont des normes de configuration et de compilation différentes (même parfois sans norme et doivent être construites en utilisant une méthode spécifique), sans compter le temps de *build* non négligeable. Tout cela fait de cette partie l'une des plus difficiles dans le processus de génération d'une distribution embarquée personnalisée. En prenant tout cela en compte, la création d'une distribution Linux embarquée, à part le développement du code, n'est pas simple et comprend de nombreuses étapes, comme trouver les logiciels compatibles (y compris les logiciels essentiels pour le démarrage du système, les applications ou les bibliothèques nécessaires), puis compiler chacun d'entre eux, ensuite gérer le problème des dépendances, et enfin reconfigurer le système. C'est pourquoi il existe des systèmes de *build* qui peuvent être utilisés pour automatiser la construction et la génération de cette distribution personnalisée, comme Yocto et Buildroot [98], mais chacun avec ses propres avantages et inconvénients.

6.1.2 Processus de démarrage d'un système basé sur Linux

Les systèmes basés sur Linux ont plusieurs façons de démarrer, mais en premier lieu, il est nécessaire d'initialiser certains éléments du système avant d'exécuter le noyau, et ceci est le principal rôle du chargeur d'amorçage, qui a également besoin de certains éléments basiques initialisés comme la mémoire externe. Sur les systèmes embarqués récents, comme illustré sur la figure 6.1, les premières lignes de code exécutées proviennent du "Boot ROM", suivi par le chargeur de programme secondaire (SPL) qui lance le chargeur d'amorçage (généralement u-boot sur les systèmes embarqués). Ensuite, le chargeur d'amorçage, grâce à ses fonctionnalités comme la gestion des fichiers et l'accès au réseau, exécute le noyau qui, à son tour, monte la partition racine et entre dans le système de fichiers en exécutant le programme "init".

Après la mise sous tension, le système entre dans un état minimal. Le contrôleur

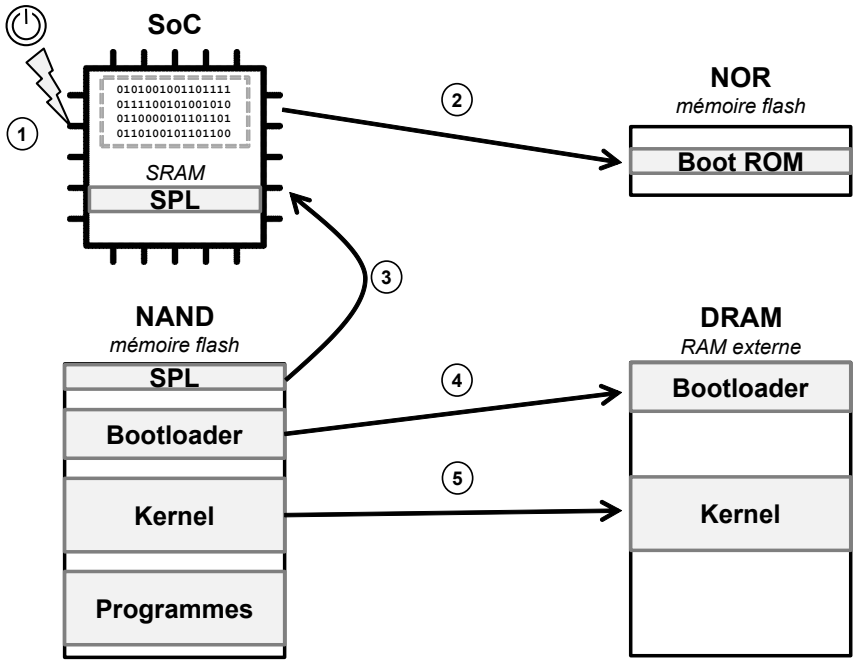

FIGURE 6.2 – Étapes de démarrage du système Linux

de RAM externe (DRAM) n'a pas été configuré donc la mémoire principale n'est pas accessible, de même que les autres interfaces ne sont également pas configurées pour que le stockage soit accessible via les contrôleurs flash de type "NAND". En général, les seules ressources opérationnelles au départ sont un cœur de processeur et un peu de mémoire statique sur la puce. Par conséquent, les étapes de démarrage se composent de plusieurs phases, chacune mettant en service une partie plus importante du système. Comme le montre la figure 6.2, après la mise sous tension, le "Boot ROM" est exécuté sur place, ceci est possible car il est placé dans la mémoire flash de type "NOR" qui permet l'exécution sur place (XiP) [10], c'est une mémoire très lente ; c'est pourquoi le "Boot ROM" est de très petite taille et ne contient que les instructions minimales comme la configuration de certaines horloges système et le chargement du SPL à partir d'un espace mémoire spécifique, pour à la fin l'exécuter.

Ensuite, le SPL doit configurer le contrôleur de mémoire externe et d'autres parties essentielles du système avant de charger le chargeur d'amorçage dans la mémoire principale (DRAM). La fonctionnalité du SPL est limitée par sa taille. Par exemple, il ne permet généralement pas d'interaction avec l'utilisateur. Il peut éventuellement contenir certains éléments de démarrage sécurisé si nécessaire. Le chargeur d'amorçage est exécuté une fois qu'il est chargé dans la mémoire par le SPL.

Enfin, un chargeur d'amorçage comme U-Boot est lancé. Habituellement, il existe une interface utilisateur en ligne de commande simple, permettant d'effectuer des tâches de maintenance telles que le chargement et le démarrage d'un noyau, mais il existe également un moyen de charger le noyau automatiquement sans intervention de l'utilisateur. À la fin de cette étape, il y a un noyau chargé dans la mémoire qui attend d'être démarré. Lorsque le chargeur d'amorçage donne le contrôle au noyau, il doit également transmettre

certaines informations telles que les détails sur le matériel détecté, y compris au moins la taille et l'emplacement de la RAM physique, la fréquence de l'horloge du système et, optionnellement, l'emplacement du fichier binaire "device tree" si utilisé.

Le démarrage sécurisé est un processus dans lequel les images et le code de démarrage du système Linux sont vérifiés et authentifiés par le matériel avant d'être utilisés dans le processus de démarrage. Pour ce faire, le matériel doit être configuré au préalable pour activer l'authentification des images. Cela garantit que seuls les logiciels crypto-authentifiés (chargeur d'amorçage, noyau, etc.) pourront s'exécuter sur une cible donnée.

6.1.3 Construction automatisée d'une distribution Linux

La création d'un système Linux manuellement a l'avantage de pouvoir contrôler entièrement les logiciels et de les personnaliser. C'est une bonne option dans le cas de systèmes basiques et lorsqu'il est nécessaire de réduire au maximum l'empreinte mémoire. Mais, dans la grande majorité des situations, le créer manuellement prend beaucoup de temps et produit des systèmes de moindre qualité et difficiles à maintenir.

Le principe d'un système de *build* est d'automatiser toutes les étapes détaillées dans la section 6.1.1. Il doit être capable de construire, à partir du code source, tout ou en partie les éléments du système Linux : chaîne de compilation, chargeur d'amorçage, noyau et système de fichiers. Un système de *build* doit être capable au minimum de :

- Télécharger le code source en ligne ou local en supportant leurs différents formats (tar.bz2, git etc...).
- Appliquer des patchs (aussi appelés correctifs).
- Effectuer une compilation croisée
- Gérer les dépendances entre les packages.
- Créer des images sous différents formats prêtes à être installées sur la cible.

6.1.3.1 Buildroot

Buildroot est un outil libre sous licence GPLv2, disponible en ligne, qui est capable de construire une chaîne de compilation, un chargeur d'amorçage, un noyau, et un système de fichiers. Il est généralement destiné à la création d'une petite et simple distribution Linux embarquée. Il reconstruit tout à partir du code source et télécharge automatiquement tous les packages nécessaires. La sortie peut être personnalisée par un outil de configuration hérité du noyau Linux, soit en mode textuel avec menuconfig comme le montre la figure 6.3, soit avec une interface graphique encombrante (xconfig ou gconfig) qui permet seulement de définir les valeurs de certaines variables.

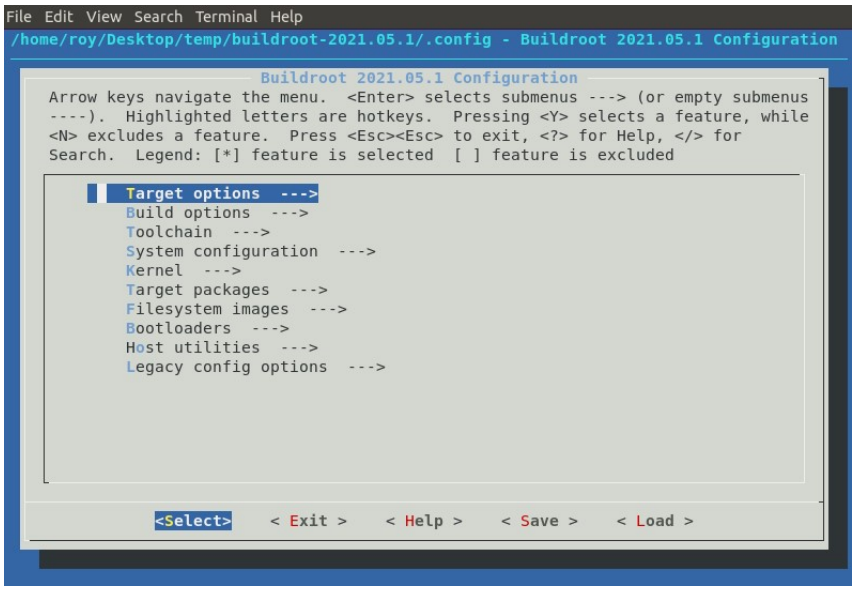

FIGURE 6.3 – Menu de configuration de Buildroot

Listing 6.1 – Exemple de fichier de configuration de Buildroot (config.in)

```
LIBFOO_VERSION = 1.0
LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
LIBFOO_LICENSE = GPLv3+
LIBFOO_LICENSE_FILES = COPYING
LIBFOO_CONFIG_SCRIPTS = libfoo-config
LIBFOO_DEPENDENCIES = host-libaaa libbbb

define LIBFOO_BUILD_CMDS
    $(MAKE) CC="$(TARGET_CC)" LD="$(TARGET_LD)" -C $(@D) all
endef

define LIBFOO_INSTALL_STAGING_CMDS
    $(INSTALL) -D -m 0644 $(@D)/foo.h $(STAGING_DIR)/usr/include/foo.h
    $(INSTALL) -D -m 0755 $(@D)/libfoo.so* $(STAGING_DIR)/usr/lib
endef

define LIBFOO_INSTALL_TARGET_CMDS
    $(INSTALL) -D -m 0755 $(@D)/libfoo.so* $(TARGET_DIR)/usr/lib
    $(INSTALL) -d -m 0755 $(TARGET_DIR)/etc/foo.d
endef

define LIBFOO_PERMISSIONS
    /bin/foo f  4755  0  0  -  -  -  -  -
endef

$(eval $(generic-package))
```

Buildroot a une arborescence de dossiers spécifiques et utilise le mécanisme de configuration (Kconfig/Kbuild) du noyau Linux. Chaque package est défini dans un dossier et doit contenir un fichier "config.in", qui possède un langage spécial avec des variables et des commandes spécifiques comme le montre l'exemple 6.1. Le plus dur avec cet outil est d'apprendre la syntaxe proche du Makefile et de comprendre comment l'utiliser. Cette définition n'est toutefois pas très portable et difficile à maintenir.

En bref, buildroot est un outil de système de *build*, très utile dans le cas où il faut générer une simple distribution Linux embarquée, où peu de configurations sont nécessaires ; en effet, dans ce cas, la génération du système se fait avec peu de commandes. Mais l'inconvénient de cet outil est dans ses limitations au niveau de la configuration, en plus du fait qu'il ne peut pas être utilisé avec une interface graphique complète.

6.1.3.2 Yocto

Le projet Yocto est plus complexe que Buildroot [103]. Non seulement il peut construire des chaînes de compilation, des chargeurs d'amorçage, des noyaux et des systèmes de fichiers, mais il peut aussi générer une distribution Linux complète ainsi que des packages binaires qui peuvent être installés à l'exécution. Yocto est constitué de plusieurs fichiers recettes, écrits à l'aide d'une combinaison de Python et de script shell, ainsi que d'un ordonnanceur de tâches appelé BitBake qui génère tout ce qui a été configuré à partir de ces fichiers recettes.

Yocto est un outil de développement système qui aide à générer une distribution Linux personnalisée, il possède également plusieurs fonctionnalités pour le développement d'applications. Yocto construit et génère le kit de développement logiciel (SDK) qui contient le compilateur, le débogueur et les bibliothèques. Il génère aussi le SDK extensible (eSDK) contenant SDK et devtool (un outil permettant aux développeurs de maintenir le code source de leurs applications).

