Maîtrise de la couche hyperviseur sur les architectures multi-coeurs COTS dans un contexte avionique

Xavier Jean

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUT OF TECHNOLOGY

TELECOM ParisTech

2015-ENST-0034

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité « Informatique »

présentée et soutenue publiquement par

Xavier JEAN

le 18 juin 2015

Maîtrise de la couche hyperviseur sur les processeurs multi-cœurs COTS dans un contexte avionique

Directeur de thèse	:	**Laurent PAUTET**
Co-encadrant de la thèse	:	**Thomas ROBERT**
Encadrant industriel	:	**Marc GATTI**
Co-encadrant industriel	:	**David FAURA**

Jury :

M. Yves MATHIEU, Professeur, Telecom ParisTech	Président
M. Laurent GEORGE, Professeur, Université Paris Est Marne la Vallée	Rapporteur
M. Alain MÉRIGOT, Professeur, Université Paris Sud	Rapporteur
Mme. Claire PAGETTI, Ingénieur de recherche HDR, Onera, Toulouse	Examinatrice
M. Mathieu JAN, Ingénieur - Chercheur, CEA Saclay	Examinateur
M. Laurent PAUTET, Professeur, Telecom ParisTech	Directeur de thèse
M. Thomas ROBERT, Maître de conférence, Telecom ParisTech	Co-encadrant
M. Marc GATTI, Directeur de l'innovation, Thales Avionics	Encadrant industriel

TELECOM ParisTech
école de l'Institut Mines-Télécom - membre de ParisTech
46 rue Barrault 75013 Paris - (+33) 1 45 81 77 77 - www.telecom-paristech.fr

T H È S E

– "Papa, c'est quoi ta thèse ?"

– "Ça parle d'avions et d'ordinateurs, Manu"

– "Pourquoi ?"

– "Parce qu'aujourd'hui, les avions sont contrôlés par des ordinateurs et qu'il faut les rendre sûrs, mais aussi plus efficaces. C'est pour cela que l'on veut utiliser des processeurs multi-cœurs."

– "Pourquoi ?"

– "Parce que comme ça on peut exécuter plusieurs logiciels en même temps. Le problème, c'est que ces logiciels interfèrent, donc c'est difficile de savoir si l'ordinateur est vraiment sûr."

– "Pourquoi ?"

– "Parce qu'ils utilisent des ressources du processeur en même temps, et qu'on ne sait pas comment le processeur partage ces ressources."

– "Pourquoi ?"

...

02/05/2015

Extrait d'une conversation avec Manu.

Remerciements

Ma thèse est finie.

Ouf !

La côte était à quatre contre un.

Et pourtant, que ce furent de belles années. Et beaucoup de gens y ont contribué.

***

Mes premiers remerciements iront à mon jury. Je tiens à remercier M. Yves Mathieu, qui a eu la gentillesse de le présider, mes rapporteurs, MM. Laurent George et Alain Mérigot, qui ont relu mon manuscrit avec une attention et une bienveillance qui m'ont touché, et enfin mes examinateurs, Mme. Claire Pagetti et M. Mathieu Jan, qui ont contribué à faire de la soutenance un moment agréable.

Je remercie également mes encadrants, Marc Gatti, David Faura, Thomas Robert et Laurent Pautet, pour leur accompagnement tout au long de cette thèse. On dit souvent que la thèse est un parcours initiatique pour le thésard, mais c'est également une épreuve de patience pour ses encadrants.

***

J'ai connu trois lieux de travail pendant ma thèse, et j'ai eu la chance de pouvoir m'épanouir dans chacun de ces endroits et d'y faire des amitiés durables.

Je pense en premier lieu à mes amis stagiaires, doctorants, post-doctorants ou permanents au sein du département INFRES à Telecom ParisTech. De manière non exhaustive, je remercie Cuauhtemoc C., Elie R., Simon P., Smail R., Romain G., Antoine J., Antoine S., Guang Z., Gilles L., Sylvain F., Jonathan M., Imen b.D, Bertrand M., Marilena O., Sébastien G., Etienne B., Nora D., Damien M., Asma S., Sarah N., Grzegorz L., Robin D., Quentin R., Azin A., Siwar R., Paul-Louis A.. Vous contribuez – ou avez contribué – à faire de ce laboratoire un lieu de convivialité où il fait toujours bon venir. Un mot de remerciement également pour Florence Besnard pour sa disponibilité et sa gentillesse au secrétariat de l'EDITE.

Une pensée pour mes anciens collègues de Thales Avionics, Thomas M., Anthony R., Vincent B., Patrice T., Antoine M., Yves M., Patrick D., Philippe L., Philippe Bu., Philippe Bi., Sébastien C., Xavier M., Martin R., Paul B., Vincent S., Stéphane B., Guy B., Joël B., Didier R., Valentin B., Henri L., Vanessa M., Alexandre G., Simon F., Alexandre T., Nicolas B. Vous avoir a été une grande richesse pour m'imprégner de la culture du domaine avionique. Vous avez aussi été d'un grand soutien dans les moments

difficiles. Je vous en suis très reconnaissant, et je vous souhaite le meilleur pour l'aventure post-Vélizy.

Beaucoup de gratitude également envers mes amis (anciens) thésards à Thales Avionics, tant à Meudon qu'à Pessac, Michaël L., Antoine C., Sébastien T., Alexandre B., Romain M. et Aurélie B. C'était de belles années au sein de la DT de CCC avec vous. Vous avez fait de belles choses et j'en garde un souvenir à la fois ému et admiratif.

Je remercie particulièrement mes anciens stagiaires à Thales Avionics, Jérémie T., Gaëlle C., Caroline Q., Joffrey C. et Bodivann K. Ça a été une chance de vous avoir et je pense que sans vous, ma thèse aurait été différente.

Un grand merci enfin à mes nouveaux collègues au sein du LSEC à Thales Research&Technology, Philippe B., Jaime d.O., Madeleine F., Sylvain G., Daniel G.P., Jimmy l.R, Arnaud G., Rafik H, Jean-Marc M., Romain S., Nicolas S., Laurent R., Charles R., Éric D., Gérard C., Éric L. Merci déjà de m'avoir laissé la marge de manœuvre nécessaire pour finaliser le manuscrit et soutenir ma thèse, en plus des projets de l'équipe. Merci également de votre soutien dans cette période pas facile à négocier. De mon côté, plus que deux mois de retard à rattraper sur les projets à ce jour :-)

***

Je pense enfin à ma sphère familiale et mes amis plus anciens, Jean-Philippe, Brice, Ronan, ADG, Sonia, Alex, qui ont toujours été là pour me rappeler qu'il n'y a pas que la thèse dans la vie. Un grand merci également à mes parents, mes frères, ma sœur, mes (futures) belles-sœurs, ainsi qu'à ma belle famille, pour ces années et les quelques unes qui les ont précédées. Je repense aux quelques frasques que j'ai pu faire pendant ma scolarité et finalement, je me dis que la thèse s'inscrivait dans la continuité des évènements.

Une pensée tendre pour mes enfants, Emmanuel et Clémence, que j'ai eu la joie d'avoir au cours de ma thèse, et qui m'ont toujours permis de relativiser sur l'importance du travail dans la vie quotidienne. Faire une thèse avec des enfants n'est pas de tout repos, mais c'est une belle aventure que je recommande à tout doctorant.

Le mot de la fin est pour Claire, mon épouse, avec qui je me suis marié au tout début de ma thèse, et qui m'a accompagné durant toutes ces années. Tu m'as remotivé quand il le fallait, tu as assuré quand j'en avais besoin. Assurément, c'est grâce à toi que ces années ont été belles.

Résumé

Nous nous intéressons dans cette thèse à la maîtrise de processeurs multi-cœurs COTS dans le but de les rendre utilisables dans des équipements avioniques, qui ont des exigences temps-réel dures. L'objectif est de permettre l'application de méthodes connues d'évaluation de pire temps d'exécution (WCET) sur un ensemble de tâches représentatif d'applications avioniques. Au cours de leur exécution, les tâches exécutées sur différents cœurs vont accéder simultanément à des ressources matérielles qui sont partagées entre les cœurs, en particulier la mémoire principale. Certains de ces accès devront être mis en attente par le matériel. On parle d'interférences. Les méthodes d'évaluation de WCET demandent que la pénalité liée aux interférences soit bornée. Dans notre contexte, marqué par l'absence de connaissance globale des tâches embarquées, ainsi que l'utilisation de processeurs COTS dont l'architecture est complexe est peu documentée, on ne sait pas déterminer cette borne. Nous nous intéressons à une famille d'approches qui consistent à organiser les différentes tâches pour qu'elles n'utilisent jamais les ressources communes en même temps. Cependant ces approches ne permettent pas de réutiliser du logiciel avionique existant. Nous cherchons à les améliorer en introduisant un concept de "logiciel de contrôle", qui vise à se substituer aux tâches embarquées, pour réaliser lui-même les accès aux ressources qu'elles initient. Ce logiciel de contrôle doit assurer une isolation temporelle des accès concurrents, tout en permettant la réutilisation de logiciel avionique existant. Nous étudions dans cette thèse le problème de l'existence d'un tel logiciel de contrôle, et celui de son efficacité vis-à-vis des tâches embarquées.

Nous formalisons la notion de logiciel de contrôle en nous appuyant sur un cadre de virtualisation issu de la littérature. Nous introduisons ensuite le théorème de contrôlabilité, qui conditionne l'existence d'un tel logiciel de contrôle. Ce théorème généralise un théorème de virtualisation issu du cadre précédent. Nous proposons ensuite une méthodologie permettant d'évaluer sur un processeur donné l'applicabilité du théorème de contrôlabilité. Nous appliquons cette méthodologie sur un processeur COTS à quatre cœurs, appartenant à la gamme QorIQ fabriquée par Freescale. Nous obtenons à l'issue de cette méthode une spécification de certains mécanismes à mettre en œuvre dans le logiciel de contrôle, ainsi que des propriétés sur l'état du cœur que ce dernier doit vérifier.

Nous avons réalisé un prototype du logiciel de contrôle implémentant cette spécification, sur lequel nous avons étudié la problématique d'efficacité. Cette évaluation a porté sur des benchmarks qui couvrent divers comportements algorithmiques, ainsi qu'une application avionique de taille moyenne. Nous avons constaté que le logiciel de contrôle élimine les interférences, et offre un temps d'exécution stable quel que soit les tâches exécutées sur les autres cœurs, mais qu'il a un impact sur le temps d'exécution individuel de chaque tâche. Cet impact est plus ou moins élevé selon le type d'algorithme évalué et ses paramètres d'entrée, mais pour chaque algorithme nous avons identifié un domaine d'usage dans lequel l'impact du logiciel de contrôle est limité. Nous envisageons enfin diverses pistes pour optimiser l'efficacité du logiciel de contrôle sur un type d'application donné.

Mots-clés: Processeur multi-cœurs, logiciel de contrôle, virtualisation, hyperviseur, Avionique Modulaire Intégrée

Abstract

We focus in this thesis on issues related to COTS multi-core processors mastering, especially regarding hard real-time constraints, in order to enable their usage in future avionics equipment. We aim at applying existing Worst Case Execution Time (WCET) evaluation methods on a set of tasks similar to those we can find in avionics software. At runtime, tasks executed among different cores are likely to access hardware resources at the same time, e.g. the main memory. It may lead to additional delays due to hardware contention, called "interferences". WCET evaluation methods expect that the penalty due to interferences is bounded. We have an industrial context that states we don't know the whole set of tasks, and we use COTS processors whose architecture is complex and partially undocumented. Given this context, such an interferences penalty cannot be assessed. We focus on a family of approaches that aim at organizing the embedded software to avoid simultaneous use of shared resources. However those approaches do not allow any reuse of legacy avionic software. We try to improve them through a concept of control software, which aims at replacing embedded tasks to manage their attempts to access shared resources. That control software enforces some time isolation of concurrent activities from each core, and enables some reuse of legacy avionics software. We study in this thesis the problem of control software's existence and efficiency over a given set of tasks.

We formalize our control software concept from an existing framework that applies on virtualization software. Then, we introduce the controllability theorem, which details existence conditions for a piece of control software. This theorem generalizes a virtualization theorem defined in the previous framework. We propose further a method that assess on a given processor the possibility to apply the controllability theorem. We apply this methodology on a quad-core COTS processor from QorIQ series manufactured by Freescale. We get through this method a specification of some software mechanisms to implement inside the control software, and some properties on the CPU's internal state the control software must ensure.

We realized a prototype of control software that implements this specification. We studied the efficiency of this prototype over a set of benchmarks that highlight different ways to use shared resources, and a mid-size avionics application. We could see that the control software removes interference, and thus ensures each task is not impacted by concurrent tasks. The individual impact of control software varies among the evaluated algorithms and their input parameters. However, for each of them we found a usage domain for which this impact remains acceptable. Finally, we propose some clues to optimize the control software's efficiency for specific applications.