Les fichiers Yocto sont structurés par plusieurs couches (layers), en structurant les recettes et autres données de configuration de cette manière, il est très facile de l'étendre en ajoutant de nouvelles couches. De cette façon, l'utiliser pour un nouveau SoC est possible en ajoutant simplement une ou plusieurs couches à celles fournies par défaut qui contiennent un ensemble de projets open source.

La difficulté principale de Yocto est la syntaxe des recettes. Un fichier de recette contient plusieurs tâches à exécuter avec Bitbake et un grand nombre de variables qui contrôlent la façon dont un package est compilé et installé. Ensuite, Bitbake est un outil de ligne de commande, qui nécessite donc quelqu'un qui a l'habitude de travailler dans cet environnement, ce qui n'est pas toujours le cas pour les développeurs d'applications.

6.1.3.3 Fonctionnalités manquantes dans les systèmes de build

Yocto, qui fournit plus de fonctionnalités que Buildroot, peut sembler être une excellente solution pour créer des distributions Linux ; c'est un outil très puissant, régulièrement mis à jour et maintenu et il fournit toutes les fonctionnalités nécessaires pour générer un système Linux entièrement fonctionnel. Mais son principal inconvénient est sa difficulté d'utilisation qui nécessite d'apprendre un nouveau langage de script ayant une syntaxe complexe, ainsi que de s'habituer à manipuler ces nombreux paramètres de configuration.

En outre, toutes ses fonctionnalités s'effectuent à l'aide de lignes de commande, comme pour le lancer, contrôler la compilation complète de la distribution Linux, l'affichage du log et de l'état du *build*. Cela peut être un point bloquant pour de nombreux développeurs d'applications, car ils ne sont généralement pas très habitués à cette méthode.

Yocto génère un SDK qui contient les outils nécessaires pour développer des applications en espace utilisateur pour le système Linux créé, mais Yocto, lui-même n'est pas un outil de développement. Bien qu'il existe quelques modules d'extension (plugins) qui peuvent être ajoutés à certains IDE, ils ont des fonctionnalités très limitées. Yocto n'est pas intuitif et la plupart du temps, les développeurs doivent utiliser un éditeur de texte basique pour modifier une recette ou le développement du code et la ligne de commande pour compiler et déboguer les applications. Développer des applications pour Linux embarqué nécessite de modifier le code au niveau du noyau, ce qui n'est pas possible avec Yocto.

6.2 SW4Linux

SW4Linux a été créé par la société Ac6 pour résoudre la complexité des outils de systèmes de *build* classiques et pour fournir plus de fonctionnalités aux développeurs. Il s'agit d'un IDE convivial basé sur Eclipse qui fournit un système de *build* Linux avec interface graphique destinée aux systèmes embarqués. Il permet non seulement de construire facilement une distribution Linux embarquée, mais aussi de développer, déployer et déboguer une application dans le même environnement.

La figure 6.4 montre la différence entre deux définitions du même package, à gauche, le modèle SW4Linux, à droite, la recette Yocto. L'avantage, à part l'affichage de l'information avec une interface intuitive, est que lorsqu'il y a des champs à remplir, il n'est nécessaire ni de deviner quelle variable Yocto doit être définie ni son format.

Chaque élément de SW4Linux est à la base un projet Eclipse. Le plus important est appelé projet de plateforme, qui définit la carte cible, le noyau et la liste de tous les packages associés. Par défaut, des plateformes pour de nombreuses cartes de fournisseurs de SoC populaires sont fournies.

En partant d'une plateforme par défaut, il est possible d'importer des packages personnalisés depuis une archive en ligne, une archive ou un dossier local, git ou un projet Eclipse

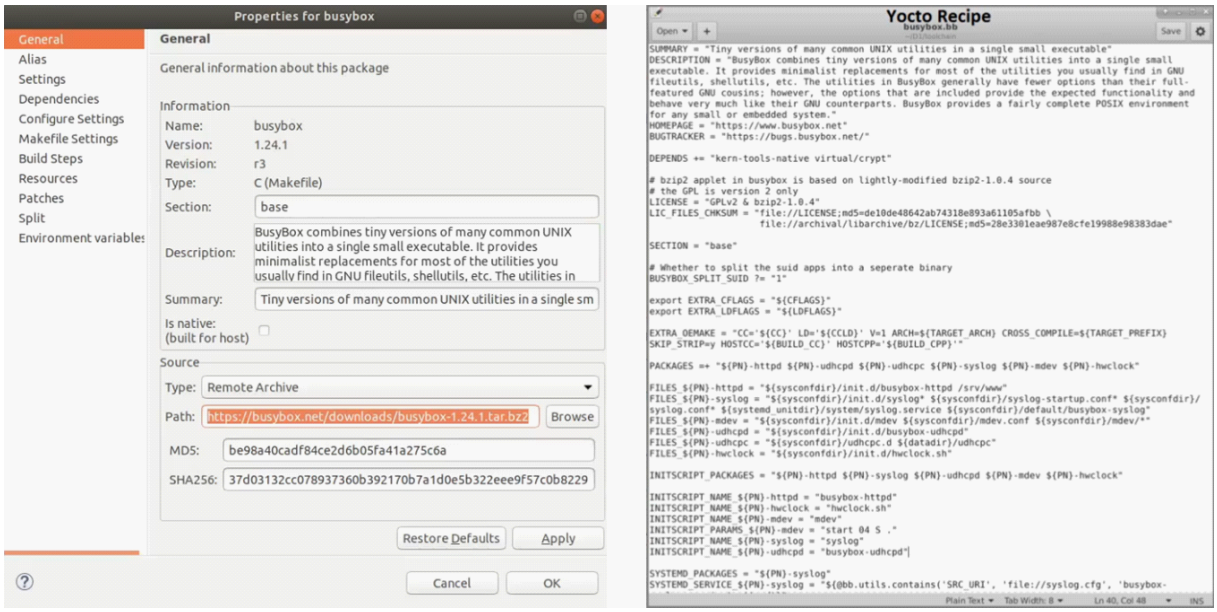

FIGURE 6.4 – Définition du package busybox : SW4Linux (gauche) vs Yocto (droite)

C/C++. Lors de la modification de la configuration d'un package, toutes les options disponibles sont affichées dans des boîtes de dialogue, aidant l'utilisateur à sélectionner une option précise de la manière la plus simple possible.

Un développeur de logiciels rencontre deux défis lorsqu'il travaille sur la construction d'un système Linux embarqué. Le premier est la génération du `device tree`, une étape particulièrement compliquée pour un débutant, car il est souvent utilisé et modifié au cours de la phase de développement ; SW4Linux détecte les options nécessaires qui permettent de le compiler en quelques clics. Le second est le déploiement et le débogage de l'application, SW4Linux fournit un environnement de débogage adapté à chaque plateforme.

SW4Linux n'est pas seulement destiné aux débutants en développement Linux, mais aussi aux experts qui travaillent sur le développement des modules et des pilotes. Cet outil supporte le développement de modules, des projets de modules se rattachent à un projet de noyau afin de les construire, ainsi que de les tester. SW4Linux offre une expérience de développement de système facile à utiliser, permettant d'éviter de perdre du temps à apprendre les nouveaux langages de script nécessaires aux systèmes de *build* traditionnels. En outre, il permet de gagner beaucoup de temps en évitant de devoir reconfigurer l'environnement pour construire les éléments liés au noyau. De plus, il offre des fonctionnalités introuvables dans d'autres IDEs comme la génération du code des pilotes en utilisant une interface graphique.

Plusieurs fonctionnalités ont été développées dans le cadre de cette thèse, notamment la fonctionnalité permettant la mesure du temps d'exécution d'une fonction écrite en C ou C++. Cette fonctionnalité duplique le projet sélectionné et injecte du code pour permettre la mesure, en fonction des méthodes de mesure personnalisées ou prédéfinies. L'outil configure le projet, effectue la mesure et extrait les données de sortie automatiquement à l'aide du débogueur. Il est également possible de fournir un ensemble de valeurs

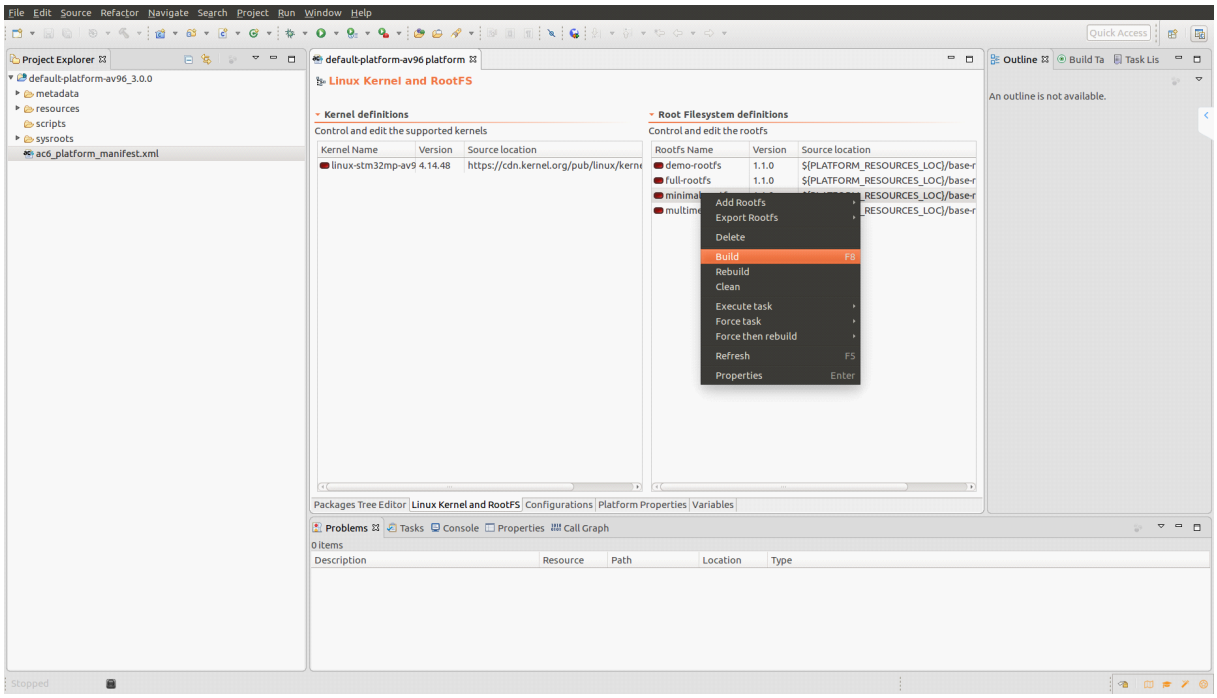

FIGURE 6.5 – Construire une image Linux embarquée en utilisant SW4Linux

d'entrée, afin d'effectuer la mesure de toutes ces données d'entrée, ce qui peut être répété en boucle plusieurs fois. Cette fonctionnalité est compatible avec les applications Linux ou microcontrôleur.

6.2.1 Aperçu

Comme SW4Linux est basé sur Eclipse, chaque élément est représenté sous un projet individuel. Le projet principal est la plateforme, il contient les listes de définitions des packages, des noyaux et des images de systèmes de fichiers compatibles avec une cible. De plus, il contient des informations sur le matériel, comme l'architecture et le compilateur à utiliser. La figure 6.5 montre comment construire un système de fichiers, ceci est fait en sélectionnant la définition de l'image voulue puis en faisant un clic droit suivi par *build*. Cette définition contient un noyau et la liste de packages qui doivent être compilés pour cette image.

Le *build* va déclencher la construction des différents packages, où pour chacun d'entre eux, un nouveau projet sera d'abord créé, puis il passera par plusieurs étapes. Par exemple, dans le cas le plus simple, la première étape est de télécharger le code source, la suivante est de le décompresser, puis d'appliquer les patchs (correctifs), de le configurer et enfin de le compiler. Lorsque la compilation de tous les packages est terminée, l'outil s'assure que toutes les bibliothèques partagées et les fichiers de configuration du système sont disponibles et correctement installés. Chaque étape est écrite en utilisant un script bash et définie dans un fichier séparé.

Une fois que le *build* est terminé, tous les packages qui ont été construits avec succès

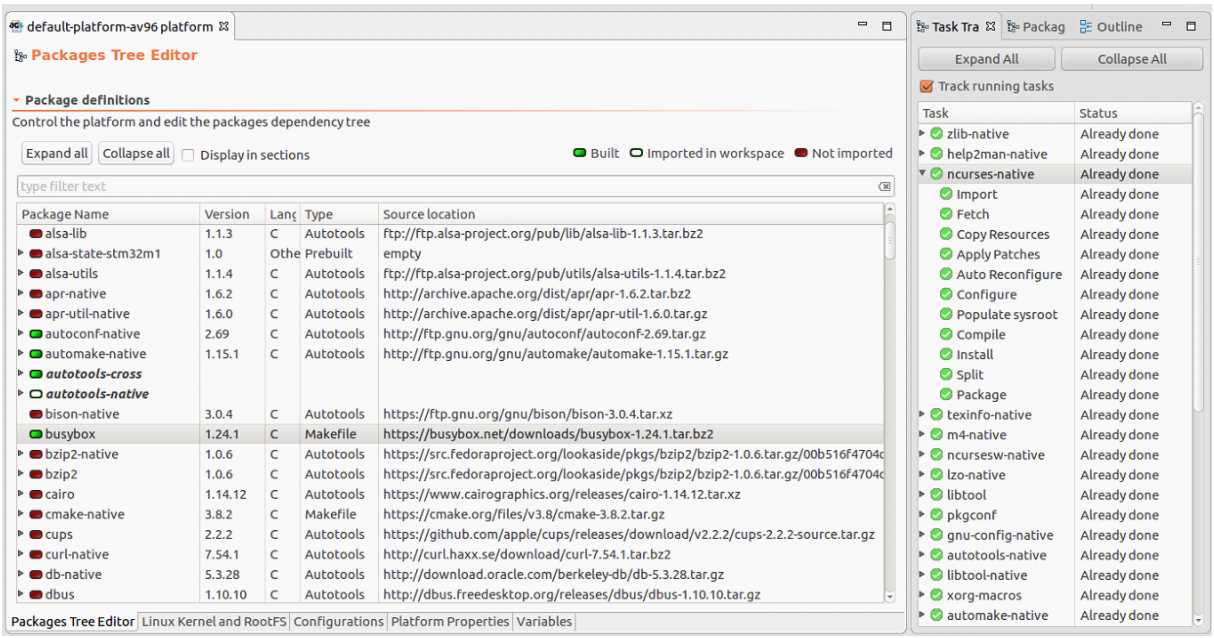

FIGURE 6.6 – SW4Linux après le build

passent en couleur verte comme le montre la figure 6.6. En sélectionnant n'importe lequel d'entre eux, cela affichera sur la droite toutes ces étapes avec leurs états (vert pour une exécution avec succès). Le résultat de chaque étape est stocké et peut être affiché en double-cliquant dessus.

6.2.2 Sécurité et démarrage sécurisé

Les systèmes embarqués étant très utilisés dans les applications critiques, ils sont confrontés à de nombreux défis en matière de sécurité. Sécuriser une image n'a jamais été une tâche facile dans la procédure de génération de la distribution Linux, car elle nécessite des connaissances spécifiques à la fois sur la partie matérielle de la puce embarquée et sur le logiciel utilisé pour sécuriser et signer l'image. Afin de répondre aux différents types d'attaques sur les systèmes électroniques et informatiques, SW4Linux intègre une fonctionnalité qui permet de créer des distributions Linux sécurisées embarquées.

Pour un démarrage sécurisé, chaque élément est signé par une clé qui permet d'effectuer une opération cryptographique. La signature est vérifiée à chaque étape de la séquence de démarrage. Cela crée une chaîne de confiance depuis la mise sous tension jusqu'au démarrage effectif du système d'exploitation, garantissant ainsi l'intégrité du système. Pour démarrer un système embarqué sécurisé, plusieurs logiciels sont nécessaires. Par exemple :

- `Arm Trusted Firmware (ATF)` est une implémentation de référence de logiciel sécurisé pour les processeurs ARMv7-A et ARMv8-A. Elle comprend un moniteur sécurisé fonctionnant au plus haut niveau de privilège réservé aux logiciels de bas niveau et au code sécurisé.
- `OP-TEE` est un environnement d'exécution sécurisé conçu pour accompagner un

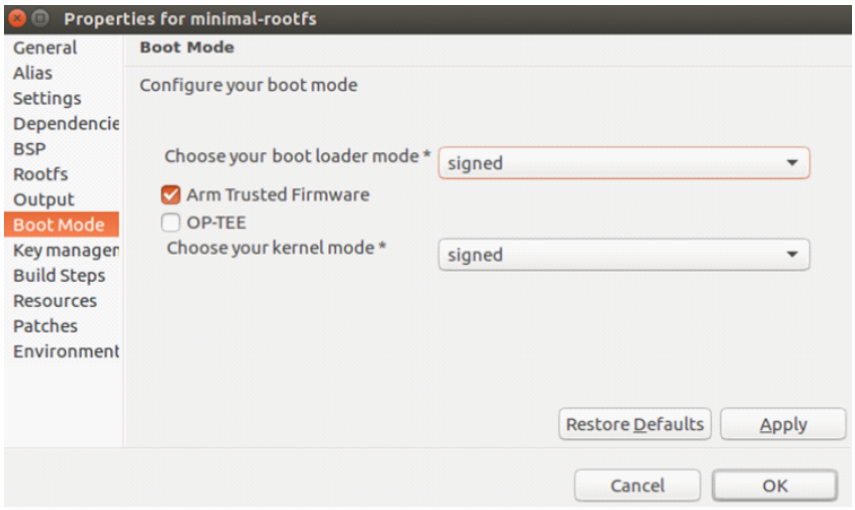

FIGURE 6.7 – Les options du mode de démarrage dans SW4Linux

noyau Linux non sécurisé fonctionnant sur ARM et utilisant un mécanisme d'isolation matérielle. L'une de ses fonctionnalités principales est le mécanisme qui permet de basculer entre le monde non sécurisé et le monde sécurisé.

Sur les HMPSoCs, la séquence de démarrage doit lancer les autres cœurs en mode sécurisé si nécessaire, dans ce cas, le processeur qui tourne en mode sécurisé doit charger le firmware puis lancer son exécution. Cette séquence de démarrage doit prendre en compte l'existence d'autres processeurs, et les périphériques partagés ne doivent pas être initialisés par plusieurs processeurs.

La simplification de la génération d'une distribution Linux embarqué sécurisée est l'objectif principal de SW4Linux. Il permet non seulement de créer, configurer et construire une image Linux embarquée à l'aide d'une interface utilisateur graphique, mais aussi de générer les clés de signature nécessaires et de signer cette image.

Nous avons développé, avec les alternants et les stagiaires de chez Ac6, une nouvelle interface pour la gestion des clés d'une manière uniforme sur toutes les plateformes. Cette fonctionnalité est basée sur un script de génération de clés basé sur plusieurs commandes OpenSSL. Elle peut être basée sur une clé existante ou sur la génération de nouvelle clé, l'utilisateur doit sélectionner le nom, l'algorithme et d'autres paramètres nécessaires. Ensuite, une structure de clé, contenant un ensemble de certificats, de clés privées et de clés publiques, est générée et utilisée automatiquement pour signer les images générées.

Pour choisir le mode de démarrage de la cible, une interface graphique est disponible qui permet à l'utilisateur de choisir la configuration, soit ATF, OP-TEE ou les deux, ainsi que la signature de démarrage (signée ou non signée) comme indiqué sur la figure 6.7.

6.2.3 Développement des pilotes Linux

Lorsqu'un processus en mode utilisateur a besoin d'accéder à des périphériques, il ne peut pas le faire directement, mais il doit passer à travers des pilotes de périphériques ou

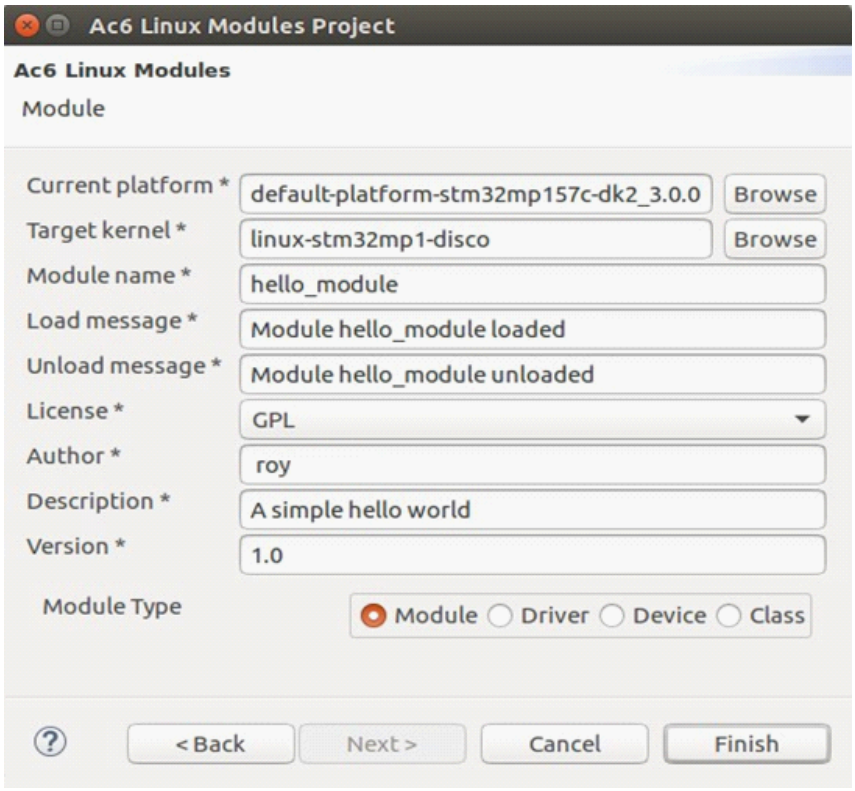

FIGURE 6.8 – Menu principal de création d'un nouveau pilote Linux

d'autres codes qui s'exécutent en mode noyau en utilisant des appels système. L'accès aux périphériques est au cœur des systèmes embarqués, c'est pourquoi la création de pilotes est essentielle et en même temps l'une des étapes les plus complexes pour la création d'applications embarquées.

Un modèle de périphérique unifié a été ajouté dans le noyau Linux afin de fournir un mécanisme uniforme pour représenter les périphériques et décrire leur structure dans le système. Le modèle de périphérique et de pilote Linux est un moyen global de les organiser dans des bus. Ceci est principalement fait afin de minimiser la duplication du code et de l'organiser. Le modèle de périphériques comprend plusieurs éléments :

- `Device` ou `Périphérique` : Un objet physique ou virtuel qui se rattache à un bus.
- `Driver` ou `Pilote` : Le logiciel qui va se connecter à un périphérique et le contrôler.
- `Bus` : Point de connexion à d'autres périphériques.
- `Class` : Catégorie de périphériques selon leur type.

Par défaut, la majorité des pilotes Linux ne sont pas intégrés à l'image du noyau. En effet, les pilotes peuvent être construits en tant que modules qui génèrent des fichiers objets du noyau qui doivent être placés dans le système de fichiers. Les modules sont chargés dynamiquement pendant l'exécution et ils font partie de l'espace noyau comme tout autre élément du noyau.

Nous avons développé une nouvelle fonctionnalité dans SW4Linux qui permet de générer le code d'un module Linux avec une interface graphique comme le montre la figure 6.8 en se basant sur des modèles génériques qui j'ai participé à développer et mettre au

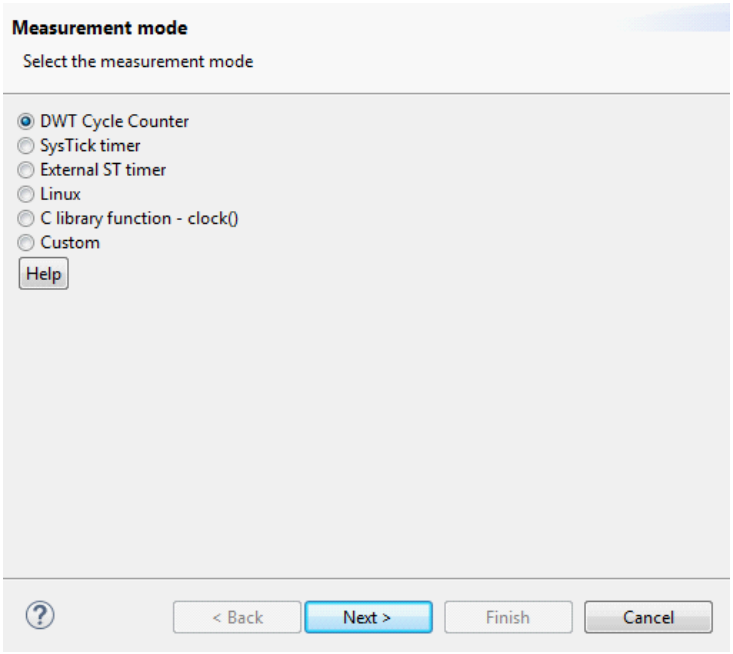

FIGURE 6.9 – Sélection de la méthode de mesure

point. Le module doit être rattaché à un projet noyau qui permet de le compiler facilement comme toute application normale. Le générateur de code de pilote crée un projet avec les paramètres de configuration nécessaires, il a plusieurs choix basés sur certaines conditions. Par exemple, il est possible de générer le code nécessaire pour les interruptions, les registres et les horloges du système. En outre, il supporte de nombreux types de bus comme PCI, USB, I2C etc...