Keywords: Multi-core processors, control software, virtualization, hypervisor, Integrated Modular Avionics

Table des matières

Table des matières **ix**

I Introduction générale 1

1 Introduction 3

1.1 Contexte 3

1.2 Problématique 4

1.3 Résumé des contributions 5

1.4 Plan du mémoire 6

II Enjeux industriels et scientifiques 7

2 Contexte Avionique 9

2.1 Exigence de déterminisme 10

2.1.1 Principes de sûreté de fonctionnement 11

2.1.2 Définition de la notion de déterminisme 13

2.1.3 Contraintes temps réel et déterminisme temporel 13

2.1.4 Problématiques liées aux contraintes temps réel 14

2.2 Technologies embarquées dans les systèmes avioniques 16

2.2.1 Évolution des systèmes avioniques 16

2.2.2 Problématiques liées à l'usage de processeurs COTS 17

2.2.3 Problématiques liées aux technologies semi-conducteurs . . . 18
2.3 Avionique Modulaire Intégrée (IMA) . . . 20
2.3.1 Architecture d'un système IMA . . . 21
2.3.2 Propriété de Partitionnement Robuste . . . 23
2.3.3 Acteurs industriels d'un système IMA . . . 24
2.3.4 Profil des applications déployées dans les systèmes IMA . . . 25
2.4 Processeurs multi-cœurs pour l'avionique . . . 25
2.4.1 Architectures multi-cœurs . . . 26
2.4.2 Problématiques spécifiques aux processeurs multi-cœurs . . . 29
2.5 Synthèse . . . 31

3 Évaluation du WCET dans les processeurs multi-cœurs 33

3.1 Méthodes existantes dans un contexte mono-cœur . . . 34
3.1.1 Vue d'ensemble . . . 35
3.1.2 Spécificités des méthodes statiques . . . 36
3.1.3 Spécificités des méthodes dynamiques . . . 37
3.1.4 Utilisation dans un processus industriel . . . 38
3.2 Applicabilité des méthodes mono-cœurs dans un cadre multi-cœurs . 38
3.2.1 Prédictibilité des processeurs multi-cœurs . . . 39
3.2.2 Gestion du manque d'information sur les processeurs COTS . 40
3.2.3 Techniques d'analyse d'interférences . . . 41
3.3 Alternatives pour l'évaluation du WCET sur des processeurs multi-cœurs 45
3.3.1 Approches par pénalité d'interférences globale . . . 45
3.3.2 Approches par processeurs déterministes . . . 46
3.3.3 Approches par logiciel déterministe . . . 47
3.4 Synthèse . . . 49

III Démarche scientifique 51

4 Problématique 53

4.1 Problème général 53

4.2 Résumé des approches existantes 54

4.3 Problématique de thèse 57

4.4 Synthèse 58

5 Approche 59

5.1 Définition et existence du logiciel de contrôle 59

5.2 Prototypage et étude de l'efficacité du logiciel de contrôle 61

5.2.1 Vue d'ensemble du prototype 61

5.2.2 Efficacité du logiciel de contrôle 62

IV Contributions 63

6 Stratégie de contrôle du processeur 65

6.1 Cadre de virtualisation de Popek et Goldberg 66

6.1.1 Définition d'un hyperviseur 66

6.1.2 Théorème de virtualisation de Popek et Goldberg 68

6.2 Cadre analogue pour le logiciel de contrôle 71

6.2.1 Définition du logiciel de contrôle 71

6.2.2 Théorème de contrôlabilité 74

6.3 Application du théorème de contrôlabilité 76

6.3.1 Vue d'ensemble de la démarche 76

6.3.2 Modèle de cœur considéré 78

6.3.3 Définitions des évènements sensibles 79

6.3.4 Classification des évènements sensibles 81

6.3.5 Construction d'un invariant 83
6.3.6 Construction des séquences de contrôle 87
6.4 Synthèse 90

7 Cas d'étude sur un processeur COTS 91
7.1 Cible matérielle 92
7.1.1 Gamme de processeurs ciblée et choix de la cible 92
7.1.2 Description du cœur e5500 94
7.2 Description du cas d'étude 97
7.3 Identification des évènements sensibles 99
7.3.1 Évènements associés au pipeline 101
7.3.2 Évènements associés à la MMU 103
7.3.3 Évènements associés aux caches 105
7.3.4 Construction des arbres d'activité au niveau du cœur 107
7.4 Classification des évènements sensibles 112
7.4.1 Classification des évènements sensibles 113
7.4.2 Construction de l'invariant 114
7.4.3 Robustesse de l'invariant vis-à-vis du logiciel utile 121
7.5 Construction des séquences de contrôle 123
7.5.1 Exemple de séquence de contrôle par neutralisation 125
7.5.2 Exemple de séquence de contrôle par émulation 127
7.6 Synthèse 129

8 Efficacité du logiciel de contrôle 131
8.1 Vue d'ensemble de Marthy 132
8.1.1 Principaux modules logiciels 132
8.1.2 Mécanisme de démarrage des cœurs secondaires 133
8.1.3 Organisation du cache L2 134

8.2 Démarche d'évaluation 134
8.3 Évaluation de benchmarks industriels 137
8.3.1 Efficacité du logiciel de contrôle sur Stream 137
8.3.2 Efficacité du logiciel de contrôle sur AES 138
8.3.3 Efficacité du logiciel de contrôle sur FFT 140
8.4 Evaluation d'une application avionique 141
8.4.1 Efficacité du logiciel de contrôle sur l'application avionique . 142
8.4.2 Pistes pour l'optimisation du logiciel de contrôle 143
8.5 Synthèse 144

V Conclusion générale 145

9 Conclusion 147

9.1 Résumé des travaux effectués 147
9.1.1 Existence du logiciel de contrôle 148
9.1.2 Efficacité du logiciel de contrôle 149
9.1.3 Limites de l'approche 150
9.2 Perspectives 150
9.2.1 Perspectives à court terme 151
9.2.2 Perspectives à long terme 152

Annexes 155

A Complément à l'étude de cas sur le P5040 157

A.1 Description du P5040 157
A.2 Complément à la première analyse du jeu d'instruction PowerPC . . 161
A.3 Arbres d'activités des différents composants du cœur e5500 165

B Techniques de virtualisation **169**

B.1 Vue générale du concept de virtualisation 169

B.2 Classification des hyperviseurs existants 172

B.2.1 Classification par environnement d'exécution 172

B.2.2 Classification par technique de virtualisation 173

B.3 Concepts implémentés dans un hyperviseur 176

B.3.1 Virtualisation de l'espace adressable 178

B.3.2 Virtualisation des périphériques 179

Bibliographie **181**

Liste des illustrations

2.1 Vue d'ensemble d'un système et d'un module IMA 20
2.2 Modèle d'un calculateur comportant un processeur multi-cœurs 26
2.3 Description d'un cœur PowerPC de type e5500 (issue du manuel de référence [43]) 27
2.4 Exemple de déploiement de partitions en configuration AMP 30

3.1 Vue d'ensemble des différentes étapes du calcul d'un WCET 35
3.2 Chronogramme représentant l'activité interne d'un processeur PowerPC exécutant une séquence d'instructions simple 40
3.3 Vue d'ensemble de la configuration d'un cœur dans l'approche par logiciel de contrôle de Jegu [54] 48

5.1 Vue d'ensemble du déploiement du prototype sur un processeur générique 61
5.2 Vue d'ensemble de la démarche d'évaluation de l'efficacité du prototype . 62

6.1 Illustration de la propriété d'équivalence [69] 67
6.2 séquences d'exécution du logiciel de contrôle et du logiciel invité 68
6.3 Déploiement du logiciel de contrôle 71
6.4 Classification des évènements sensibles 72
6.5 Exemple d'arbre d'activité d'une opération de lecture sur un cache 77
6.6 Vue d'ensemble des traitements à effectuer selon la sensibilité et la contrôlabilité des évènements 78
6.7 Modélisation d'un cœur à deux niveaux de caches internes 79
6.8 Activité d'un cache sollicité par une requête en lecture 80
6.9 Activité d'un cache sollicité par une requête en écriture 81
6.10 Arbre d'activités associé à un évènement de type *load* 82
6.11 Construction de l'invariant à partir de l'arbre d'activités 84
6.12 Vue d'ensemble d'une séquence de contrôle 86

6.13 Processus de synchronisation sur une fenêtre temporelle 87

7.1 Vue d'ensemble de la gamme QorIQ 93
7.2 Vue d'ensemble du processeur P5040 et des périphériques associés 94
7.3 Vue d'ensemble du cœur e5500 95
7.4 Description du cœur e5500 issue du manuel de référence [43] 101
7.5 Arbre d'activité d'une MMU traitant un évènement de type *load* ou *store* 104
7.6 Arbre d'activité du DLFB traitant un évènement de type *write* 106
7.7 Arbre d'activité du cache L_1D traitant un évènement de type *write* . . . 107
7.8 Principaux arbres d'activité du cache L_2 108
7.9 Arbre d'activité du cœur e5500 sur un évènement de type *load* 110
7.10 Arbre d'activité du cœur e5500 sur un évènement de type *fetch* 111
7.11 Arbre d'activité du cœur e5500 sur un évènement de type *flush* 112
7.12 Sémantique de l'instruction *mfspr*, décrite dans le manuel de référence des cœurs e500 [41] 128

8.1 Vue d'ensemble de l'architecture logicielle de Marthy 132
8.2 Mapping de la mémoire garantissant la séparation des données entre cœurs134
8.3 Organisation du cache L_2 effectuée par Marthy 135
8.4 Vue d'ensemble de la démarche d'évaluation de l'efficacité du prototype . 135
8.5 Mesures de temps d'exécution de l'application de chiffrement AES dans les différentes configurations 139
8.6 Mesures de temps d'exécution de l'application de FFT dans les différentes configurations 141
8.7 Mesures de temps d'exécution de l'application avionique 142

A.1 Vue d'ensemble du processeur P5040 et des périphériques associés 158
A.2 Exemple d'activité nécessitant l'émission de plusieurs requêtes sur CoreNet 159
A.3 Arbre d'activité d'une MMU traitant un évènement de type *fetch* 165
A.4 Arbres d'activités associés au cache L1D 166
A.5 Arbres d'activités associés au cache L1I 167
A.6 Arbres d'activités associés au cache L2 167

B.1 Intégration d'un logiciel de virtualisation selon Smith et Nair [89] 170
B.2 Illustration des techniques de virtualisation. 173
B.3 Virtualisation de l'espace adressable physique par un hyperviseur 178

Liste des tableaux

2.1 Niveaux de criticité des scénarii de défaillance 12

6.1 Terminologies analogues 72
6.2 Classification des évènements définis sur le modèle 83

7.1 Classes d'instructions considérées dans les deux analyses 100
7.2 Evènements associés à l'exécution d'instructions PowerPC 104
7.3 Classification des évènements émis par le pipeline 115
7.4 Classification des évènements du cœur e5500 (hors pipeline) 116

8.1 Bandes passantes mesurées par Stream dans les différentes configurations 138

A.1 Evènements associés aux instructions PowerPC supportées par le cœur [85, section 6.4] 162
A.2 Evènements associés aux instructions PowerPC supportées par le cœur (suite) [85, section 6.4] 163
A.3 Evènements associés aux instructions PowerPC supportées par le cœur [85, section 6.4] (suite et fin) 164

B.1 Classification des hyperviseurs existants 177

Première partie

Introduction générale

Chapitre 1

Introduction

Sommaire

1.1 Contexte 3
1.2 Problématique 4
1.3 Résumé des contributions 5
1.4 Plan du mémoire 6

1.1 Contexte

Dans les avions de dernière génération, de plus en plus de systèmes critiques font intervenir de l'électronique numérique et des composants logiciels. Pour chaque composant matériel et logiciel embarqués dans de tels systèmes, il est nécessaire d'apporter des garanties sur son bon fonctionnement. On parle informellement de satisfaire une *exigence de déterminisme*. Cette notion (cf. section 2.1.2) couvre des exigences portant sur le comportement du logiciel vis-à-vis de sa spécification, ainsi que le respect de contraintes temporelles.

La plupart des systèmes avioniques critiques sont dits "temps réel durs". Cela signifie qu'ils doivent remplir leur fonction dans une durée finie. Quand ils embarquent des tâches logicielles, on associe à ces tâches des échéances, qui sont en général périodiques ou bien définies à partir d'évènements externes. Une analyse d'ordonançabilité peut être effectuée pour montrer que chaque tâche respecte son échéance. Cependant, cette analyse suppose que toutes les tâches complètent leur exécution en un temps fini si elles sont exécutées seules. Ce problème est habituellement abordé par des techniques d'évaluation de *pire temps d'exécution* (ou WCET).

Depuis le milieu des années 2000, la communauté avionique considère les processeurs multi-cœurs comme de bons candidats pour équiper les prochaines générations de systèmes embarqués. L'enjeu est à la fois d'obtenir des systèmes plus efficaces en termes de puissance de calcul, plus économes en consommation et en place occupée, mais également de trouver une solution pérenne à l'obsolescence des processeurs mono-cœurs.

Comme toute nouvelle technologie, l'usage de processeurs multi-cœurs dans les systèmes avioniques ne doit pas entraîner de régression au niveau de l'exigence de déterminisme. A ce titre, il est nécessaire de montrer qu'il est possible d'évaluer le WCET d'un ensemble d'applications exécutées sur un processeurs multi-cœurs. Nous étudions ce problème général dans cette thèse.

Nous considérons la question de l'évaluation du WCET sur un processeur multi-cœurs en nous appuyant sur le contexte industriel suivant :

Systèmes IMA. Nous considérons des systèmes avioniques développés selon le concept d'Avionique Modulaire Intégrée (IMA). Ce concept vise à permettre la coexistence sur une même plateforme –composée d'un processeur et d'un système d'exploitation– de plusieurs applications avioniques de niveaux de criticité hétérogènes. Le système d'exploitation met en œuvre la propriété de *Partitionnement Robuste*, qui garantit que chaque application peut être certifiée de manière indépendante.

Processeurs COTS. Nous supposons que les processeurs utilisés sont des processeurs COTS [1], c'est-à-dire achetés sur étagère auprès d'un fabricant. Ce dernier vise des marchés larges pour une production de masse, et ne partage pas l'exigence de déterminisme propre aux systèmes avioniques. La difficulté consiste alors à apporter des garanties sur le comportement du processeur, sachant que notre niveau de connaissances sur celui-ci est limité aux informations que le fabricant accepte de divulguer, et aux tests que l'on peut effectuer en interne.

Exigence de rétro-compatibilité. Les coûts de redéveloppement et de certification du logiciel avionique existant sont élevés. Il est difficilement envisageable d'introduire de changement majeur dans les méthodes courantes de développement du logiciel. Nous souhaitons donc permettre la réutilisation de logiciel existant avec un effort d'adaptation minimal.

1.2 Problématique

Dans cette thèse, nous cherchons à appliquer des méthodes d'évaluation de WCET issues de l'état de l'art sur un ensemble de tâches représentatives d'applications avioniques, exécutées sur un processeur multi-cœurs COTS.

Les méthodes existantes supposent entre autres que le processeur est prédictible, c'est-à-dire que les temps d'accès des cœurs à des ressources matérielles partagées, comme la mémoire principale, sont bornés. Cependant, au cours de son exécution, le logiciel exécuté sur chaque cœur accèdera à des ressources matérielles partagées entre tous les cœurs. Ces dernières n'ayant pas la capacité de traiter simultanément des accès issus de tous les cœurs, il s'ensuivra des mises en attentes de certains accès que nous appelons *interférences*. Cependant, sur des processeurs COTS, la gestion par le processeur des accès

1. Component Off-The-Shelf, composant sur étagère

aux ressources partagées est mal décrite. Par conséquent, l'identification des situations d'interférences et de leur impact sur le temps d'exécution du logiciel est un problème ouvert, dans le contexte dans lequel nous l'abordons.

Certaines approches présentes dans l'état de l'art rendent le processeur prédictible en éliminant les interférences. Ces dernières garantissent à chaque cœur un accès exclusif aux ressources communes dans certaines plages temporelles. On parle de politique TDMA [2]. Pour appliquer cette approche à notre contexte, il est nécessaire de contrôler les dates d'accès des cœurs aux ressources partagées. Or les processeurs COTS ne proposent pas de mécanismes prévus pour réaliser ce contrôle. Nous envisageons donc l'introduction de tels mécanismes dans un logiciel dédié, que nous appelons *logiciel de contrôle*.

Nous étudions dans cette thèse le concept de logiciel de contrôle, ce contrôle s'exerçant sur les accès des cœurs aux ressources communes. Nous traitons ce problème à travers les points suivants :

Définition et existence. Le concept de logiciel de contrôle visera en particulier à éliminer les occurrences d'évènements matériels dits *sensibles*, i.e. menant à des accès aux ressources dont la date n'est pas maîtrisée. Nous cherchons à déterminer sous quelles conditions l'existence d'un tel logiciel peut être démontrée, et comment ces conditions peuvent être remplies. L'approche que l'on adoptera pour remplir ces conditions sera appelée *stratégie de contrôle* du processeur.

Efficacité. La présence d'un logiciel de contrôle a un impact sur le temps d'exécution du logiciel utile, i.e. qui subit le contrôle. Pour un processeur donné, il est nécessaire de connaître les types d'application pour lesquels cet impact est acceptable et ceux pour lesquels il ne l'est pas.

Nous présentons dans la section suivante un résumé des solutions apportées aux problèmes décrits ci-dessus.

1.3 Résumé des contributions

Comme expliqué ci-dessus, nous traitons notre problématique en deux temps, à savoir la question de l'existence d'un logiciel de contrôle sur un processeur donné, et celle de son efficacité. Les contributions de cette thèse s'articulent autour de ces deux points.

Existence du logiciel de contrôle

Nous formalisons le concept de logiciel de contrôle par analogie avec la notion de *gestionnaire de machines virtuelles* de Popek et Goldberg [69], encore appelé *hyperviseur* dans la littérature plus récente. Cet hyperviseur est présenté comme un logiciel de contrôle, ce contrôle portant sur l'allocation des ressources matérielles du processeur au logiciel utile, qui est effectuée lors de l'exécution d'instructions dites *sensibles*.

2. Time Division Multiple Access

Notre logiciel de contrôle cherche à maîtriser non pas l'allocation des ressources au logiciel utile mais les dates auxquelles ce dernier y accède. Ces accès ont lieu lors de l'exécution de certaines instructions, mais pas seulement. Nous étendons la notion d'instruction sensible à celle *d'évènements matériels* dits sensibles. A partir de cette définition, la question de l'existence du logiciel de contrôle est résolue en généralisant le théorème de virtualisation de Popek et Goldberg [69]. Cette version généralisée est appelée *théorème de contrôlabilité* dans le reste de ce mémoire.

L'application du théorème de contrôlabilité demande d'effectuer plusieurs analyses sur l'architecture du processeur, de ses cœurs, et sur le jeu d'instructions. Nous illustrons ces analyses dans un premier temps sur un modèle simplifié de cœur, puis sur un P5040, qui est un processeur quad-core de la gamme QorIQ, développée par Freescale. Cela aboutit à la description fonctionnelle du logiciel de contrôle que nous avons réalisé. Ce dernier est intégré dans un hyperviseur à faible empreinte que nous avons également développé. Il exerce son contrôle sur du logiciel utile exécuté au sein de machines virtuelles.

Efficacité du logiciel de contrôle

Nous évaluons enfin l'efficacité de notre logiciel de contrôle en nous concentrant sur le ratio entre le temps d'exécution mesuré d'un ensemble de benchmarks avec et sans logiciel de contrôle, pour différentes configurations de ce dernier. Nous montrons que pour certains benchmarks, il existe un domaine d'usage dans lequel la distribution des temps d'exécution est acceptable. Les benchmarks sélectionnés couvrent différents types d'algorithmes, ainsi qu'un cas d'étude de taille moyenne représentatif d'une application avionique.

1.4 Plan du mémoire

Ce mémoire est organisé en trois parties, suivies d'une conclusion générale :

Enjeux industriels et scientifiques. Dans le chapitre 2, nous détaillons un ensemble de problématiques industrielles liées aux systèmes avioniques. Ces problématiques influent notamment sur la sélection des processeurs, l'architecture des systèmes IMA et son évolution vers les processeurs multi-cœurs. Dans le chapitre 3, nous résumons les techniques d'évaluation de WCET existantes, puis nous détaillons les problèmes de prédictibilité des processeurs multi-cœurs, ainsi que les alternatives étudiées dans la communauté.

Démarche Scientifique. Après une synthèse de l'état de l'art, nous détaillons la problématique abordée dans cette thèse (chap. 4) et notre approche (chap. 5).

Contributions. Nous présentons les contributions en quatre chapitres. Le chapitre 6 présente le cadre dans lequel nous définissons notre logiciel de contrôle ainsi que le théorème de contrôlabilité. Il se termine par l'application de ce théorème sur un modèle de cœur simplifié. Le chapitre 7 montre l'application du théorème de contrôlabilité sur un processeur COTS. Le chapitre 8 développe une étude expérimentale sur l'efficacité du logiciel de contrôle.

Deuxième partie

Enjeux industriels et scientifiques

Chapitre 2

Contexte Avionique

Sommaire

2.1 Exigence de déterminisme 10
2.1.1 Principes de sûreté de fonctionnement 11
2.1.2 Définition de la notion de déterminisme 13
2.1.3 Contraintes temps réel et déterminisme temporel 13
2.1.4 Problématiques liées aux contraintes temps réel 14
2.2 Technologies embarquées dans les systèmes avioniques . . 16
2.2.1 Évolution des systèmes avioniques 16
2.2.2 Problématiques liées à l'usage de processeurs COTS . . 17
2.2.3 Problématiques liées aux technologies semi-conducteurs 18
2.3 Avionique Modulaire Intégrée (IMA) 20
2.3.1 Architecture d'un système IMA 21
2.3.2 Propriété de Partitionnement Robuste 23
2.3.3 Acteurs industriels d'un système IMA 24
2.3.4 Profil des applications déployées dans les systèmes IMA 25
2.4 Processeurs multi-cœurs pour l'avionique 25
2.4.1 Architectures multi-cœurs 26
2.4.2 Problématiques spécifiques aux processeurs multi-cœurs 29
2.5 Synthèse 31

On décompose un système avionique en unités d'exécutions autonomes, appelées *calculateurs*, qui communiquent via des liens réseaux. Sur un calculateur on distingue trois types de composants : le support d'exécution qui est en général un *processeur*, les briques logicielles appelées aussi *applications* et les briques matérielles (excepté le processeur).

Nous abordons la thématique des systèmes avioniques sous l'angle de trois problématiques générales :

Déterminisme. Pour être certifiables, les systèmes avioniques doivent être déterministes. Cette exigence porte sur des aspects d'intégrité de l'exécution du logiciel

avionique, mais également sur des aspects temporels. Nous détaillons ce point dans la section 2.1.

Efficacité. Pour être industriellement viables, les systèmes avioniques doivent être performants. L'efficacité d'un système se mesure sur des critères tels que sa puissance de calcul moyenne et en pire cas, ou encore sa consommation électrique. La problématique d'efficacité est habituellement abordée à travers l'usage de processeurs COTS (cf section 2.2.2), l'emploi de technologies semi-conducteurs récentes (cf section 2.2.3), mais également le développement de systèmes intégrés (cf problématique suivante). Enfin, la demande d'efficacité est la principale motivation de l'introduction de processeurs multi-cœurs, que nous détaillons dans le chapitre 3.

Intégration. De nombreuses applications avioniques sont déployées dans le cadre de systèmes intégrés. Cela signifie que les calculateurs affectés à ces systèmes hébergent plusieurs applications avioniques qui sont en général indépendantes et de niveaux de criticité différents. Cela permet notamment de limiter le nombre de calculateurs à embarquer dans un avion, ce qui contribue globalement à l'efficacité des systèmes avioniques. Les ressources matérielles d'un calculateur et le temps de calcul sont partagés entre les différents composants logiciels par un algorithme d'ordonnancement. Ce principe a été développé dans le concept d'Avionique Modulaire Intégrée (IMA), présenté dans la section 2.3.

Nous développons dans ce chapitre les trois problématiques exposées ci-dessus, en nous concentrant sur la manière dont elles peuvent rentrer en conflit. Par exemple, on présente souvent l'exigence de déterminisme et le besoin d'efficacité des systèmes avioniques difficilement conciliables. Il est donc nécessaire d'étudier ces problématiques conjointement. Nous terminons ce chapitre en abordant l'introduction de processeurs multi-cœurs dans les systèmes avioniques, et les problématiques qui en résultent.

2.1 Exigence de déterminisme

Un avion est un système critique dont la défaillance, définie par la perte de sa capacité à voler et atterrir, a en général des conséquences lourdes sur le plan humain. Afin d'éviter ce type de scenario, on demande à un avion et aux systèmes qui le composent de remplir des exigences de sûreté de fonctionnement. On dit informellement qu'ils doivent être déterministes. La notion de déterminisme à proprement parler n'a cependant été introduite que récemment [5] dans la règlementation portant sur les systèmes IMA, qui sont un sous-ensemble des systèmes avioniques.

L'objectif de cette section est de définir la notion de déterminisme dans le cadre d'un système soumis à des exigences de sûreté de fonctionnement, et d'expliciter la notion de déterminisme temporel dans le cadre de systèmes temps réel durs.