Cet outil est un atout pour le développement des systèmes embarqués sous Linux car les pilotes sont le seul moyen d'accéder aux périphériques. Il est particulièrement adapté aux HMPSoCs comme dans le cas où il est nécessaire de créer un mécanisme de communication personnalisé entre les cœurs hétérogènes, ou sur les cartes qui intègrent un FPGA sur lequel il est possible de concevoir des périphériques personnalisés.

6.2.4 Plugin de mesure de temps d'exécution

Après les recherches menées sur les différentes méthodes de mesure, comme détaillé dans les chapitres 4 et 5, et la façon dont elles doivent être utilisées, ces méthodes ont été implémentées dans un nouveau plugin que j'ai développé pour SW4Linux qui permet de mesurer le temps d'exécution d'une fonction spécifique définie n'importe où dans un projet Eclipse C ou C++. Il s'agit d'une technique intrusive qui injecte du code de mesure en se basant sur des méthodes par défaut ou personnalisées.

Le plugin est formé de deux parties, la première est chargée de dupliquer le projet contenant la fonction à mesurer. Ensuite de modifier le code en injectant des boucles qui permettent de mesurer les différentes valeurs d'entrée fournies. La deuxième partie consiste à récupérer les données automatiquement en utilisant la session de débogage d'Eclipse,

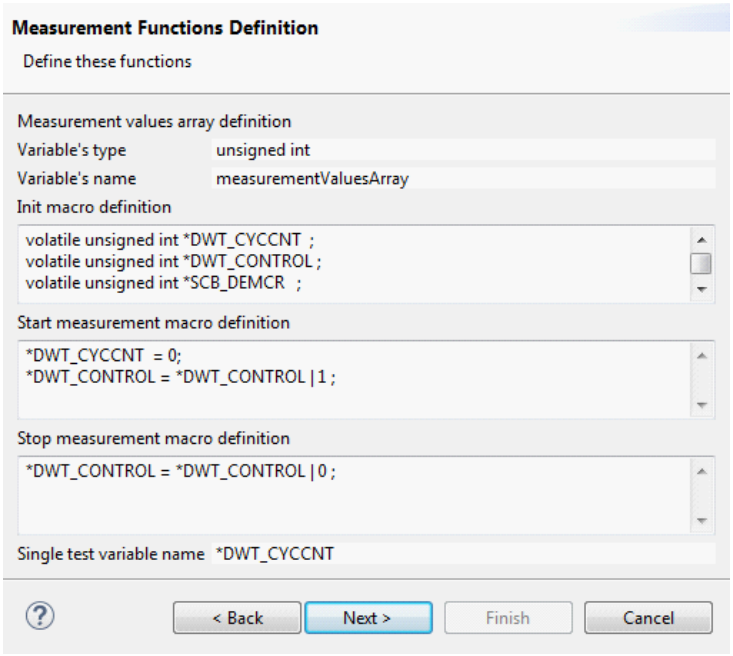

FIGURE 6.10 – Complétion du code de mesure

puis à générer un fichier (en format csv) contenant les mesures et les statistiques.

Il existe différentes méthodes pour mesurer le temps d'exécution, mais il n'y a pas de meilleure technique comme déjà discuté dans la section 4.5. En effet, chaque technique est un compromis entre plusieurs attributs, tels que la résolution, la précision, la granularité et la difficulté. Ce plugin propose plusieurs méthodes prédéfinies comme le montre la figure 6.9. En fonction de l'application et de la cible, la méthode la plus adaptée à la cible doit être utilisée ; il est également possible d'utiliser une méthode de mesure personnalisée.

L'étape suivante, affiche le code de mesure à utiliser ; pour les méthodes prédéfinies, il sera rempli automatiquement par le code correspondant comme par exemple la figure 6.10 qui montre le code basé sur le compteur de cycle des cœurs Cortex-M. Ensuite, il est possible de le compléter ou de le modifier, si nécessaire. Ce code utilise trois macros de style C pour les mesures. Tout d'abord, la macro d'initialisation, qui est appelée une fois avant de lancer les tests. Ensuite, les macros de début et de fin, qui sont appelées avant et après l'exécution de la fonction mesurée.

Après avoir saisi le nom de la fonction ainsi que son emplacement dans le projet, le plugin trouvera alors automatiquement cette fonction en se basant sur l'indexeur d'Eclipse et extraira toutes les données nécessaires la concernant, comme par exemple tous ses paramètres et leurs types.

La dernière étape est montrée sur la figure 6.11, chaque ligne représente un jeu de test (ou de mesure) qui nécessite un ensemble spécifique de valeurs d'entrée. Il faut configurer des jeux de test, représentant plusieurs valeurs d'entrée qui conduiront à un comportement particulier de la fonction à mesurer, si possible permettant de trouver une valeur proche de la borne supérieure, et qui sont chacun caractérisés par un nombre de mesures à effectuer. La création du projet qui prend les valeurs d'entrée est automatisée, ce qui simplifie

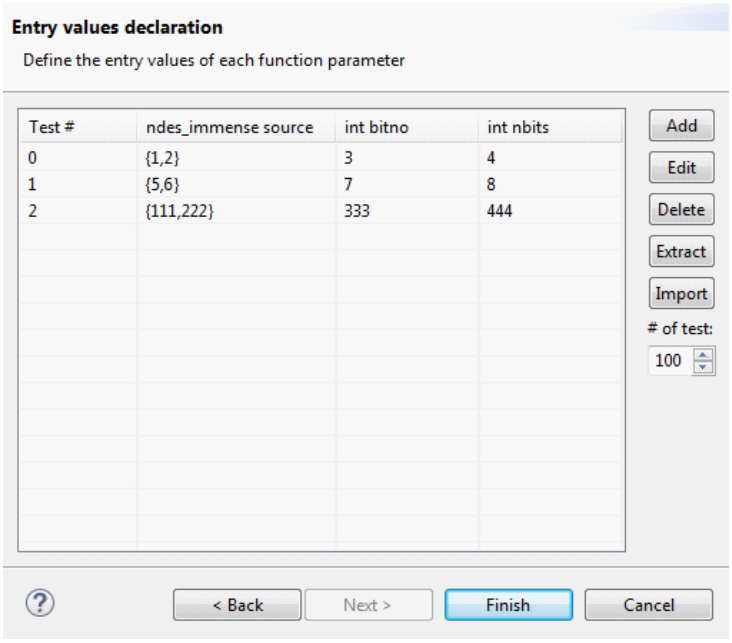

FIGURE 6.11 – Les valeurs d'entrée de la fonction à mesurer

	A	B	C	D	E	F	G
1	Measure	Occurence		Max	Min	Avg	Median
2	8764	1		10068	8764	9809.7	9924
3	9148	1					
4	9364	1					
5	9394	2					
6	9396	2					
7	9426	2					
8	9428	2					
9	9456	2					
10	9458	3					
11	9466	1					
12	9492	1					
13	9518	2					
14	9520	1					
15	9582	1					

FIGURE 6.12 – Un extrait du fichier de résultats de mesure

le travail du testeur. Les différents jeux de tests, qui sont généralement très nombreux, peuvent être écrits dans un fichier puis importés dans le plugin, au lieu de manuellement créer puis modifier le projet pour mettre les valeurs d'entrée au bon endroit une par une.

En utilisant ce projet créé, la fonctionnalité d'obtention de données doit d'abord être activée, puis il faut lancer la session de débogage. À la fin de l'exécution, l'outil récupère automatiquement les données et crée de nouveaux fichiers sous le dossier `Debug` dans le projet Eclipse avec les statistiques de mesure. Les données sont récupérées en utilisant le mécanisme de débogage propre à Eclipse (qui est généralement utilisé pour afficher les informations de débogage) et non directement par le débogueur, car le but de l'outil est de le rendre le plus générique possible, donc si un autre débogueur est utilisé, il sera compatible avec le plugin car toutes les données passent à travers le mécanisme interne d'Eclipse.

La figure 6.12 montre un exemple de fichier généré (en format csv), qui contient les

résultats d'un jeu de test. Il contient toutes les valeurs mesurées et leur occurrences, ainsi que la valeur maximale, minimale, moyenne et médiane. Dans cet exemple, l'unité des mesures est un cycle d'horloge parce que la méthode de mesure utilisée est basée sur le compteur de cycles.

6.3 Outils de développement des microcontrôleurs

Un IDE pour les microcontrôleurs est un outil de développement de code avec une interface graphique composée de plusieurs outils, qui permet, en utilisant un langage de programmation, de créer un exécutable compatible avec une cible, ainsi que de la déboguer. L'IDE se compose d'un éditeur de code permettant de naviguer rapidement entre les nombreux fichiers qui constituent un projet, une chaîne de compilation croisée (qui comprend le compilateur, l'éditeur de lien etc...) et des outils de débogage (comme le débogueur GDB et le serveur GDB). Les outils pour microcontrôleurs doivent fournir le code minimal spécifique à la cible nécessaire pour créer une application compatible, ce qui inclut le code de démarrage (qui contient la table de vecteur) et le script d'édition de liens.

L'IDE doit fournir une fonctionnalité de débogage avec une interface graphique, ainsi que tous les pilotes nécessaires pour accéder aux sondes de débogage (comme J-LINK ou ST-LINK). En outre il doit fournir des paramètres spécifiques à la cible pour rendre possible le débogage comme décrit dans la section 3.12. Les fonctionnalités apportées sont : l'exécution pas à pas, l'ajout de points d'arrêt, la lecture des adresses mémoire, la lecture des valeurs des registres etc.

Un firmware spécifique à une cible ou à une famille de cibles doit également être fourni. Cela inclut tous les fichiers d'en-tête qui contiennent la définition des fonctions qui peuvent contrôler et configurer le matériel. Certains IDEs offrent des fonctionnalités plus avancées, ils fournissent non seulement le firmware, mais aussi la possibilité de configurer le matériel (y compris les horloges système, l'alimentation, les interruptions et les options spécifiques aux périphériques) en utilisant une interface graphique qui génère le code correspondant.

Les IDEs existants pour les systèmes embarqués sont soit des outils commerciaux payants, soit des outils gratuits. Habituellement, les outils payants proposent plus de fonctionnalités et possèdent un compilateur propriétaire qui offre certains avantages par rapport aux compilateurs open source. Presque tous les fournisseurs de SoC proposent un IDE gratuit qui a toutes les fonctionnalités nécessaires pour créer une application complète contenant le firmware requis. Ces outils sont généralement basés sur Eclipse et peuvent être utilisés pour développer le côté microcontrôleur sur un HMPSoC. Parmi ces outils :

- `SW4STM32` ou System Workbench for STM32, est un outil développé par Ac6 [2], il peut être utilisé pour développer sur des microcontrôleurs basés sur STM32, il fournit tous les fichiers nécessaires y compris le firmware avec la possibilité d'ajouter FreeRTOS et quelques autres middlewares comme LWIP. L'outil peut aussi faire la

compilation et le débogage croisés.

- `STM32CubeIDE` Développé par STMicroelectronics [109]. Il offre les mêmes fonctionnalités que le SW4STM32 avec quelques caractéristiques supplémentaires, la plus importante étant la possibilité de configurer le matériel avec une interface graphique.
- `MCUXpresso` Cet outil est développé par NXP [84], il a les mêmes caractéristiques principales que STM32CubeIDE mais pour les microcontrôleurs basés sur NXP.

6.4 Environnement de développement unifié

L'IDE unifié est un outil qui devrait être capable de couvrir tous les aspects du développement sur les HMPSoCs. Il a au moins deux spécificités, le côté des cœurs à haute performance qui nécessite Linux et l'autre côté avec un microcontrôleur utilisé en bare-metal ou avec un RTOS. Typiquement, ce type de système nécessite la connaissance de plusieurs types de développement de systèmes embarqués, chacun avec un outil différent : microcontrôleur (chaque fournisseur a son propre IDE), distribution Linux (en utilisant Yocto ou Buildroot), espace utilisateur Linux (en utilisant n'importe quel IDE compatible mais cela nécessite une configuration manuelle) et espace noyau Linux (cela est fait manuellement).

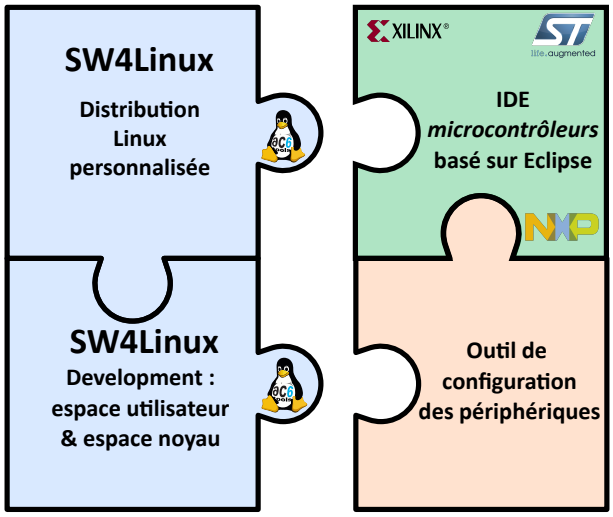

FIGURE 6.13 – Fonctionnalités de l'IDE unifié

SW4Linux peut être installé sur n'importe quel IDE basé sur Eclipse en l'ajoutant comme un plugin supplémentaire. Cela permet de développer une plateforme basée sur Linux avec le même IDE, ce qui donne un environnement unifié pour le développement sur puce à cœurs asymétriques. J'ai participé à la création de la liste des éléments qui doivent être développés pour supporter le développement des applications AMP. Ensuite, après le développement de ces éléments par mes collègues et moi-même, je les ai testés de manière intensive afin de m'assurer qu'ils fonctionnent correctement et d'indiquer les points d'amélioration. La figure 6.13 montre les éléments de l'IDE unifié. Du côté microcontrôleur, le firmware et la configuration des périphériques sont réalisés à l'aide de

l'outil spécifique au microcontrôleur et SW4Linux le complète avec les fonctionnalités nécessaires pour développer tout ce qui concerne le coté Linux embarqué : la génération d'une distribution Linux personnalisée et sécurisée, le développement au niveau de l'espace utilisateur et noyau, la compilation du `device tree` et enfin la mesure du temps d'exécution des fonctions lorsque cela est nécessaire, qui fonctionne à la fois sous Linux et sur les microcontrôleurs.

6.4.1 Développement d'applications AMP

Le développement d'une application AMP nécessite la création de deux projets pour les deux cotés : Linux et microcontrôleur.

Tout d'abord, l'application du côté Linux est créée à l'aide du projet de compilation croisée Eclipse en utilisant les options par défaut ; sans avoir besoin de remplir les informations sur le compilateur ou le chemin du compilateur. Ensuite, le projet doit être rattaché à la plateforme SW4Linux de la cible. Ce projet sera alors automatiquement configuré pour le rendre compatible avec la distribution générée. Le compilateur et les bibliothèques sont détectés, et toutes les informations nécessaires sont ajoutées aux paramètres du projet. La compilation de l'application croisée Linux se fait comme pour tout autre projet Eclipse, en appuyant sur l'icône du marteau ou en faisant un clic droit sur le projet puis *build* du projet.

La communication entre les cœurs hétérogènes est nécessaire ; cela se fait grâce au protocole de communication asymétrique OpenAMP (ou un autre alternatif). Sur le microcontrôleur, la bibliothèque de ce protocole fait généralement partie du firmware ; il suffit d'inclure la bibliothèque nécessaire dans le projet. Du côté de Linux, avant de commencer le développement, il est important de s'assurer que le pilote du protocole est activé dans le noyau. SW4Linux fournit une fonction permettant d'activer et de désactiver les options du noyau à l'aide de cases à cocher, en fonction des fragments de configuration du noyau ; pour les cartes AMP, il faut utiliser le pilote de transmission de messages rpmsg. La figure 6.14 suivante montre le pilote rpmsg activé.

Le pilote de communication AMP, comme tout autre pilote, est accessible par un fichier particulier dans /dev (/dev/ttyRPMSG) ; ce pilote permet aux programmes de l'espace utilisateur d'envoyer et de recevoir des messages AMP, en mode lecture ou écriture, à partir d'un périphérique série virtuel en utilisant les appels système standard. Du côté du microcontrôleur, les fonctions définies par la bibliothèque OpenAMP permettent d'envoyer et de recevoir les messages avec le côté Linux. Cette bibliothèque est disponible soit pour une configuration baremetal, soit pour un système d'exploitation temps réel comme FreeRTOS.

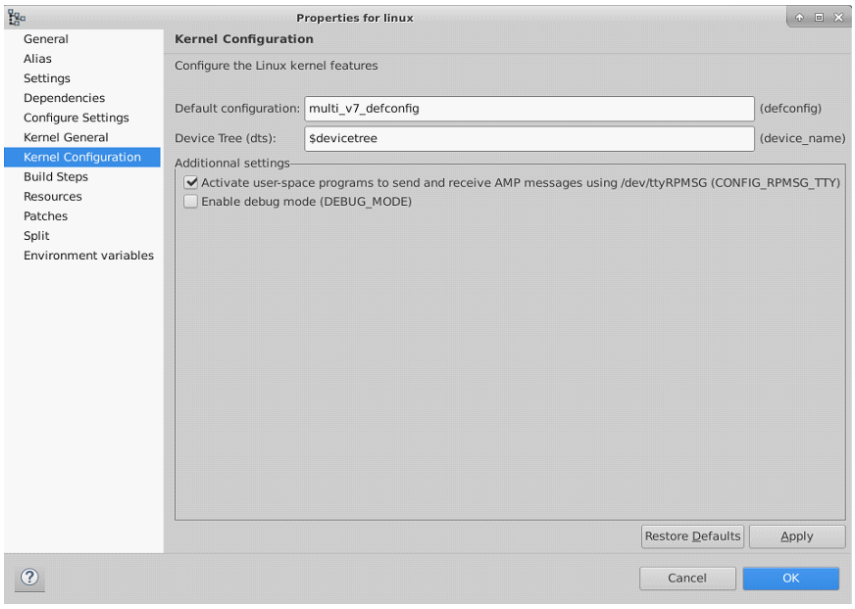

FIGURE 6.14 – Menu de configuration du noyau

6.4.2 Débogage des applications AMP

Le projet du programme utilisé pour développer l'application côté Linux peut être déployé, testé ou débogué en utilisant SW4Linux. La fonctionnalité "AC6 C/C++ Remote Debugging" est utilisée à cette fin ; elle détecte automatiquement la plateforme de la cible et fournit le débogueur compatible avec les paramètres de débogage correspondants. La seule information qui doit être remplie manuellement est l'adresse IP de la cible, celle-ci est nécessaire pour trouver la cible, afin de transférer, installer et exécuter le programme à distance, en plus du démarrage du serveur de débogage.

Pour le côté microcontrôleur, il est basé sur l'outil de développement spécifique au microcontrôleur, chaque fournisseur ayant des paramètres de débogage différents, tout ce qu'il faut faire est de lancer la session de débogage. Grâce à ces étapes simples, un système entièrement opérationnel sera prêt à être utilisé pour le développement et le débogage des applications AMP.

La figure 6.15 montre une capture d'écran de cet IDE unifié (formé de SW4STM32 et SW4Linux) : sur le côté gauche, la session de débogage du microcontrôleur s'exécute avec l'application bloquée sur un point d'arrêt ; sur le côté droit, la session de débogage de Linux à distance développé par Ac6 avec le code également bloqué sur un point d'arrêt. Dans cette application, le protocole OpenAMP qui fournit le mécanisme de communication entre les cœurs AMP a été utilisé et un point d'arrêt a été fixé lors de l'envoi ou de la réception de messages.

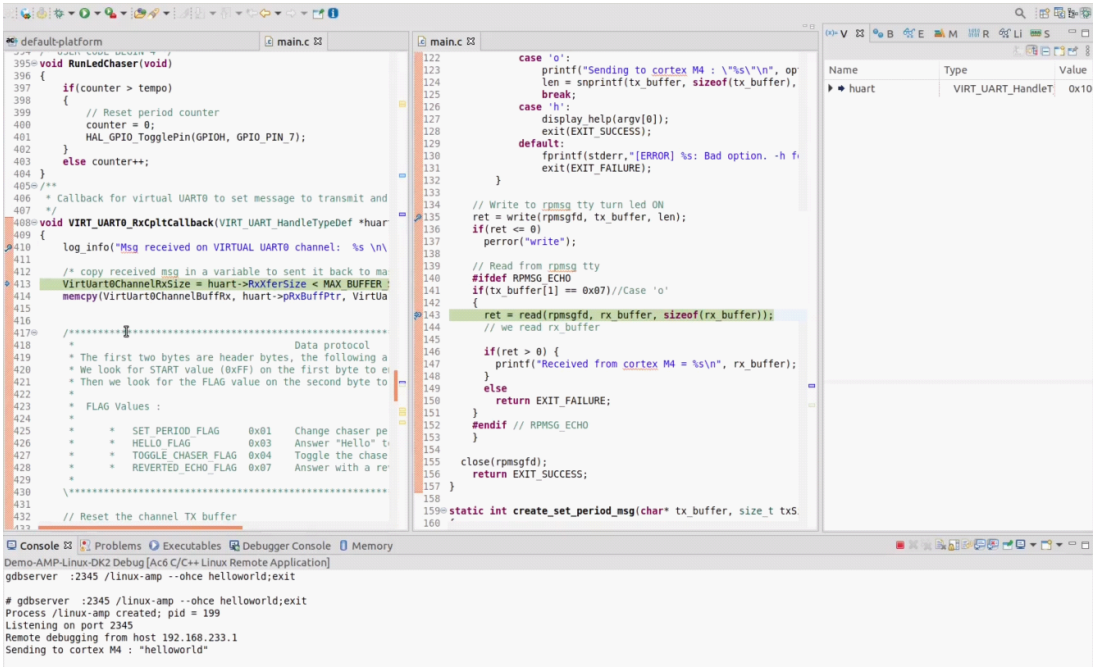

FIGURE 6.15 – Deux sessions de débogage simultanées sur des cœurs hétérogènes

6.5 Conclusion

Une nouvelle fonctionnalité a été développée qui permet de mesurer le temps d'exécution d'une fonction. Elle injecte le code de mesure en se basant sur une méthode personnalisée ou une des méthodes prédéfinies. Chaque mesure nécessite le jeu de valeurs d'entrée de la fonction, sachant que plusieurs peuvent être fournies. Ensuite, une session de débogage doit être lancée afin d'extraire automatiquement les valeurs de ces mesures.

Le développement d'un système Linux, en particulier sur le HMPSoC, présente un niveau élevé de complexité, comme le montre ce chapitre ; il nécessite également de nombreux outils. Ceci est applicable à tous les niveaux de développement, système, application et noyau. Ce chapitre décrit la configuration du démarrage du système, y compris ses différentes étapes complexes ainsi que les éléments nécessaires à la création d'une distribution Linux personnalisée et au développement et au débogage d'une application compatible avec elle.

SW4Linux unifie le développement de l'ensemble du système basé sur Linux dans un seul environnement. Cet outil peut générer une distribution Linux personnalisée, sécurisée et signée, en utilisant une interface graphique. Il peut également être utilisé pour développer et déboguer des applications au niveau de l'utilisateur et du noyau. En le fusionnant avec un IDE pour microcontrôleurs, on obtient un seul outil permettant de développer tous les éléments du HMPSoC. Le côté Linux est géré par SW4Linux et le côté microcontrôleur par son IDE spécifique. Il est également possible de déboguer des cœurs hétérogènes, en utilisant deux sessions de débogage lancées simultanément chacune sur un côté.

Chapitre 7

Conclusion et perspectives

Sommaire

7.1 Aperçu de la thèse 165
7.2 Mesure de durée d'exécution 165
7.2.1 Étude sur les méthodes de mesure sur un cœur 165
7.2.2 Contribution industrielle 167
7.3 Application AMP 167
7.3.1 Mesure de temps d'exécution d'un message AMP 167
7.3.2 Migration d'une tâche entre processeurs hétérogènes 168
7.3.3 Perspectives 168
7.4 Environnement de développement unifié 169
7.5 Fiabilisation des mesures et analyses de durées 169

7.1 Aperçu de la thèse

Dans cette dernière partie, nous présentons une synthèse générale de la recherche puis les contributions qui en découlent. Nous discuterons enfin des perspectives de recherche dans le but d'enrichir les résultats obtenus.

Les processeurs multicœurs hétérogènes, formés d'un processeur à haute performance et d'un microcontrôleur à basse consommation pour les applications temps réel représentent le cœur de notre travail de recherche. Ces derniers rendent possible la création de nouvelles applications qui n'étaient pas envisageables auparavant où l'amélioration de l'efficacité d'une même application. Or, plusieurs défis existent à plusieurs niveaux et en particulier pour les développeurs étant donné qu'il n'existe pas d'outil capable de gérer le développement de l'ensemble du système. Un autre défi concerne les applications temps réel, car ces plateformes sont très complexes ce qui rend plus difficile l'estimation statique et dynamique du temps d'exécution du code.

Cela nous a amenés à étudier plusieurs techniques de mesure qui permettent d'obtenir un ordre de grandeur du temps d'exécution d'un bloc de code en utilisant des solutions logicielles ou des composants matériels dans le SoC. Puis à intégrer ces méthodes dans une nouvelle fonctionnalité développée pour l'IDE SW4Linux. Nous les avons comparées en considérant des critères tels que la disponibilité de la méthode dans le système, la simplicité de mise en place et la précision requise selon les contraintes temporelles de l'application. En outre, nous avons discuté de la manière dont les valeurs mesurées peuvent être extraites de la cible à l'aide du débogueur de façon à réduire l'impact des campagnes de mesure sur la mémoire et le cache.