2.1.1 Principes de sûreté de fonctionnement

L'objectif premier d'un avion est d'être sûr, c'est-à-dire de pouvoir voler et atterrir sans mettre en péril des vies humaines. Pour cela, son développement fait l'objet d'un processus de certification. Au cours de ce processus, on associe à chaque fonction un niveau de criticité. Cette étape s'appelle *l'analyse fonctionnelle des risques*[3]. Le niveau de criticité d'un système correspondra au risque encouru sur la sûreté du vol en cas de condition de défaillance (cf définition 2.1) impliquant ce système (cf table 2.1). La règlementation en vigueur est constituée de l'ARP4761 [3] et l'ARP4754 [2], qui sont les standards qui ont uniformisé la règlementation américaine (CTR-25) et européenne (CS-25 [6], §25.1309). La notion de sûreté de fonctionnement se transpose donc de l'avion à ses systèmes, à un degré dépendant de leur niveau de criticité.

Définition 2.1 (Défaillance système)**.** *Une défaillance système (Failure conditions) correspond à une défaillance d'une fonction complète d'un avion. Celle-ci a des effets observables sur l'avion ou ses occupants.*

On considère un système avionique comme sûr quand on a démontré que sa probabilité de défaillance par heure de vol est inférieure à une valeur spécifiée par la règlementation, selon son niveau de criticité. Au niveau le plus critique s'ajoute l'obligation d'empécher qu'une défaillance système soit le résultat d'une défaillance unique. Ce sont des obligations de résultats. La manière d'obtenir ces résultats est cependant contrainte par la règlementation. Il y a donc une obligation de moyens, résumée dans le *plan de sûreté du système* (SSP), qui porte sur la mise en œuvre des actions suivantes :

Analyses qualitatives de risques. Ces analyses visent à déterminer les défaillances auxquelles le système peut être soumis. Ces défaillances sont souvent représentées sous forme d'arbres afin d'illustrer leurs liens de causalités. Il y a sept classes d'analyses de risques recensées dans la littérature [70]. Chaque classe d'analyse regroupe des méthodes qui lui sont propres. On considère que la couverture des défaillances est correcte quand une méthode appartenant à chaque classe a été appliquée.

Analyses quantitatives de risques. Ces analyses visent à déterminer, étant donné les cas de défaillance connus, la probabilité d'une défaillance système. Les techniques d'analyses les plus répandues sont les FMEA[4] et les FTA[5]. Un aperçu de ces techniques est donné par Ericson [35].

Analyse des Causes Communes de Défaillances. Cette phase d'analyse vise à explorer les dépendances entre systèmes afin de déterminer dans quels cas deux défaillances de systèmes fonctionnellement indépendant n'ont pas des probabilités d'occurrences indépendantes. Dans le cas de systèmes avioniques, une cause commune de défaillance classique est la perte d'une source d'alimentation électrique, qui cause l'arrêt de tous les systèmes alimentés par cette source.

3. Functional Hazards Assessement
4. Failure Mode and Effect Analysis
5. Fault Tree Analysis

Les standards en vigueur sont actuellement la DO-178B et DO-178C [1; 9] pour les composants logiciels, et la DO-254 [4] pour les composants matériels. L'EASA a également publié en 2011 un *Certification Memorandum* [8], qui se veut comme un complément de la DO-254. Ces standards définissent des niveaux DAL [6], allant du niveau A pour les systèmes les plus critiques au niveau E pour les systèmes les moins critiques (voire table 2.1). Ces standards spécifient pour chaque niveau DAL les actions à mener parmi celles définies précédemment sur les composants logiciels et/ou matériels.

TABLE 2.1 – Niveaux de criticité des scénarii de défaillance[a]

Gravité d'une condition de défaillance	Impact sur la sûreté du vol[b]	Probabilité de défaillance par heure de vol	Niveau DAL associé
Catastrophique (*Catastrophic*)	Perte de l'avion	$P < 10^{-9}$.[c]	DAL-A
Dangereux (*Hazardous*)	Diminution importante de la capacité de l'avion et/ou de l'équipage à faire face à des évènements hostiles. Situation de détresse dans l'équipage	$P < 10^{-7}$	DAL-B
Majeur (*Major*)	Diminution de la capacité de l'avion et/ou de l'équipage à faire face à des évènements hostiles. Augmentation de la charge de travail de l'équipage. Eventuelles blessures légères	$P < 10^{-5}$	DAL-C
Mineur (*Minor*)	Impact peu significatif sur la sûreté du vol. Peu de charge supplémentaire pour l'équipage	Non spécifié	DAL-D
Sans effet (*No Safety Effect*)	Aucun effet sur la sûreté du vol	Non spécifié	DAL-E

[a] La table 5 de l'ARP4761 [3] référence les phases d'analyses requises selon le niveau de criticité

[b] Les définitions complètes des conditions de défaillances sont disponibles dans les "Acceptable Means of Compliance" (AMC 25.1309, §7) [6]

[c] Dans le cas d'une défaillance système catastrophique, la condition ne doit pas résulter d'une défaillance simple au niveau d'un sous système.

6. Design Assurance Level

2.1.2 Définition de la notion de déterminisme

La notion de déterminisme n'a été introduite que récemment dans les standards avioniques. La DO-297 [7] [5] le présente comme *"La capacité à produire un résultat prédictible en fonction des opérations précédentes. Ce résultat doit être produit dans une plage de temps spécifiée qui peut se répéter"*. C'est donc une notion qui s'applique à la fois sur le système complet et sur ses composants. La définition proposée par la DO-297 est proche de la définition de *correction* d'un système telle que définie par Powell [71]. Elle se formule comme une obligation de résultats. On peut également proposer une définition duale du déterminisme exprimée comme une obligation de moyens, et qui s'appuie sur des standards avioniques plus anciens [2; 3].

Définition 2.2 (Source de non déterminisme)**.** *Une source de non déterminisme est un mode de défaillance non quantifié, c'est-à-dire dont la probabilité d'occurrence n'est pas connue.*

Définition 2.3 (Déterminisme)**.** *Un système est dit déterministe s'il ne comporte pas de sources de non déterminisme.*

Lorsqu'une cause de défaillance n'est pas bien maîtrisée, par exemple si sa probabilité d'occurence n'est pas connue, on parle de *source de non-déterminisme*. Un système déterministe (cf définition 2.3) se définit alors par opposition à un système non déterministe. On peut donc prouver qu'un système est déterministe en montrant qu'il n'est pas non-déterministe. La difficulté de cette preuve est l'identification et la couverture des sources de non déterminisme. Elle est effectuée lors du plan de sûreté du système dans les phases d'analyses qualitatives. On peut donc apporter la preuve de déterminisme en se conformant à une obligation de moyens.

2.1.3 Contraintes temps réel et déterminisme temporel

Les systèmes avioniques sont pour la plupart soumis à des exigences temps réel dures. Cela signifie qu'ils doivent remplir leurs fonctions en respectant des contraintes de ponctualité pour leurs traitements. Le manquement d'une échéance constitue une défaillance du système au même titre qu'un résultat erroné. D'après la définition 2.2, un système avionique sera considéré comme temporellement déterministe lorsque la probabilité d'occurrence d'un manquement d'échéance sera connue.

En général, un système avionique met en œuvre une chaine de traitements impliquant du logiciel, mais pas uniquement. On considère par exemple la gestion des entrées/sorties, ainsi que les transferts sur le réseau. Pour garantir le respect de l'échéance finale, il est courant de spécifier des échéances dites "locales" pour les différents éléments de la chaine de traitement. Pour un logiciel, respecter une échéance signifie avoir terminé son exécution avant celle-ci. L'exigence de déterminisme temporel est en général étendue au

7. On peut d'ailleurs noter que la DO-297, qui est le standard IMA, n'est pas applicable à tous les systèmes avioniques

respect des échéances locales auxquelles les éléments de la chaine de traitement sont soumis.

2.1.4 Problématiques liées aux contraintes temps réel

Nous considérons une configuration logicielle classique dans des systèmes avioniques : le cas où plusieurs tâches logicielles sont exécutées sur un même calculateur, sous le contrôle d'un ordonnanceur. Chaque tâche est périodique et doit respecter des échéances de terminaison au plus tard. Plusieurs problématiques découlent des contraintes temps réel :

Garantie du respect des échéances. Les multiples éléments de la chaine de traitements d'un système ont à répondre à des échéances. Pour un logiciel, le manquement d'une échéance peut résulter :

- D'une erreur d'ordonnancement, qui a privé le logiciel du temps dont il aurait eu besoin pour respecter son échéance. Ce problème est classiquement abordé dans la théorie de l'ordonnancement. Ce problème dispose de solutions matures dans le cadre mono-cœurs, et dans une moindre mesure dans le cadre multi-cœurs. Nous en détaillons certains aspects dans la section 2.3.
- D'une mauvaise évaluation du temps d'exécution du logiciel, qui a dépassé le budget qui lui était alloué. Ce problème est abordé par la notion de calcul du "pire temps d'exécution" (WCET). Nous développons ce problème, qui est central dans cette thèse, dans le chapitre 3.
- D'une défaillance du support d'exécution matériel qui n'a pas offert au logiciel les ressources nécessaires pour mener à bien son exécution. Ce problème, qui relève du déterminisme d'une manière plus générale, est abordé sous deux aspects. D'une part on peut considérer des défaillances intrinsèques du support d'exécution et de l'environnement matériel au sens large, par exemple en se référant à l'étude sur les processeurs COTS [8] de Faubladier et Rambaud [37], ou encore à celle de Bieth et Brindejonc [23] pour l'EASA [9]. D'autre part on peut s'intéresser aux erreurs de partage des ressources entre plusieurs logiciels. Elles relèvent de la notion de *Partitionnement Robuste*, qui a notamment été développée par Rushby [79], et que nous détaillons dans la section 2.3.2.

Dimensionnement du support d'exécution. Il est nécessaire de dimensionner au mieux le support d'exécution afin de satisfaire les contraintes temps réel sans consommer trop de ressources. Dans le cas d'un support d'exécution embarquant un ordonnanceur et un ensemble de tâches, un support d'exécution peut être surdimmensionné pour deux raisons :

- Le temps alloué à chaque tâche sur une période est trop important. C'est par exemple le cas s'il a été évalué de manière trop pessimiste. Cela peut arriver lorsque l'évaluation du pire temps d'exécution (WCET) d'une tâche

8. Components Off-The-Shelf
9. European Aviation Safety Agency, l'organisme de certification en Europe

nécessite des approximations trop grossières pour rester correcte, par exemple sur le comportement du processeur. Une approche probabiliste a été proposée par Bernat [21] et par Cucu-Grosjean [30]. Cette approche vise à évaluer des WCET probabilistes plus représentatifs de la distribution des temps d'exécution. Le système reste déterministe au sens de la définition 2.3 car la probabilité de défaillance, en l'occurrence le dépassement d'une échéance, est connue.

- La période et/ou les échéances des tâches sont trop rapprochées pour le besoin réel du système. Andrianiaina [13; 14] a illustré ce phénomène sur des systèmes de contrôle-commande représentatifs de systèmes de commandes de vol, qui sont des exemples de systèmes habituellement soumis à des exigence temps réel dures. Il y montre que de tels systèmes restent stables malgré plusieurs manquements d'échéances successifs, en l'occurrence quatre. Cela suggère que la contrainte temps réel dure n'est pas pertinente, ou bien que le système est inutilement surdimmensionné. Il est toutefois délicat de conclure sur cette affirmation, car le système peut être surdimmensionné intentionellement en supposant que les manquements successifs de plusieurs échéances sont peu corrélés, ce qui rend cette combinaison très improbable.

Tolérance aux fautes. Un logiciel avionique est en faute s'il manque une deadline. S'il est exécuté sous le contrôle d'un ordonnanceur, cette faute peut perturber l'ordonnancement. L'ordonnanceur peut redémarrer la tâche fautive, mais également essayer de lui allouer un temps de calcul complémentaire s'il en dispose, et si cela ne perturbe pas les autres tâches. C'est la notion de *robustesse* de l'ordonnancement, qui a été étudiée par George [46] dans un cadre mono-cœur, et par Fauberteau [36] dans un cadre multi-cœurs.

L'exigence de déterminisme temporel soulève donc deux grandes problématiques : l'évaluation du besoin en ressources et en temps de calcul par le logiciel, et le dimensionnement des supports d'exécution pour offrir au logiciel les ressources nécessaires, y compris en présence de défaillances. Dans cette thèse, nous nous intéressons principalement à la première problématique, à savoir l'évaluation du pire temps d'exécution d'une tâche séquentielle.

Le déterminisme est la principale problématique à laquelle sont soumis les systèmes avioniques. Toutefois ces derniers doivent également répondre à une problématique d'efficacité. C'est pourquoi l'architecture des systèmes avioniques a globalement suivi l'évolution des technologies existantes dans le domaine des processeurs. C'est l'objet de la section suivante.