Nous avons mis en œuvre une solution pour la migration des tâches entre processeurs hétérogènes. Cette méthode exploite des technologies comme le contrôleur de communication et de synchronisation inter-cœurs ainsi que la mémoire partagée. Nous avons également fourni plusieurs méthodes de mesure, basées sur un timer externe sur puce, qui permettent de mesurer le temps de communication AMP. Chacune a des conditions et des limitations différentes, nous les avons étudiées, nous avons présenté les avantages et les inconvénients de chacune d'entre elles. Cela nous a permis de déterminer que la méthode `zero-gauche-droite-stop` était la plus simple à mettre en oeuvre et la plus flexible, permettant d'extraire les données soit côté émetteur, soit côté récepteur, ceci en permettant d'effectuer des séries de mesures.

7.2 Mesure de durée d'exécution

7.2.1 Étude sur les méthodes de mesure sur un cœur

Nous avons souligné l'importance de l'analyse de temps d'exécution en comparant les différents types d'analyse notamment l'analyse statique qui représente la méthode la plus sûre qui devrait être appliquée dans le cas d'une application temps réel dur comme dans le

domaine médical ou avionique de criticité haute. Mais nous avons montré les contraintes qui y sont liées et la difficulté que la méthode statique représente à cause de la nature très complexe des processeurs récents et le coût élevé pour l'appliquer, surtout pour les HMPSoCs. En effet, le problème pour supporter ce type d'analyse demanderait une quantité de travail très importante face à une technologie qui évolue très rapidement et propose régulièrement des nouvelles puces. En outre, si cela devait être fait, les interférences possibles aux différents niveaux (bus mémoire, mémoire externe, mémoire cache, composants heuristiques, etc.) conduiraient à une très forte surestimation du WCET due aux nombreux comportements possibles du programme et de la plateforme d'exécution. Par ailleurs, les architectures modernes ont tendance à optimiser le temps d'exécution moyen au détriment du déterminisme. L'autre façon de calculer le temps d'exécution moyen ou le pire cas sur une plateforme est appelée *dynamique*, et consiste à mesurer le temps d'exécution réel du programme analysé sur la plateforme cible. Cette technique peut être appliquée sur une nouvelle plateforme dont les HMPSoCs. Cette technique n'est pas sûre, elle devrait être utilisée pour les systèmes ayant des contraintes temporelles moins strictes, appelés systèmes temps réel *mous*. Il faut noter que si l'environnement nécessite des contraintes temporelles plus strictes, une analyse hybride, qui représente un mixe entre l'analyse statique et dynamique ou une analyse probabiliste basée sur des estimations statistiques, peuvent être utilisées.

La complexité de différentes plateformes récentes rend l'approche basée sur la mesure plus adaptée. Une comparaison entre différentes méthodes de mesure de temps d'exécution, basée sur plusieurs attributs (la précision, la difficulté, la granularité et la résolution) a été effectuée sur plusieurs cibles de différentes caractéristiques. Il a été constaté expérimentalement que la plupart des méthodes de mesure basées sur des compteurs donnent des résultats similaires, ce qui implique que le choix des méthodes doit reposer sur la simplicité et l'accessibilité. Toutes ces méthodes sont basées sur le même principe de chronomètres qui est une méthode de base pour mesurer le temps écoulé entre le début et la fin d'un bloc de code. Elles nécessitent un périphérique de mesure comme le compteur de cycles d'horloge, qui est en général intégré dans la plupart des processeurs récents, ou un timer (ou compteur) sur puce qui est un périphérique dans le SoC à l'extérieur du processeur et offre une plus grande flexibilité et de nombreuses fonctionnalités qui permettent de le configurer de nombreuses façons. Par exemple, la résolution du timer peut être modifiée en augmentant ou en diminuant sa fréquence. Les méthodes chronométriques peuvent également être basées sur des solutions logicielles telles que les fonctions du standard C comme `time()` et `clock()`.

Enfin, nous avons pu observer expérimentalement en mesurant divers benchmarks que les cœurs hétérogènes sur le même HMPSoC ne partagent pas les accélérateurs, ce qui indique la nécessité de les placer dans une nouvelle catégorie de multicœurs : `architecture consistante`, dans laquelle le processeur rapide peut généralement exécuter toutes les tâches plus rapidement que le processeur le plus lent.

7.2.2 Contribution industrielle

Nous avons développé un nouveau module pour l'outil SW4Linux qui permet aux utilisateurs de réaliser des mesures de temps d'exécution des fonctions C/C++, il injecte le code de mesure en se basant sur une méthode personnalisée ou une des méthodes prédéfinies. Chaque mesure nécessite le jeu de valeurs d'entrée de la fonction, sachant que plusieurs peuvent être fournies. Pour cet outil, nous avons essayé de rendre la technique facile à utiliser, ne nécessitant que le matériel cible et l'ordinateur de développement, donnant la plus haute résolution possible en fonction des horloges ou du système d'exploitation disponibles. Le module développé récupère les données de mesure automatiquement à travers le débogueur et les exporte vers une fiche de données. La limitation, comme pour les autres méthodes dynamiques, est l'impossibilité de connaître la précision.

7.3 Application AMP

Nous avons vu la structure d'une application AMP qui utilise tous les processeurs disponibles dans les HMPSoCs, ainsi que le mécanisme de communication AMP et les ressources partagées. Un HMPSoC est généralement constitué de deux processeurs hétérogènes, le premier, un microprocesseur, sous Linux et le second, un microcontrôleur, sans système d'exploitation (ou avec un système d'exploitation temps réel sans MMU). Les deux tournent en parallèle, en exécutant des codes distincts qui sont compilés à l'aide de compilateurs différents, communiquent entre eux en utilisant au niveau matériel les périphériques de communication, de synchronisation et la mémoire partagée, et au niveau logiciel un protocole d'envoi et de réception de messages comme OpenAMP.

7.3.1 Mesure de temps d'exécution d'un message AMP

Ce type de systèmes comporte de nombreux défis pour l'analyse du temps d'exécution, en raison de l'architecture complexe du matériel. Les méthodes basées sur la mesure sont plus simples à mettre en œuvre que les méthodes d'analyse statique. Les méthodes dépendantes du cœur ne peuvent pas être utilisées à cette fin car elles ne sont pas accessibles par d'autres parties du système.

Plusieurs méthodes de mesure ont été proposées et comparées, elles sont des méthodes chronométriques basées sur le timer à usage général accessible par les différents processeurs et qui est souvent disponible sur tous les SoCs récents. Le choix dépend de la simplicité, la disponibilité des périphériques de synchronisation comme le sémaphore matériel, ainsi que des conditions comme la possibilité d'établir la connexion entre le processeur et le débogueur, ou de reconfigurer et réinitialiser le timer.

7.3.2 Migration d'une tâche entre processeurs hétérogènes

Une méthode de migration d'une tâche entre processeurs hétérogènes a été présentée, qui permet à une tâche de migrer entre des cœurs ayant des ISA et des fréquences différentes. Cette méthode est basée sur des points de migration dans l'application, où les données locales sont transférées en utilisant le protocole de communication AMP. Le temps de latence de communication a été mesuré à l'aide de la méthode de mesure proposée, et il a été constaté qu'il est d'un ordre de magnitude supérieur au temps de migration entre deux cœurs du même cluster et varie beaucoup d'un cœur à l'autre. Cela nous a permis de montrer expérimentalement la nécessité de différencier les deux types de migration (intra-cluster, vs. inter-cluster) sur le modèle de plateforme HMPSoC, ainsi que d'adopter un modèle hiérarchique dans lequel les cœurs de même type seront groupés dans des clusters.

Enfin, afin d'observer expérimentalement si une application pouvait tirer parti d'un ordonnancement global sur plateforme HMPSoC, une étude de cas basée sur ROSACE a été réalisée, dans laquelle nous avons comparé le temps d'exécution en utilisant uniquement le cluster SMP à celui utilisant le cœur hétérogène (plus lent). Il a été observé que le coût de la migration entraînera un temps d'exécution global plus long, ce qui signifie un système moins performant. Ensuite, nous l'avons comparé avec un scénario où le cluster de processeurs rapides étaient stressé et très chargé, et dans ce cas, un léger avantage a été observé.

En conclusion, bien que, en théorie, l'ordonnancement temps réel global permette une totale utilisation de la plateforme, en pratique, celui-ci est complexe à mettre en œuvre. Dans le cas où il n'y a pas besoin d'un processeur dédié pour une application spécifique, ce processeur peut être utilisé grâce à la migration hétérogène, mais il doit respecter certaines conditions. Car la méthode de migration qu'on a suggérée dans cette thèse a montré qu'elle n'est pas toujours avantageuse, la raison principale étant le coût élevé de la migration.

7.3.3 Perspectives

Il serait intéressant de créer un ordonnanceur qui permette la migration hétérogène, ceci simplifierait au moins le travail du développeur. Il faudrait mettre en place au préalable les points de migration logiciels dans le code, par conséquence c'est l'ordonnanceur qui déciderait d'effectuer la migration de cette tâche selon les points instrumentés en utilisant le protocole de communication sur lequel la migration hétérogène est basée. L'ordonnanceur principal serait implémenté sur un cluster maître qui prendrait les décisions de migration des tâches vers d'autres clusters ou cœurs esclaves. Ces derniers devraient inclure un logiciel ou une bibliothèque permettant la migration et capable de communiquer avec l'ordonnanceur.

La migration hétérogène des tâches ne devrait donc être autorisée que si le microcontrôleur n'est pas occupé et lorsque le processeur principal est très chargé. En fonction de

ces conditions, l'ordonnanceur peut utiliser le système de manière plus efficace.

7.4 Environnement de développement unifié

Le développement d'un système Linux, en particulier sur le HMPSoC, présente un niveau élevé de complexité notamment pour les trois types de développement : système, application et noyau. le processus de démarrage d'un système Linux comprend des différentes étapes complexes. Sur les HMPSoCs, cette séquence de démarrage doit prendre en compte l'existence d'autres processeurs, et les périphériques partagés ne doivent pas être initialisés par plusieurs processeurs.

SW4Linux unifie le développement de l'ensemble du système basé sur Linux dans un seul environnement. Cet outil peut générer une distribution Linux personnalisée, sécurisée et signée, en utilisant une interface graphique. Il peut également être utilisé pour développer et déboguer des applications au niveau de l'utilisateur et du noyau. En le fusionnant avec un IDE pour microcontrôleurs, on obtient un seul outil permettant de développer tous les éléments du HMPSoC. Le côté Linux est géré par SW4Linux et le côté microcontrôleur par son IDE spécifique. Il est également possible de déboguer des cœurs hétérogènes, en utilisant deux sessions de débogage lancées simultanément chacune sur un côté.

7.5 Fiabilisation des mesures et analyses de durées

Les méthodes de mesure présentées dans cette thèse ont pris le parti d'être utilisables sans matériel spécifique additionnel, que nous estimons pas nécessairement présent dès la sortie d'une nouvelle carte, et généralement coûteuse. De plus, nous avons voulu nos méthodes portables, d'où la non utilisation de registres dédiés, sur certains types de cœurs, à l'étude des performances. Il serait intéressant de s'intéresser à ces registres spécifiques, quitte à perdre en portabilité en ajoutant des méthodes de mesure spécifiques lorsqu'elles sont disponibles sur la cible.

Enfin, ces méthodes ne permettent que d'extraire des histogrammes de mesures statiques, de fonctions paramétrables. Il serait intéressant de les étendre à des analyses hybrides permettant de mieux caractériser leur représentativité, ou bien de les utiliser dans un cadre d'analyse de WCET probabiliste.

Bibliographie

[1] M. O. Aboelhassan, O. Bartik, and M. Novak. Embedded multi-core systems for mixed-critical applications with rpmsg protocol based on xilinx zynq-7000. In *2017 7th IEEE International Conference on Control System, Computing and Engineering (ICCSCE)*, pages 162–167. IEEE, 2017. (Cité en page 82)

[2] Ac6. System workbench for stm32. https://www.openstm32.org/, 2018. (Cité en page 157)

[3] S. Akhter and J. Roberts. *Multi-core programming*, volume 33. Intel press Hillsboro, 2006. (Cité en page 33)

[4] S. Altmeyer and C. Burguière. Cache-related preemption delay via useful cache blocks : Survey and redefinition. *Journal of Systems Architecture*, 57(7) :707–719, 2011. (Cité en page 94)

[5] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio. Performance comparison of vxworks, linux, rtai, and xenomai in a hard real-time application. *IEEE Transactions on Nuclear Science*, 55(1) :435–439, 2008. (Cité en page 45)

[6] S. Baruah, M. Bertogna, and G. Buttazzo. *Multiprocessor scheduling for real-time systems*. Springer, 2015. (Cité en page 129)

[7] K. Baukus and R. Van Der Meyden. A knowledge based analysis of cache coherence. In *International Conference on Formal Engineering Methods*, pages 99–114. Springer, 2004. (Cité en page 57)

[8] F. Baum and A. Raghuraman. Making full use of emerging ARM-based heterogeneous multicore SoCs. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016. (Cité en page 63), (Cité en page 80)

[9] F. Baum and A. Raghuraman. Making full use of emerging arm-based heterogeneous multicore socs. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016. (Cité en page 80)

[10] T. Benavides, J. Treon, J. Hulbert, and W. Chang. The enabling of an execute-in-place architecture to reduce the embedded system memory footprint and boot time. *J. Comput.*, 3(1) :79–89, 2008. (Cité en page 144)

[11] A. Bertout, J. Goossens, E. Grolleau, and X. Poczekajlo. Template schedule construction for global real-time scheduling on unrelated multiprocessor platforms.