2.2 Technologies embarquées dans les systèmes avioniques

2.2.1 Évolution des systèmes avioniques

Les systèmes avioniques doivent obéir à des exigences de déterminisme, mais ils doivent également répondre à une problématique d'efficacité qui résulte d'une demande industrielle croissante en terme de performances, tout en conservant un volume, un poids, une consommation électrique et une dissipation thermique raisonnables [10]. C'est pourquoi les systèmes avioniques évoluent en même temps que les technologies semi-conducteurs, en particulier celles que l'on rencontre dans les processeurs, les mémoires et les composants programmables de type FPGA [11]. Cette évolution s'est faite de manière continue en mettant à jour les technoogies courantes, à l'exception de quelques ruptures technologiques.

Les premiers systèmes avioniques ont été introduits sur Concorde dans les années 1960 avec notamment des commandes de vol électriques. Ces systèmes relevaient de l'électronique analogique et étaient dits *fédérés*, c'est-à-dire que chaque fonction était remplie par un système dédié. Le logiciel est apparu dans les systèmes avioniques avec la génération des Airbus A300, Boeing 737 et 747 dans les années 1970, puis Airbus A310 dans les années 1980, avec une vingtaine d'équipements avioniques embarqués [52]. Les processeurs utilisés étaient dédiés à l'exécution d'une fonction unique, adaptés des technologies venant des équipements militaires. Il y a eu une nouvelle rupture avec la génération Airbus A320 puis A330, A340 et Boeing 777 dans les années 1990. Les processeurs embarqués sont devenus des processeurs COTS [12], c'est-à-dire achetés dans le commerce et prévus pour un marché plus large que l'avionique. On dénombrait vingt à soixante fonctions avioniques par avion. Enfin, pour réduire le nombre de calculateurs, une troisième rupture a été introduite dans l'avionique avec la génération Airbus A380, Boeing 787, ATR 72, Sukhoï SSJ-100 et Airbus A350 dans les années 2000. Il s'agit en fait de remplacer les architectures dites *fédérées* où chaque équipement dispose de l'ensemble de ses senseurs/actuateurs pour réaliser une fonction dédiée par des architectures dites *intégrées* ou *distribuées*. Ces architectures sont conçues pour permettre aux équipements de pouvoir embarquer plusieurs fonctions avioniques, de 3 à 4 pour l'A380 à une dizaine pour l'A350, en partageant également les entrées/sorties nécessaires à leur bonne exécution. Cette rupture a permis d'augmenter le nombre de fonctions embarquées tout en stabilisant, voire réduisant le nombre d'équipements [52].

À l'exception de quelques systèmes comme les commandes de vol, la tendance générale des systèmes avioniques est de migrer vers des architectures intégrées et d'augmenter le nombre d'applications par calculateurs. Cela vise à augmenter l'efficacité globale de l'avionique, à l'échelle de l'avion.

10. On parle de SWaP : Size, Weight and Power
11. Field Programable Gate Array
12. Components Off The Shelf

2.2.2 Problématiques liées à l'usage de processeurs COTS

L'utilisation de processeurs COTS s'est généralisée dans l'avionique à partir des années 1980. Les principales motivations étaient d'une part l'abandon des normes MIL par le domaine militaire [62], qui imposaient le recours à des composants spécifiques qui ne pouvaient être développés qu'en interne, et l'explosion des marchés de masse entraînant une réduction des coûts et une augmentation des performances des processeurs. Il devenait également impossible pour les équipementiers avioniques de maintenir en interne une capacité de production qui pouvait suivre l'évolution des technologies semi-conducteurs [20; 51]. L'introduction d'un processeur COTS dans un équipement avionique fait donc naître une intéraction entre un équipementier qui veut proposer un équipement avionique certifié, et un fabricant de processeurs. Du côté de l'équipementier, utiliser un processeur COTS soulève deux problématiques : celle de la sélection du fabricant et celle de la sélection du processeur.

Le choix du fabricant de processeurs s'effectue en fonction de critères informels [47; 39] portant entre autre sur son espérance de vie, la qualité et la répétabilité de son processus de fabrication, et sa volonté de collaborer avec l'équipementier lors du processus de certification. Cette volonté se manifeste via diverses actions telles que des annonces de presse [84], ou bien un engagement de production sur une longue durée [85], afin de limiter des problèmes d'obsolescence. Par ailleurs, le choix du fabricant intervient souvent avant celui du processeur. Historiquement, les processeurs COTS utilisés dans l'avionique sont issus du marché des infrastructures réseaux –les serveurs et les switchs par exemple. Les fabricants historiques sont IBM et Motorola –actuellement Freescale–, principalement autour de leurs gammes PowerPC.

La sélection du processeur, qui intervient en général après celle du fabricant, est un processus qui vise à identifier les processeurs qui répondent aux problématiques suivantes :

Effort de certification. Aucun fabricant ne vend de processeur COTS directement prêts à être certifié, et aucun fabricant n'accepte la responsabilité liée à la garantie du bon fonctionnement d'un processeur dans un usage avionique. Les fournisseurs d'équipements doivent donc conduire eux-mêmes le processus de certification du processeur, éventuellement avec le soutien du fabricant. Avoir un tel soutien peut être difficile car la part de marché représentée par l'avionique est très faible, moins de 2% du marché des semi-conducteurs. La règlementation en vigueur a été adaptée par l'EASA dans un "Certification Memorandum" [8], qui donne la position de l'EASA sur le processus de certification à adopter pour les processeurs COTS. Ce document relâche certaines exigences de la DO 254 [4] qui n'étaient pas compatibles avec des processeurs COTS.

Disponibilité des informations. Les fabricants de processeur sont en général peu enclins à communiquer l'intégralité des informations concernant l'architecture, la fabrication ou encore le comportement de leur processeur. On parle informellement de composants "boîtes noires", ou "boîtes grises". L'utilisation de composants boîtes noires limite la possibilité d'apporter une preuve constructive du déter-

minisme. La règlementation [8] recommande d'utiliser localement des méthodes empiriques, qualifiées d'expérience en service (In Service Experience).

Les processeurs COTS embarquent souvent des accélérateurs matériels faisant intervenir du micro-code propriétaire. Ce micro-code doit cependant faire l'objet d'une certification à part entière. Il est difficile pour l'équipementier de réaliser cette certification sans disposer du code source. À notre connaissance, ce problème est évité en inhibant les fonctions des composants concernés.

Maîtrise de la complexité interne. Les processeurs COTS visent principalement à fournir de bonnes performances, et pour cela ils embarquent de nombreux mécanismes d'optimisation. Pour un fabricant, l'exigence de déterminisme est secondaire, et vient souvent de besoins d'intégrité des données, notamment dans le cas des infrastructures pour les réseaux. La notion de déterminisme temporel n'est en général pas étudiée par le fabricant. La complexité des processeurs COTS amène de nouveaux modes de défaillances [37; 23], y compris temporelles. Ces défaillances sont en général liées à des phénomènes qui échappent au contrôle logiciel, par exemple la reconfiguration silencieuse de certaines fonctions du processeur.

Du point de vue de l'équipementier, cela signifie qu'il est difficile d'apporter la preuve du déterminisme du processeur, car il faut prendre en compte toutes ces optimisations en caractérisant le comportement du processeur. C'est pourquoi il est courant de les désactiver, quitte à dégrader les performances de ce dernier.

Les problématiques liées à l'usage de processeurs COTS sont principalement liées au manque de maîtrise et d'informations du côté des fournisseurs d'équipements avioniques. La majorité des approches proposées dans la littérature et dans la règlementation tendent vers une caractérisation du comportement du processeur par des méthodes empiriques. Dans notre contexte, nous retenons particulièrement les problématiques de la maîtrise de la complexité de l'architecture du processeur ainsi que du manque d'information communiquées par le fabricant. La combinaison de ces problématiques complique la tâche d'assurance du déterminisme pour l'équipementier.

2.2.3 Problématiques liées aux technologies semi-conducteurs

Les processeurs utilisés dans les équipements avioniques sont basés sur des technologies CMOS [13] considérées comme matures. La tendance générale des technologies CMOS est d'aller vers une réduction de la finesse de gravure des transistors. En dessous d'une finesse de 90 nm, on parle de technologie DSM [14]. Cela permet d'améliorer la densité de transistors présents sur une puce, et donc le nombre de transistors à surface constante, et in fine la puissance de calcul. Les technologies utilisées dans l'avionique suivent cette tendance pour des raisons de performances, mais aussi d'anticipation d'obsolescence. Les équipements actuellement en service embarquent des technologies avec des finesses de

13. Complementary Metal-Oxyd Semiconductor
14. Deep Sub-Micronic

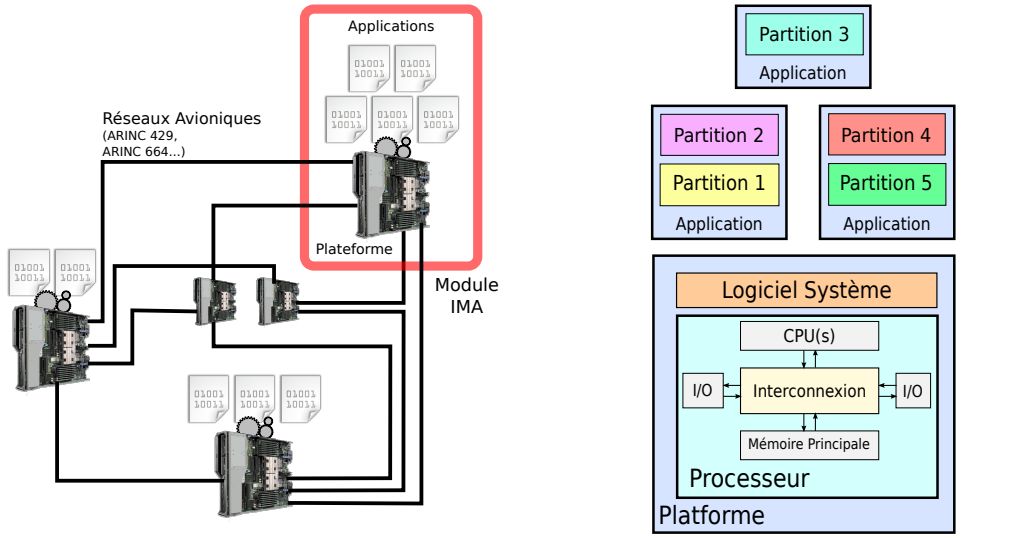
gravure situées entre 180 nm et 90 nm. Ces technologies datent du début des années 2000. La réduction des finesses de gravure soulève toutefois plusieurs problématiques :

Le vieillissement des processeurs. Tout processeur vieillit, même s'il n'est pas utilisé. Au niveau DSM, les principaux phénomènes qui entraînent ce vieillissement sont appelés électro-migration, HCI (Hot Carrier Injection) et NBTI (Negative Bias Temperature Instability) [12; 76]. Ils altèrent la tension de seuil à laquelle un transistor commute. A grande échelle et à long terme, cela peut entraîner des retards dans les temps de réponse, des surconsommations voire des défaillances sur certaines fonctions du processeur. Avec la réduction des finesses de gravure, ce type de phénomène se manifeste plus tôt dans le cycle de vie d'un processeur [91]. Par conséquent, les durées de vie garanties par les fabricants ne peuvent plus être considérées comme constantes, mais dépendent des conditions d'utilisation dudit processeur. Un composant vieillissant peut manifester des comportements défaillants du point de vue temporel, mais aussi vis-à-vis de l'intégrité des données et des signaux internes. Par exemple une mémoire peut renvoyer des données erronées, ou encore voir son temps de réponse augmenter. La principale difficulté liée au vieillissement consiste à évaluer son impact, car cela se fait par des méthodes empiriques sur des plages de temps longues, même en utilisant des techniques qui accélèrent le vieillissement.

L'anticipation de l'obsolescence. Une technologie semi-conducteurs a une durée de vie comprise entre le début de sa commercialisation dans des composants COTS et son obsolescence, i.e. l'arrêt de sa fabrication. Cette durée de vie est actuellement inférieure à celle d'un avion [76]. De plus, il n'est pas possible de maintenir des stocks à cause d'effets de vieillissement des composants, qui se produisent même si ceux-ci ne sont pas utilisés. C'est pourquoi un équipement avionique doit évoluer en se mettant à jour des technologies courantes.

La maîtrise des courants de fuite. La consommation électrique d'un transistor est divisée en deux : une partie dynamique nécessaire à sa commutation, et une partie fixe dite "courant de fuite", qui est présente même si le transistor ne commute pas. Dans les anciennes technologies semi-conducteurs, le courant de fuite était négligeable. Ce n'est plus le cas avec les technologies inférieures à 90 nm [25]. Il est alors nécessaire de désactiver les éléments du processeur qui ne sont pas utilisés, afin de limiter sa consommation globale et la dissipation thermique.