In *2020 Design, Automation & Test in Europe Conference & Exhibition*, pages 216–221. IEEE, 2020. (Cité en page 124)

[12] A. Bertout, J. Goossens, E. Grolleau, and X. Poczekajlo. Workload assignment for global real-time scheduling on unrelated multicore platforms. In *Proceedings of the 28th International Conference on Real-Time Networks and Systems*, pages 139–148, 2020. (Cité en page 39), (Cité en page 124), (Cité en page 128)

[13] A. Bonenfant, I. Broster, C. Ballabriga, G. Bernat, H. Cassé, M. Houston, N. Merriam, M. de Michiel, C. Rochange, and P. Sainrat. Coding guidelines for wcet analysis using measurement-based and static analysis techniques. *IRIT-Institut de Recherche en Informatique de Toulouse, Tech. Rep, Tech. Rep. IRIT/RR-2010-8-FR*, 2010. (Cité en page 89)

[14] V. Boppana, S. Ahmad, I. Ganusov, V. Kathail, V. Rajagopalan, and R. Wittig. Ultrascale+ mpsoc and fpga families. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–37. IEEE, 2015. (Cité en page 64), (Cité en page 69)

[15] S. Borkar. Thousand core chips : a technology perspective. In *Proceedings of the 44th annual design automation conference*, pages 746–749, 2007. (Cité en page 65)

[16] J. H. Brown and B. Martin. How fast is fast enough ? choosing between xenomai and linux for real-time applications. In *proc. of the 12th Real-Time Linux Workshop (RTLWS'12)*, pages 1–17, 2010. (Cité en page 48)

[17] S. Bünte, M. Zolda, M. Tautschnig, and R. Kirner. Improving the confidence in measurement-based timing analysis. In *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 144–151. IEEE, 2011. (Cité en page 96)

[18] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. H. Anderson, and S. K. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms., 2004. (Cité en page 107)

[19] P. Caspi and O. Maler. From control loops to real-time programs. In *Handbook of networked and embedded control systems*, pages 395–418. Springer, 2005. (Cité en page 30)

[20] F. J. Cazorla, J. Abella, J. Andersson, T. Vardanega, F. Vatrinet, I. Bate, I. Broster, M. Azkarate-Askasua, F. Wartel, L. Cucu, et al. Proxima : Improving measurement-based timing analysis through randomisation and probabilistic analysis. In *2016 Euromicro Conference on Digital System Design (DSD)*, pages 276–285. IEEE, 2016. (Cité en page 93)

[21] F. J. Cazorla, E. Quinones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, et al. Proartis : Probabilistically analyzable real-time systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s) :1–26, 2013. (Cité en page 93)

[22] D. Chen, A. Mok, and S. Baruah. On modeling real-time task systems. In *School organized by the European Educational Forum*, pages 153–169. Springer, 1996. (Cité en page 34)

[23] C. S. Committee et al. Iso international standard iso/iec 14882 : 2020, programming language c++. *Geneva, Switzerland : International Organization for Standardization (ISO)., Tech. Rep*, 2020. (Cité en page 116)

[24] A. Crespo, I. Ripoll, M. Masmano, P. Arberet, and J. Metge. Xtratum an open source hypervisor for tsp embedded systems in aerospace. *Data Systems In Aerospace DASIA, Istanbul, Turkey*, 2009. (Cité en page 33)

[25] M. Dalui and B. K. Sikdar. An efficient test design for cmps cache coherence realizing mesi protocol. In *Progress in VLSI Design and Test*, pages 89–98. Springer, 2012. (Cité en page 57)

[26] T. Drage, J. Kalinowski, and T. Braunl. Integration of drive-by-wire with navigation control for a driverless electric race car. *IEEE Intelligent transportation systems magazine*, 6(4) :23–33, 2014. (Cité en page 19)

[27] S. J. Eggers and R. H. Katz. Evaluating the performance of four snooping cache coherency protocols. In *Proceedings of the 16th annual international symposium on Computer architecture*, pages 2–15, 1989. (Cité en page 56)

[28] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener. Taclebench : A benchmark collection to support worst-case execution time research. In *16th International Workshop on Worst-Case Execution Time Analysis*, 2016. (Cité en page 104)

[29] S. Fan and B. C. Lee. Evaluating asymmetric multiprocessing for mobile applications. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 235–244. IEEE, 2016. (Cité en page 22)

[30] C. Ferdinand, R. Heckmann, and B. Franzen. Static memory and timing analysis of embedded systems code. In *Proceedings of VVSS2007-3rd European Symposium on Verification and Validation of Software Systems, 23rd of March*, pages 07–04, 2007. (Cité en page 90)

[31] C. Ferri, A. Viescas, T. Moreshet, R. I. Bahar, and M. Herlihy. Energy efficient synchronization techniques for embedded architectures. In *Proceedings of the 18th ACM Great Lakes symposium on VLSI*, pages 435–440, 2008. (Cité en page 72)

[32] W. Gay. Cross-compiling. In *Advanced Raspberry Pi*, pages 357–387. Springer, 2018. (Cité en page 142)

[33] G. Giannopoulou, K. Lampka, N. Stoimenov, and L. Thiele. Timed model checking with abstractions : Towards worst-case response time analysis in resource-sharing manycore systems. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 63–72, 2012. (Cité en page 95)

[34] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof : A call graph execution profiler. *SIGPLAN Not.*, 39(4) :49–57, 2004. (Cité en page 99)

[35] P. Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, 17, 2011. (Cité en page 65)

[36] R. Grisenthwaite. Armv8 technology preview. In *IEEE Conference*, 2011. (Cité en page 60)

[37] J. L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5) :532–533, 1988. (Cité en page 65)

[38] D. Hardy, B. Rouxel, and I. Puaut. The heptane static worst-case execution time estimation tool. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017. (Cité en page 90)

[39] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of wcet tools. *Proceedings of the IEEE*, 91(7) :1038–1054, 2003. (Cité en page 91)

[40] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7) :33–38, 2008. (Cité en page 65)

[41] A. Holdings. Arm cortex-a53 mpcore processor, technical reference manual, 2014. (Cité en page 54)

[42] N. Holsti and S. Saarinen. Status of the bound-t wcet tool. *Space Systems Finland Ltd*, 2002. (Cité en page 90)

[43] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *2010 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 108–109. IEEE, 2010. (Cité en page 56)

[44] T. Instruments. Architecture reference manual, omap4430 multimedia device silicon revision 2. x. *vol. SWPU231N*, 2010. (Cité en page 72)

[45] ISO. Road vehicles – Functional safety, 2011. (Cité en page 24)

[46] ISO. *ISO/IEC 9899 :2018 Information technology — Programming languages — C*. pub-ISO, pub-ISO :adr, June 2018. (Cité en page 101), (Cité en page 116)

[47] B. Jeff. Advances in big. little technology for power and energy savings. *ARM White paper*, page 33, 2012. (Cité en page 66)

[48] B. Jeff. big. little technology moves towards fully heterogeneous global task scheduling. *ARM white paper*, 2013. (Cité en page 66)

[49] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceedings of the 41st annual Design Automation Conference*, pages 275–280, 2004. (Cité en page 20)

[50] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5) :390–395, 1986. (Cité en page 38)

[51] Y. Joseph. The definitive guide to arm cortex-m3 and cortex-m4 processors. *ISBN-13*, pages 978–0124080829, 2014. (Cité en page 78)

[52] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd. Power7 : Ibm's next-generation server processor. *IEEE micro*, 30(2) :7–15, 2010. (Cité en page 56)

[53] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel. Cohesion : An adaptive hybrid memory model for accelerators. *IEEE micro*, 31(1) :42–55, 2011. (Cité en page 56)

[54] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Bus-aware multicore wcet analysis through tdma offset bounds. In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 3–12. IEEE, 2011. (Cité en page 107)

[55] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C. Wang. Heterogeneous computing : challenges and opportunities. *Computer*, 26(6) :18–27, 1993. (Cité en page 23)

[56] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 123–134. IEEE, 2008. (Cité en page 62)

[57] C. I. King. Stress-ng. *URL : http ://kernel. ubuntu. com/git/cking/stressng. git/(visited on 28/03/2018)*, 2017. (Cité en page 132)

[58] R. Kirner, I. Wenzel, B. Rieder, and P. Puschner. Using measurements as a complement to static worst-case execution time analysis. *Intelligent Systems at the Service of Mankind*, 2 :8, 2005. (Cité en page 92)

[59] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A practitioner's handbook for real-time analysis : guide to rate monotonic analysis for real-time systems*. Springer Science & Business Media, 2012. (Cité en page 38)

[60] Kokke. tiny-bignum-c, 2019. (Cité en page 129)

[61] M. Kostya. crystal-benchmarks-game, 2018. (Cité en page 129)

[62] R. Krishnamoorthy, K. Krishnan, B. Chokkalingam, S. Padmanaban, Z. Leonowicz, J. B. Holm-Nielsen, and M. Mitolo. Systematic approach for state-of-the-art architectures and system-on-chip selection for heterogeneous iot applications. *IEEE Access*, 9 :25594–25622, 2021. (Cité en page 22)

[63] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11), 2005. (Cité en page 63)

[64] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, pages 64–75. IEEE, 2004. (Cité en page 65)

[65] K. S. Lakshmanan, G. Bhatia, and R. Rajkumar. Autosar extensions for predictable task synchronization in multi-core ecus. Technical report, SAE Technical Paper, 2011. (Cité en page 71)

[66] E. Le Sueur and G. Heiser. Dynamic voltage and frequency scaling : The laws of diminishing returns. In *Proceedings of the 2010 international conference on Power aware computing and systems*, pages 1–8, 2010. (Cité en page 62)

[67] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. Chronos : A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3) :56–67, 2007. (Cité en page 90)

[68] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *2009 30th IEEE Real-Time Systems Symposium*, pages 57–67. IEEE, 2009. (Cité en page 107)

[69] D. J. Lilja. *Measuring computer performance : a practitioner's guide*. Cambridge university press, 2005. (Cité en page 98)

[70] B. Lisper. Sweet–a tool for wcet flow analysis. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 482–485. Springer, 2014. (Cité en page 90)

[71] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1) :46–61, 1973. (Cité en page 38)

[72] P. Lokuciejewski and P. Marwedel. *Worst-case execution time aware compilation techniques for real-time systems*. Springer Science & Business Media, 2010. (Cité en page 89)

[73] J. Luo and N. Jha. Static and dynamic variable voltage scheduling algorithms for real-time heterogeneous distributed embedded systems. In *Proceedings of ASP-DAC/VLSI Design 2002. 7th Asia and South Pacific Design Automation Conference and 15h International Conference on VLSI Design*, pages 719–726. IEEE, 2002. (Cité en page 37)

[74] G. Macher, A. Höller, E. Armengaud, and C. Kreiner. Automotive embedded software : Migration challenges to multi-core computing platforms. In *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, pages 1386–1393. IEEE, 2015. (Cité en page 39)

[75] S. Martello and P. Toth. Bin-packing problem. *Knapsack problems : Algorithms and computer implementations*, pages 221–245, 1990. (Cité en page 39)

[76] M. M. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 55(7) :78–89, 2012. (Cité en page 56)

[77] S. Maxwell and R. Roophnath. Miniature wireless quadcopter. In *ASEE 2014 Zone I Conference, University of Bridgeport, Bridgeport, CT, USA*, 2014. (Cité en page 31)

[78] N. Merriam, P. Gliwa, and I. Broster. Measurement and tracing methods for timing analysis. *International Journal on Software Tools for Technology Transfer*, 15(1) :9–28, 2013. (Cité en page 93)

[79] M. Mitić and M. Stojčev. An overview of on-chip buses. *Facta universitatis-series : Electronics and Energetics*, 19(3) :405–428, 2006. (Cité en page 58)

[80] K. S. Mohamed. Soc buses and peripherals : Features and architectures. In *IP Cores Design from Specifications to Production*, pages 77–96. Springer, 2016. (Cité en page 58)