La sensibilité aux radiations. Ce type de phénomènes apparait lorsqu'un processeur est soumis à un rayonnement d'ions lourds, de protons ou de neutrons, majoritairement en provenance de l'espace. Lorsqu'une particule frappe un transistor, il peut se produire des commutations de transistors qui peuvent amener à des changements d'états de bits aléatoires. On parle de Single Event Upset (SEU), (respectivement de Multiple Bits Upset – MBU), lorsque le phénomène touche un (respectivement plusieurs) bit(s) contigu(s) [92; 64]. Avec la diminution des finesses de gravure, les processeurs sont de plus en plus sensibles à ces phénomènes [27]. Les mémoires sont les zones les plus exposées. La plupart des processeurs



(a) Représentation d'un système IMA (b) Représentation d'un module IMA

FIGURE 2.1 – Vue d'ensemble d'un système et d'un module IMA

disposent de codes correcteurs d'erreur capables de corriger l'effet d'un SEU et de détecter un MBU sur deux bits.

Les problématiques liées aux technologies semi-conducteurs ont un impact fort sur la sélection et la certification de processeurs pour l'avionique. Dans notre contexte, elles sont secondaires, même si elles peuvent influer sur la résolution des problèmes d'évaluation du temps d'exécution d'un logiciel. Par exemple, il est pertinent d'anticiper l'impact du vieillissement des mémoires sur leur temps de réponse dans une analyse de pire temps d'exécution.

Nous avons abordé dans ces deux sections des problématiques portant sur le matériel : la nature COTS des processeurs utilisés, et les problématiques de fiabilité des technologies semi-conducteurs récentes. La section suivante s'intéresse au concept d'Avionique Modulaire Intégrée (IMA), et aborde des problématiques orientées vers le logiciel avionique, au niveau applicatif et au niveau du système d'exploitation.

2.3 Avionique Modulaire Intégrée (IMA)

Le concept d'Avionique Modulaire Intégrée (Integrated Modular Avionics) a été conçu pour permettre d'embarquer sur un même calculateur plusieurs applications indépendantes, de niveau de criticité potentiellement différents, tout en conservant la capacité à les certifier séparément. L'IMA est censé également permettre à une application d'évoluer sans nécessiter la re-certification du système complet. C'est ce qui est appelé le principe de *certification incrémentale*. Ce principe permet de répondre à la problématique d'inté-

gration, présentée au début de ce chapitre. Le concept d'IMA aborde les problématiques suivantes :

Allocation de ressources. Plusieurs applications avioniques sont déployées sur un même calculateur. Elles partagent l'ensemble des ressources matérielles (mémoire, entrées/sorties) et le temps de calcul sur le cœur. L'IMA définit la manière dont ces ressources doivent être allouées. Le temps de calcul est partagé par un algorithme d'ordonnancement hiérarchique décrit dans la section 2.3.1. Les ressources matérielles sont allouées de manière statique à chaque application.

Réutilisation de logiciel. L'un des objectifs de l'IMA est de faciliter la réutilisation d'applications avioniques existantes en minimisant leur effort de portage. C'est pourquoi les interactions qu'elles ont avec les ressources matérielles de la plateforme et les services du système d'exploitation sont standardisées dans la recommandation ARINC 653 [7]. Ce point est détaillé dans la section 2.3.1.

Isolation entre applications. Cela permet de garantir que les applications les plus critiques ne seront pas impactées par les applications moins critiques. Cette problématique d'isolation est adressée par la propriété de Partitionnement Robuste, détaillée en section 2.3.2.

Collaboration industrielle. Un système IMA est le fruit d'une collaboration entre plusieurs acteurs industriels, ayant chacun leurs propres intérêts, et pouvant parfois être concurrents. La certification du système demande une certaine coopération entre ces acteurs, que le concept d'IMA a normalisé en imposant la communication de certains documents. Nous détaillons cette problématique dans la section 2.3.3.

L'IMA est caractérisé par l'adaptation dans un contexte avionique de concepts éprouvés dans les systèmes non critiques, par exemple la notion de plateforme d'exécution générique, d'applications inconnues ou encore d'isolation. La plupart de ces concepts sont mis en œuvre par une couche logicielle dite "logiciel système", qui contrôle l'exécution des applications. En général c'est un système d'exploitation temps réel. Les sections suivantes développent les problématiques présentées ci-dessus et la manière dont elles sont résolues dans le cadre de l'IMA.

2.3.1 Architecture d'un système IMA

Comme illustré sur la figure 2.1, un système IMA est composé d'un ou plusieurs modules reliés par un ou plusieurs réseaux avioniques. Chaque module comporte une plateforme et un ensemble d'applications. La plateforme est composée du support d'exécution (processeur) et du système d'exploitation temps réel (RTOS). Les applications constituent la charge utile du module. Une application est exécutée à l'intérieur d'une ou plusieurs unités d'exécution isolées, ce sont les partitions. Une partition –qui peut se concevoir comme un processus lourd au sens Unix– embarquera un ou plusieurs processus –qui sont l'analogue des processus léger dans Unix.

Du point de vue logiciel applicatif, la partition constitue l'unité élémentaire de logiciel à certifier. La DO 297 [5], qui est la règlementation en vigueur dans les systèmes IMA, impose l'utilisation d'un modèle de partition standardisé. La recommandation ARINC 653 [7] s'est imposée de fait [15] comme un standard [74]. Elle y décrit une partition comme la combinaison :

- d'un ensemble de tâches concurrentes qui sont ordonnancées à l'intérieur des fenêtres temporelles définies précédemment. Chaque tâche a une échéance.
- de plages mémoires réservées à la partition qu'elle peut utiliser sans restrictions. Les plages mémoires de chaque partitions sont disjointes, et tout accès d'une partition hors d'une de ses plages est considéré comme une défaillance.
- d'une ou plusieurs fenêtres temporelles périodiques à l'intérieur desquelles l'exécution du logiciel inclus dans la partition est autorisée.
- d'un ensemble de ports de communication ouverts à la partition afin de communiquer, via le système d'exploitation, avec les ressources matérielles du processeur et les autres partitions.

Cette définition décrit une enveloppe générique à l'intérieur de laquelle une partition peut s'exécuter de manière autonome en utilisant les ressources qui lui ont été allouées. Le système d'exploitation assure que chaque partition dispose d'une fenêtre temporelle d'exécution. Il contrôle également l'exécution du logiciel à l'intérieur des partitions à travers un algorithme d'ordonnancement intra-partition qui est préemptif et qui se base sur les priorités des tâches. Ces priorités peuvent être altérées, par exemple en cas de prise de verrou. RMS [16] et EDF [17] sont les algorithmes d'ordonnancement les plus couramment employés. Ces familles d'algorithmes d'ordonnancement ont deux propriétés très utiles dans les systèmes temps réels [31] :

Faisabilité. Il existe un test d'ordonnancement hors ligne applicable sur l'ensemble des tâches. Ce test permet de garantir que chaque tâche respectera ses échéances.

Viabilité. Si une tâche s'exécute plus vite que son budget alloué, et que le reliquat est récupéré par l'ordonnanceur, les autres tâches respectent quand même leurs échéances [18].

Ces deux propriétés permettent d'effectuer le test d'ordonnancement en supposant que chaque tâche consomme son WCET, en ayant la garantie qu'au final chaque tâche respectera son échéance quelle que soit son exécution.

A travers ce modèle de partition, l'IMA répond donc à deux problématiques : celle de la réutilisation de logiciel existant en en facilitant la portabilité et celle du partage des ressources par une allocation statique des ressources mémoire et des entrées/sorties, des ports de communication et des fenêtres d'exécution de chaque partition. Cette allocation statique vise également à assurer la propriété de Partitionnement Robuste, qui est détaillée dans la section suivante.

15. La conformité à ARINC 653 n'est jamais explicitement requise dans la DO 297, elle est seulement mentionnée comme exemple
16. Rate Monotonic Scheduling
17. Earliest Deadline First

2.3.2 Propriété de Partitionnement Robuste

La possibilité de certifier modulairement plusieurs applications repose sur une propriété d'isolation de fautes : le *Partitionnement Robuste*, dont la définition la plus précise, dite *Gold Standard*, a été donnée par Rushby [79] :

> *"Un système qui vérifie la propriété de Partitionnement Robuste offre le même niveau d'isolation de fautes qu'un système équivalent où chaque partition est exécutée sur un processeur dédié et dispose de canaux de communications dédiés avec les autres partitions"*

Cette définition présente le Partitionnement Robuste comme une propriété de non régression sur la sûreté de fonctionnement par rapport aux systèmes fédérés, et s'applique quel que soit le modèle de fautes. Cette définition est une obligation de résultat. Dans les systèmes actuels, la propriété de Partitionnement Robuste est assurée par la mise en place de mécanismes d'isolation de l'activité relative aux différentes partitions. C'est une obligation de moyens. ARINC 653 ([7] §2.3.1) introduit notamment la notion de partitionnement spatial et temporel : les ressources matérielles et temporelles statiquement allouées à chaque partition sont disjointes. D'une manière plus générale, les recommandations de ARINC 653 remplissent les conditions d'une propriété plus forte, dite *Alternative Gold Standard* [99] :

> *"Le comportement et les performances du logiciel d'une partition ne doivent pas être affectées par le logiciel d'une autre partition"*

La contrainte de Partitionnement Robuste restreint la manière dont on peut montrer qu'une partition est temporellement déterministe, i.e. que toutes ses tâches respectent leurs échéances. Elle empêche en effet de faire des hypothèses sur le fonctionnement des partitions concurrentes, même si elles sont connues, à l'exception des restrictions mises en place par le système d'exploitation. On peut donc par exemple évaluer le WCET d'une tâche sans envisager le cas où son empreinte mémoire est corrompue par une autre partition. Cela correspondrait alors à une défaillance du système d'exploitation et donc de la plateforme.

La principale problématique relative aux systèmes IMA est celle de l'isolation de défaillances entre partitions, condition nécessaire à la modularité du processus de certification. Cette isolation est assurée par le fournisseur de plateforme et doit être valable pour toute partition conforme à sa spécification, par exemple celle décrite par ARINC 653. C'est également la base du processus de certification incrémentale, qui permet de faire évoluer une application sans re-certifier tout le module.

Un module IMA se caractérise également par le fait que plusieurs acteurs industriels interviennent au cours de son cycle de vie, sans que l'un d'entre eux ait accès à l'intégralité des documents et code source du logiciel déployé. L'IMA décrit donc la manière dont ces acteurs doivent interagir, entre autres pour préserver le caractère déterministe du système, notamment d'un point de vue temporel. C'est l'objet de la section suivante.

2.3.3 Acteurs industriels d'un système IMA

Le concept d'Avionique Modulaire Intégrée décrit un processus industriel dans lequel différents acteurs interviennent, et la manière dont ils doivent coopérer pour satisfaire les exigences de certification, en premier lieu l'exigence de déterminisme. Les principaux acteurs sont les suivants :

L'architecte système. Il est en charge de choisir les applications qui seront hébergées sur chaque module, et leur affecte un niveau de criticité. Il est également en charge de la configuration des modules et de leur déploiement. Pour cela il doit dimensionner l'ordonnancement afin d'allouer un budget de temps suffisamment important à chaque application pour que celle-ci puisse respecter ses échéances. La configuration d'un module doit respecter le domaine d'usage de sa plateforme. Le rôle d'architecte système, appelé également *intégrateur*, du système, est en général occupé par l'avionneur.

Le fournisseur de plateforme. Il développe une plateforme d'exécution générique sans savoir a priori quelles applications seront déployées et exécutées. Il est en charge entre autres d'assurer le partitionnement robuste et de permettre à un tiers d'évaluer le temps d'exécution d'une application embarquée sur la plateforme. Il doit pour cela fournir suffisamment d'informations sur le comportement de la plateforme, en particulier sur le logiciel système, qui est en général propriétaire. Il doit également documenter un *domaine d'usage* de la plateforme, qui résume les restrictions d'utilisation au delà desquelles le comportement de la plateforme n'est plus assuré. Les documents contenants le domaine d'usage et les informations temporelles sur le comportement de la plateforme constituent son interface observable pour les autres acteurs. Ils engagent la responsabilité du fournisseur de plateforme.

Le fournisseur d'application. Il est en charge de développer une application à un niveau de criticité donné. Il ne connait pas les applications qui seront hébergées sur la même plateforme. Il a également la charge d'évaluer les ressources nécessaires pour que son application puisse s'exécuter correctement et respecter ses échéances. Cette évaluation dépend notamment d'informations communiquées par le fournisseur de plateforme.

On a donc plusieurs acteurs industriels qui interviennent dans la réalisation et l'exploitation d'un système IMA. Chaque acteur est confronté à une problématique de visibilité sur le logiciel déployé. Par exemple, le fournisseur de plateforme ne sait pas quelles applications vont être déployées sur sa plateforme, et doit donc proposer une plateforme générique à la fois déterministe et performante pour une gamme large d'applications. De même, le fournisseur d'applications et l'intégrateur du système n'ont a priori pas accès au code source du système d'exploitation, alors qu'il a un rôle central dans l'allocation et l'utilisation des ressources du processeur. Le fournisseur de plateformes doit donc être en mesure de documenter une méthode d'évaluation du pire temps d'exécution pour une application exécutée sur sa plateforme, dans une configuration donnée.

2.3.4 Profil des applications déployées dans les systèmes IMA

Un fournisseur de plateformes n'a pas de visibilité sur les applications déployées dans un système IMA. Néanmoins, certaines informations communiquées par des avionneurs comme Airbus [52; 22] montrent que les applications hébergées dans un système IMA couvrent des domaines tels que :

- La planification du vol : planification de trajectoire, Flight Management System.
- Les équipements du cockpit : écrans de contrôle, alarmes, communications, pesée et équilibrage.
- le contrôle des installations en cabine : alimentation et surchauffe des équipements, détection de fuites, contrôle de la pression et de la ventilation
- Le contrôle de l'alimentation des circuits électriques et hydrauliques
- Les "utilities" : mesure du niveau de carburant, transferts de carburant, contrôle du freinage, contrôle du train d'atterrissage.

Ces applications se classent selon trois profils : celles qui ont un fort besoin de puissance de calcul, celles qui ont un fort besoin de bande passante en entrées/sorties, et celles qui n'ont ni l'un ni l'autre. Le premier cas concerne principalement les applications de planification de vol. Le deuxième cas concerne entre autres la gestion des écrans du cockpit. La majorité des applications relève de la troisième catégorie.

Le profil "type" d'une application IMA est donc celui d'une petite application, de l'ordre de quelques milliers à quelques dizaines de milliers de lignes de code pour une empreinte mémoire de l'ordre de quelques dizaines à quelques centaines de kilo-octets. Une application typique d'un système IMA est séquentielle, et donc composée d'une partition et d'un processus. Enfin, l'exécution d'une application typique peut être découpée en trois phases : une phase de lecture des données d'entrées via les ports ARINC 653 gérés par le système d'exploitation, une phase d'exécution autonome, et une phase d'écriture sur les sorties via le système d'exploitation. Durant toutes les phases de son exécution, l'application peut accéder à la mémoire principale. Une application ayant ce profil ne requiert pas l'intégralité des ressources d'un processeur COTS récent pour respecter ses échéances.

2.4 Processeurs multi-cœurs pour l'avionique

Les processeur multi-cœurs sont apparus sur le marché au début des années 2000 quand les fabricants ont été stoppés dans leur course à l'augmentation de la fréquence de fonctionnement, à cause de problèmes de consommation électrique, d'échauffement ou encore d'interférences électro-magnétiques qui devenaient difficiles à maîtriser. Les fabricants se sont alors tourné vers l'intégration de plusieurs cœurs sur une même puce, ce qui est devenu possible grâce à la réduction de la finesse de gravure. La course à la performance est donc passée de la maîtrise de la fréquence de fonctionnement à la maîtrise de l'exécution en parallèle de plusieurs tâches. Cela permettait d'augmenter globalement la puissance de calcul et ainsi de suivre la loi de Moore. L'introduction de processeurs

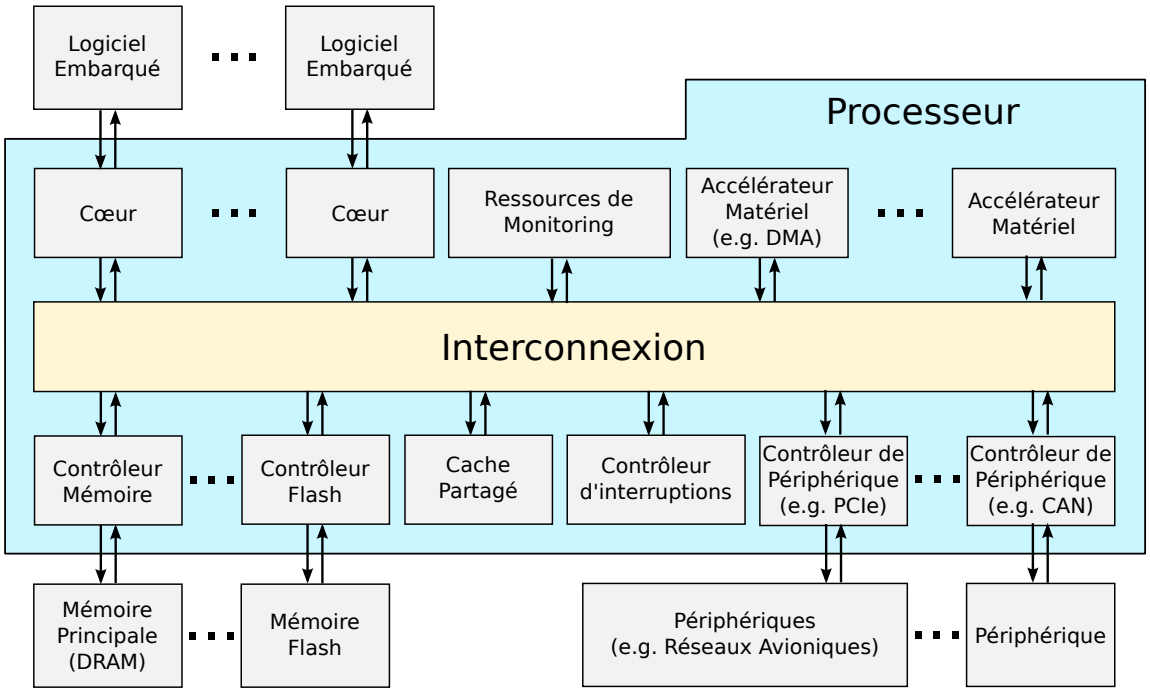


FIGURE 2.2 – Modèle d'un calculateur comportant un processeur multi-cœurs

multi-cœurs dans les systèmes avioniques est d'actualité depuis la fin des années 2000. Elle vise à apporter une solution pérenne à deux problématiques :

- L'efficacité des systèmes (cf section 2.1), à conditions que ceux-ci puissent tirer parti de l'exécution parallèle de plusieurs tâches logicielles sur les différents cœurs.
- L'anticipation de l'obsolescence (cf section 2.2.3). Les fabricants de processeurs s'orientent tous vers la technologie multi-cœurs. Une bonne maîtrise de la technologie permettra d'assurer à long terme la disponibilité de composants COTS pour des équipements avioniques.

Nous détaillons dans la section 2.4.1 quelques éléments caractéristiques de l'architecture d'un concept de calculateur avionique embarquant un processeur multi-cœurs. Dans la section 2.4.2, nous décrivons plusieurs problématiques associées à l'usage de processeurs multi-cœurs dans l'avionique.

2.4.1 Architectures multi-cœurs

Comme illustré sur la figure 2.2, on représente habituellement un processeur de manière modulaire comme un ensemble d'unités matérielles, dont l'activité électronique permet l'exécution du logiciel embarqué, et remplit les services offerts par le processeur. Ces unités matérielles sont en général des IP [18], qui sont développées indépendamment pour

18. Intellectual Property

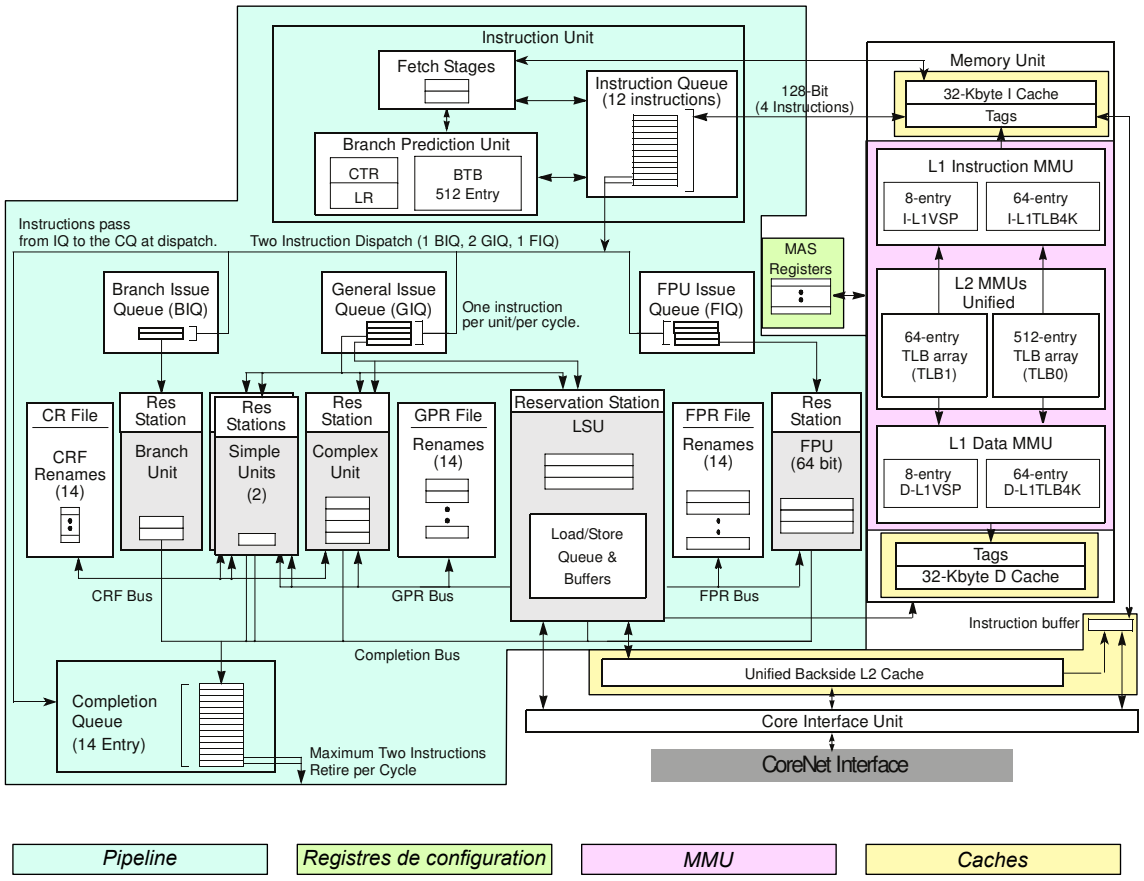


FIGURE 2.3 – Description d'un cœur PowerPC de type e5500 (issue du manuel de référence [43])

être utilisées dans différents types de processeurs. Leurs activités, au niveau électronique, s'effectuent de manière concurrente.

Sur un processeur multi-cœurs, on peut rencontrer les types d'unités suivantes, qui sont illustrées sur la figure 2.2 :

les cœurs. Ils contiennent la logique nécessaire pour exécuter un programme et déployer un système d'exploitation. Cette logique comprend les unités suivantes (cf figure 2.3) :

- ***Pipeline*** Le pipeline implémente dans des unités dédiées différents traitements effectués au cours de l'exécution de l'instruction. On rencontrera par exemple une unité pour charger les instructions, plusieurs unités arithmétiques et logiques capables d'effectuer des opérations entières ou flottantes, une unité capable de lire et écrire des données vers la mémoire principale.
- ***MMU.*** Les cœurs utilisés dans l'avionique disposent d'une MMU [19]. Il s'agit d'une unité matérielle chargée de la traduction d'adresse entre l'espace *effectif*, vu

19. Memory Management Unit

par le logiciel, et l'espace *physique*, vu par le processeur. Elle est également en charge de vérifier les droits d'accès.

Caches. Les caches agissent comme des mémoires de petite taille à accès rapide. Ils disposent d'une politique de remplacement, et permettent d'améliorer significativement les performances du logiciel embarqué.

Registres d'états et de configuration. Un cœur embarque un certain nombre de registres de configuration, dont la connaissance fine est nécessaire dans un usage avionique.

Les unités de communication. On trouve en général dans un processeur un composant central appelé *interconnexion*, *bus* ou *NoC*[20] dans le cadre de processeurs massivement multi-cœurs. L'interconnexion a pour fonction de propager des transactions, qui matérialisent les interactions entre les différents composants du processeur.

Les caches partagés. On trouve souvent en amont de la mémoire principale un ou plusieurs niveaux de caches qui sont accessibles par plusieurs cœurs. Ces caches partagés s'inscrivent dans la mémoire hiérarchique.

Les contrôleurs mémoire. Leur rôle est d'implémenter le protocole de communication avec les mémoires. Les contrôleurs vers la mémoire principale, qui est en général de type DDR[21] sont les plus complexes.