[81] G. Moore. Moore's law. *Electronics Magazine*, 38(8) :114, 1965. (Cité en page 53)

[82] V. Nélis, P. M. Yomsi, and L. M. Pinho. Methodologies for the wcet analysis of parallel applications on many-core architectures. In *2015 Euromicro Conference on Digital System Design*, pages 748–755. IEEE, 2015. (Cité en page 97)

[83] NXP. *RPMsg-Lite User's Guide*. (Cité en page 82)

[84] NXP. Mcuxpresso. https://www.nxp.com/design/software/development-software/mcuxpresso-software-and-tools-/mcuxpresso-integrated-development-environment-ide:MCUXpresso-IDE, 2017. (Cité en page 158)

[85] C. S. Pabla. Completely fair scheduler. *Linux Journal*, 2009(184) :4, 2009. (Cité en page 44)

[86] C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron. The ROSACE case study : From simulink specification to multi/many-core execution. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*, pages 309–318. IEEE Computer Society, 2014. (Cité en page 113), (Cité en page 132)

[87] C. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. In *Proceedings 11th Real-Time Systems Symposium*, pages 72–81. IEEE, 1990. (Cité en page 87)

[88] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software engineering notes*, 17(4) :40–52, 1992. (Cité en page 29), (Cité en page 40)

[89] M. Poirier. In kernel switcher : A solution to support arm's new big. little technology. In *Embedded Linux Conference*, 2013. (Cité en page 66), (Cité en page 67)

[90] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-time systems*, 1(2) :159–176, 1989. (Cité en page 87)

[91] C. V. Ramamoorthy and H. F. Li. Pipeline architecture. *ACM Computing Surveys (CSUR)*, 9(1) :61–102, 1977. (Cité en page 54)

[92] R. Randhawa. Software techniques for arm big. little systems. *ARM, Apr*, 2013. (Cité en page 66)

[93] P. Rashinkar, P. Paterson, and L. Singh. *System-on-a-chip Verification : Methodology and Techniques*. Springer Science & Business Media, 2007. (Cité en page 63)

[94] F. Reghenzani, G. Massari, and W. Fornaciari. Mixed time-criticality process interferences characterization on a multicore linux system. In *2017 euromicro conference on digital system design (DSD)*, pages 427–434. IEEE, 2017. (Cité en page 47)

[95] F. Reghenzani, G. Massari, and W. Fornaciari. The real-time linux kernel : A survey on preempt_rt. *ACM Computing Surveys (CSUR)*, 52(1) :1–36, 2019. (Cité en page 46), (Cité en page 48)

[96] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt. Run : Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 104–115. IEEE, 2011. (Cité en page 107)

[97] L. Rierson. *Developing safety-critical software : a practical guide for aviation software and DO-178C compliance*. CRC Press, 2017. (Cité en page 20)

[98] O. Salvador and D. Angolini. *Embedded Linux Development with Yocto Project*. Packt Publishing Ltd, 2014. (Cité en page 141), (Cité en page 143)

[99] L. Santinelli, J. Morio, G. Dufour, and D. Jacquemart. On the sustainability of the extreme value theory for wcet estimation. In *14th International Workshop on Worst-Case Execution Time Analysis*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014. (Cité en page 93)

[100] N. Semiconductors. *i.MX 7ULP Applications Processor-Consumer Products Data Sheet*. NXP Semiconductors, 2020. (Cité en page 63)

[101] H. Shah, A. Coombes, A. Raabe, K. Huang, and A. Knoll. Measurement based wcet analysis for multi-core architectures. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, pages 257–266, 2014. (Cité en page 108)

[102] D. Shepelow and A. Fedorova. Scheduling on heterogeneous multicore processors using architectural signatures. *Proceedings of WIOSCA, at ISCA*, 35, 2008. (Cité en page 23)

[103] C. Simmonds. *Mastering Embedded Linux Programming*. Packt Publishing Ltd, 2017. (Cité en page 143), (Cité en page 147)

[104] J. Souyris, E. Le Pavec, G. Himbert, G. Borios, V. Jégu, and R. Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In *5th International Workshop on Worst-Case Execution Time Analysis (WCET'05)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007. (Cité en page 87)

[105] P. Stakem. Flightlinux : A new option for spacecraft embedded computers. In *2001 Earth Science Technology Conference, Collage Park, MD*, 2001. (Cité en page 46)

[106] J. A. Stankovic and R. Rajkumar. Real-time operating systems. *Real-Time Systems*, 28(2-3) :237–253, 2004. (Cité en page 45)

[107] D. B. Stewart. Measuring execution time and real-time performance. In *Embedded Systems Conference (ESC)*, volume 141, 2001. (Cité en page 97)

[108] STMicroelectronics. *Arm® dual Cortex®-A7 650 MHz+ Cortex®-M4 MPU, 3D GPU, TFT/DSI, 37 comm. interfaces, 29 timers, adv. analog, crypto*, 2020. (Cité en page 64), (Cité en page 68), (Cité en page 70)

[109] STMicroelectronics. Stm32cubeide. https://www.st.com/en/development-tools/stm32cubeide.html, 2020. (Cité en page 158)

[110] J. Sun, M. Jones, S. Reinauer, and V. Zimmer. *Embedded Firmware Solutions : Development Best Practices for the Internet of Things*. Springer Nature, 2015. (Cité en page 77)

[111] W.-T. Sun, E. Jenn, and H. Cassé. Build your own static wcet analyser : the case of the automotive processor aurix tc275. In *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*, 2020. (Cité en page 90)

[112] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Systems*, 18(2) :157–179, 2000. (Cité en page 107)

[113] I. S. . TM-2003. Ieee standard for information technology—standardized application environment profile (aep)—posix® realtime and embedded application support, 2003. (Cité en page 48)

[114] W. Tracz. Dssa (domain-specific software architecture) pedagogical example. *ACM SIGSOFT Software Engineering Notes*, 20(3) :49–62, 1995. (Cité en page 37)

[115] H. N. Tran, F. Singhoff, S. Rubini, and J. Boukhobza. Instruction cache in hard real-time systems : modeling and integration in scheduling analysis tools with aadl. In *2014 12th IEEE International Conference on Embedded and Ubiquitous Computing*, pages 104–111. IEEE, 2014. (Cité en page 94)

[116] K. H. Tsoi and W. Luk. Power profiling and optimization for heterogeneous multi-core systems. *ACM SIGARCH Computer Architecture News*, 39(4) :8–13, 2011. (Cité en page 113)

[117] A. Wafaa. Introducing the 64-bit armv8 architecture. In *Open Source Arm Ltd. EuroBSDCon conference, Malta*, page 68, 2013. (Cité en page 60)

[118] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3) :36, 2008. (Cité en page 87)

[119] W. Wolf. What is embedded computing ? *Computer*, 35(1) :136–137, 2002. (Cité en page 18)

[120] C. Wong, I. Tan, R. Kumari, J. Lam, and W. Fun. Fairness and interactive performance of o (1) and cfs linux kernel schedulers. In *2008 International Symposium on Information Technology*, volume 4, pages 1–8. IEEE, 2008. (Cité en page 43)

[121] W. Ya-gang. Analysis and transplant of embedded bootloader mechanism [j]. *Computer Engineering*, 6, 2010. (Cité en page 142)

[122] J. Yiu. *The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors*. Newnes, 2013. (Cité en page 103)

[123] Y. Zhang. Hackbench. *https ://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c*, 2008. (Cité en page 104)

Abstract

Recent heterogeneous multiprocessor systems on a chip (HMPSoCs), typically add a low-power processor optimized for real-time applications to one or more high-performance processors. This technology brings many challenges at several levels, especially for developers, as there is no development tool capable of handling deployment on heterogeneous clusters of identical cores. Another challenge concerns real-time applications, as these platforms are very complex, which makes it more difficult to measure the execution time of the code.

This thesis was carried out under a Cifre contract with AC6, whose objective is to support the research and development needed to support HMPSoCs, using an integrated development environment, System Workbench for Linux (SW4Linux), which allows the generation of a customized, secure and signed Linux distribution using a graphical user interface. When merged with a microcontroller development environment, the result is a unified tool for developing and debugging all HMPSoC components.

We have studied the different methods of measuring execution time. Numerous tests on different targets were performed with the aim of comparing these methods based on several attributes : accuracy, difficulty, granularity and resolution. It was found experimentally that most counter-based measurement methods give similar results, implying that the choice of methods should be based on simplicity and accessibility. These methods were used in a new feature added to the SW4Linux tool that allows to measure the execution time of a function and to extract the measurements automatically through a debugging session.

We presented a heterogeneous migration method, which allows a task to migrate between cores with different instruction sets and frequencies. This method is based on the asymmetric communication protocol. The communication latency was measured using a core-independent method accessible by different parts of the system. It was found that it is not negligible and varies significantly from core to core and that the inter-cluster migration was in the order of magnitude of several tens compared to the intra-cluster migration. This allowed to show experimentally that the classical academic model of defining heterogeneous platforms as ”flat” sets of heterogeneous cores was less suitable than a hierarchical cluster representation distinguishing the types (and thus the costs) of migration. Similarly, the experimental studies conducted in the thesis confirmed that HMPSoCs platforms had intermediate properties between heterogeneous cores and uniform cores, which placed these platforms in a new category that our co-authors have named consistent cores.

Keywords : Dynamic measurement, Timing analysis, Heterogeneous multiprocessors, System Workbench for Linux (SW4Linux), Multiprocessors, Computer scheduling, Real-time data processing, Embedded computer systems, Systems on a chip

Résumé

Les systèmes multiprocesseurs hétérogènes sur une puce récents (HMPSoCs), ajoutent généralement un processeur à basse consommation et optimisé pour les applications temps réel à un ou plusieurs clusters de processeurs performants. Cette technologie entraîne de nombreux défis à plusieurs niveaux, en particulier pour les développeurs, car il n'existe pas d'outil de développement capable de gérer le déploiement sur des clusters hétérogènes de cœurs identiques. Un autre défi concerne les applications temps réel, car ces plateformes sont très complexes ce qui rend plus difficile la mesure de durée d'exécution du code.

Cette thèse a été effectuée en contrat Cifre avec AC6, dont l'objectif est de soutenir la recherche et le développement nécessaires pour supporter les HMPSoCs, grâce à un environnement de développement intégré, System Workbench for Linux (SW4Linux) qui permet de générer une distribution Linux personnalisée, sécurisée et signée, en utilisant une interface graphique. En le fusionnant avec un environnement de développement pour microcontrôleurs, on obtient un outil unifié qui permet de développer et déboguer tous les éléments du HMPSoC.

Nous avons étudié les différentes méthodes de mesure du temps d'exécution. De nombreux tests sur différentes cibles ont été effectués dans le but de comparer ces méthodes en fonction de plusieurs attributs : la précision, la difficulté, la granularité et la résolution. Il a été constaté expérimentalement que la plupart des méthodes de mesure basées sur des compteurs donnent des résultats similaires, ce qui implique que le choix des méthodes doit reposer sur la simplicité et l'accessibilité. Ces méthodes ont été utilisées dans une nouvelle fonctionnalité ajoutée à l'outil SW4Linux permettant de mesurer le temps d'exécution d'une fonction et d'extraire les mesures automatiquement grâce à une session de débogage.

Nous avons présenté une méthode de migration hétérogène, qui permet à une tâche de migrer entre des cœurs ayant des jeux d'instructions et des fréquences différentes. Cette méthode est basée sur le protocole de communication asymétrique. Le temps de latence de communication a été mesuré à l'aide d'une méthode indépendante du cœur et accessible par les différentes parties du système. Il a été constaté qu'il n'est pas négligeable et varie beaucoup d'un cœur à l'autre et que la migration inter-clusters était d'un degré de magnitude de plusieurs dizaines comparé à la migration intra-clusters. Cela a permis de montrer expérimentalement que le modèle académique classique qui consiste à définir les plateformes hétérogènes comme des ensembles `"plats"` de cœurs hétérogène était moins adaptée qu'une représentation hiérarchique par clusters distinguant les types (et donc les coûts) de migration. De même, les études expérimentales menées dans la thèse ont permis de confirmer que les plateformes HMPSoCs avaient des propriétés intermédiaires entre cœurs hétérogènes et cœurs uniformes, qui plaçaient ces plateformes dans une nouvelle catégorie que nos co-auteurs ont dénommée à cœurs consistants.

Mots-clés : Mesure dynamique, Analyse de temps, Multiprocesseurs hétérogènes, System Workbench for Linux (SW4Linux), Ordonnancement (informatique), Temps réel (informatique), Systèmes embarqués (informatique), Systèmes sur puce

---

Secteur de recherche : Informatique et applications

Laboratoire d'Informatique et d'Automatique pour les Systèmes
Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
Téléport 2 – 1 Av. Clément Ader – BP 40109 – 86961 FUTUROSCOPE CHASSENEUIL CEDEX
Tél : 05.49.49.80.63 – Fax : 05.49.49.80.64