Les contrôleurs de périphériques. Il s'agit d'unités matérielles qui implémentent des protocoles de communication sur des périphériques tout en en masquant leur complexité au logiciel embarqué. À titre d'exemple, on peut considérer les bus PCIe, PCI et CAN qui sont utilisés dans les systèmes avioniques. Un contrôleur de périphériques dispose d'une interface de configuration qui est programmable.

Les accélérateurs matériels. Ce sont des unités qui ont pour but de réaliser efficacement certaines opérations très spécifiques afin d'en décharger les cœurs. Ces opérations concernent par exemple le transfert de données depuis et vers la mémoire que peut effectuer un contrôleur DMA[22]. On rencontre également des accélérateurs pour le traitement de paquets sur un réseau ethernet, des co-processeurs pour les opérations vectorielles efficaces pour le traitement du signal, ou encore des coprocesseurs spécialisés dans le traitement de flux videos. À l'exception des contrôleurs DMA, ces composants ne sont pas considérés dans notre étude car pas utilisés dans les calculateurs avioniques.

Le contrôleur d'interruptions. Le contrôleur d'interruptions agit comme un routeur, qui redirige les interruptions générées par des évènements matériels vers un ou plusieurs cœurs. Les deux standards existants sont APIC[23] qui est implémenté dans les processeurs Intel et AMD, et OpenPIC qui est utilisé par la majorité des processeurs PowerPC, ainsi que dans de nombreuses implémentations de processeurs ARM.

20. Network on Chip
21. Double Data Rate
22. Direct Memory Access
23. Advanced Programable Interrupt Controller

Les ressources de debogage et de monitoring. Ces ressources ont pour objectif de fournir des informations sur le fonctionnement du processeur. Ces informations ont pour finalité de diagnostiquer des sources de bugs dans le logiciel embarqué et dans le matériel, mais aussi de localiser les ressources ou les phases d'exécution dont l'efficacité pourrait être améliorée. On distingue deux types de ressources : les *moniteurs de performances*, qui sont des compteurs qui s'incrémentent lors des occurrences de certains évènements, et les modules de *trace*, qui maintiennent un historique de toutes les occurrences de certains évènements.

On peut regrouper les composants –que nous désignons également par le terme "périphériques"– décrits ci-dessus en trois catégories. Les périphériques dits *maîtres* sont proactifs. Ils initient de l'activité dans le processeur. On retrouve les cœurs, les contrôleurs DMA et certains contrôleurs de périphériques comme les contrôleurs de bus PCIe. À l'inverse, les composants *esclaves* n'initient pas d'activité mais répondent à celle qui est initiée par les périphériques maîtres. Enfin, l'interconnexion est un périphérique de communication, dont le seul rôle est de propager et contrôler l'activité qui résulte des échanges entre composants maîtres et esclaves.

La complexité électronique d'un processeur multi-cœurs est au final proche de celle d'un processeur mono-cœur. On présente souvent cette technologie comme étant en rupture, notamment du fait de la présence d'activités électroniques concurrentes émises par les cœurs à destination de ressources partagées. Bien qu'électroniquement des situations similaires existent sur des processeurs mono-cœurs, par exemple en activant des DMA, on trouve des problématiques purement spécifiques aux processeurs multi-cœurs. Nous développons certaines d'entre elles dans la section suivante.

2.4.2 Problématiques spécifiques aux processeurs multi-cœurs

La technologie multi-cœurs remet en cause un certain nombre de principes et de solutions qui ont été développées dans un cadre mono-cœur. Nous nous concentrons sur les aspects suivants, qui couvrent principalement les problématiques liées au déterminisme temporel, et aux systèmes IMA :

Évaluation du pire temps d'exécution d'une tâche. Comme introduit dans la section 2.1.4, il est nécessaire d'identifier les facteurs qui introduisent de la variabilité dans le temps d'exécution du logiciel embarqué. Un processeur multi-cœurs introduit de nouveaux composants et mécanismes qui peuvent altérer le temps d'exécution du logiciel. L'impact de ces mécanismes doit être quantifié. Nous développons ce point, central dans cette thèse, dans le chapitre 3.

Maîtrise de l'ordonnancement des tâches. L'ordonnancement des tâches est plus complexe, car il demande d'affecter chaque tâche à un cœur. Cette affectation peut être statique, on parle d'algorithme *partitionné*, ou dynamique, on parle d'algorithme *global*. De plus, dans les systèmes IMA, l'ordonnancement est hié-

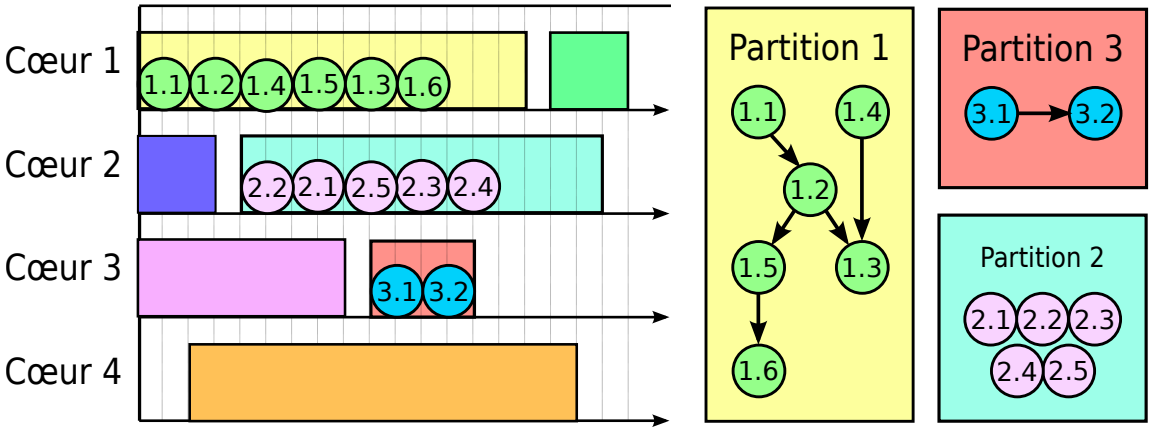


FIGURE 2.4 – Exemple de déploiement de partitions en configuration AMP

rarchique, ce qui laisse la possibilité d'introduire du parallélisme entre partitions et à l'intérieur d'une partition.

Dans le cadre de cette thèse, nous considérons des déploiements de partitions dans une configuration dite AMP [24], dans laquelle chaque partition est mono-cœur, plusieurs partitions de niveau de criticité arbitraire pouvant être actives simultanément. Cette configuration favorise la réutilisation de logiciels avioniques existants, conçus pour des processeurs mono-cœurs. De plus, elle favorise l'intégration de petites applications, qui sont par nature difficilement parallélisables. Un exemple de déploiement de partitions en configuration AMP est donné sur la figure 2.4.

Isolation entre applications. Sur une plateforme IMA, il est nécessaire d'assurer la propriété de Partitionnement Robuste, signifiant l'isolation des défaillances entre les partitions. Cela est classiquement fait par l'isolation complète de l'activité au niveau spatial et temporel entre les partitions. Sur un processeur multi-cœurs, l'exécution simultanée de plusieurs partitions sur différents cœurs rompt l'isolation temporelle. Les partitions peuvent en effet accéder simultanément à des ressources matérielles partagées entre les cœurs, en particulier la mémoire principale. La propriété de Partitionnement Robuste reste vraie lorsque les deux propriétés suivantes sont vraies :

- L'intégrité des données et des communications est préservée dans le processeur. Cela signifie que la présence d'activités non isolées n'entraînera jamais la perte d'un message, ou l'altération de ses données.
- Il est possible d'évaluer le pire temps d'exécution d'une tâche exécutée sur un cœur de manière indépendante des tâches exécutées sur les autres cœurs.

Réutilisation de composants logiciels. La réutilisation de composants logiciels existants est une problématique centrale dans l'Avionique Modulaire Intégrée. Elle se pose habituellement au niveau des applications avioniques, qui doivent être portables d'une plateforme à une autre sans connaître le système d'exploitation.

24. Asymetrical Multi-Processing

Pour le fournisseur de plateforme, la problématique se pose également pour le système d'exploitation. Lors du passage d'un processeur mono-cœurs à un processeur multi-cœurs, le design du système d'exploitation a vocation à être remis en question aussi peu que possible. Une solution répandue consiste à exécuter tout ou partie du système d'exploitation au sein d'une machine virtuelle. Cette dernière lui donne l'illusion d'être exécuté sur un processeur aux ressources restreintes, qui sont compatibles avec son implémentation. La couche logicielle qui implémente la ou les machines virtuelles est appelée *hyperviseur*.

Un développement des techniques de virtualisation est effectué dans l'annexe B.

L'introduction de processeurs multi-cœurs dans les équipements avioniques est prévue pour la prochaine génération d'équipements. Cette technologie soulève de nouveaux problèmes, qui sont pour certains directement liés à la présence de plusieurs cœurs, les autres étant inhérents aux technologies employées dans les nouvelles générations de processeurs. Nous nous intéressons dans ce manuscrit à la question de l'évaluation du pire temps d'exécution (WCET), qui fait l'objet de travaux depuis les années 2000.

2.5 Synthèse

Les choix effectués lors de la conception de systèmes avioniques résultent de plusieurs problématiques générales, dont nous avons donné une vue d'ensemble dans ce chapitre. Ces systèmes doivent ainsi répondre à une problématique de déterminisme qui se décline en plusieurs aspects, en particulier les aspects temporels auxquels nous nous intéressons dans cette thèse. On parle de satisfaire des exigences temps-réel dures. Cette problématique est abordée en appliquant des méthodes d'évaluation de pire temps d'exécution –ou WCET– impliquant à la fois le logiciel embarqué et le matériel. Le choix des composants embarqués, en particulier des processeurs, doit prendre en compte leur capacité à satisfaire ces exigences.

Pourtant, le critère de déterminisme, en particulier temporel, n'est pas mis en avant lors de la sélection de processeurs pour l'avionique. Par exemple, avant de chercher un processeur déterministe, un équipementier recherchera un processeur efficace pour un prix raisonnable. Les processeurs COTS répondent à cette problématique d'efficacité. L'équipementier recherchera ensuite un fabricant qui accepte de collaborer sur des problématiques présentes dans l'avionique, par exemple la durée de fabrication du processeur, qui doit être comprise entre dix et quinze ans au minimum, ou encore la durée de vie d'un composant, qui diminue avec les nouvelles technologies semi-conducteurs.

La collaboration avec un fabricant de processeurs apporte des réponses lorsque les problématiques sont partagées avec d'autres domaines d'applications, en particulier les infrastructures de télécommunications. C'est par exemple le cas des problématiques d'intégrité des données. Les processeurs actuellement exploités répondent bien à ces problématiques, et c'est sur cette base que leur sélection est faite en général. L'exigence de déterminisme temporel est souvent prise en compte après cette sélection. Sur les pro-

cesseurs actuellement en cours d'exploitation, qui sont mono-cœurs, cette exigence est considérée comme satisfaite à un niveau suffisant.

L'introduction de processeurs multi-cœurs dans l'avionique remet cependant en cause certaines pratiques et solutions éprouvées sur les processeurs mono-cœurs, mais également la manière d'exprimer les problématiques avioniques. Ainsi, l'exigence de déterminisme temporel dépend de la manière dont le logiciel est déployé sur tous les cœurs du processeur, qui dans un module IMA dépend d'autres problématiques comme la réutilisation de logiciel existant, ou encore l'isolation entre partitions ARINC 653.

Dans le cadre de cette thèse, nous nous sommes placés dans l'hypothèse d'un déploiement logiciel dans une configuration dite AMP. Cette dernière consiste à rendre chaque cœur fonctionnellement indépendant en lui associant un système d'exploitation ARINC 653 et un ensemble d'applications qui seront exécutées sur un seul cœur. Cette configuration favorise la réutilisation de logiciel existant, ainsi que l'exécution de petites applications, qui ne peuvent pas se paralléliser, et qui sont nombreuses dans les systèmes IMA.

Dans la configuration que nous avons adoptée, l'exigence de déterminisme temporel s'instancie de la même manière que sur les processeurs mono-cœurs. Nous cherchons donc à reprendre les pratiques existantes, ce qui consiste à appliquer des méthodes de calcul de WCET. Nous abordons dans le chapitre suivant la question de l'applicabilité de ces méthodes à des processeurs multi-cœurs, dans notre contexte industriel.

Chapitre 3

Évaluation du WCET dans les processeurs multi-cœurs

Sommaire

3.1 Méthodes existantes dans un contexte mono-cœur 34
3.1.1 Vue d'ensemble 35
3.1.2 Spécificités des méthodes statiques 36
3.1.3 Spécificités des méthodes dynamiques 37
3.1.4 Utilisation dans un processus industriel 38
3.2 Applicabilité des méthodes mono-cœurs dans un cadre multi-cœurs 38
3.2.1 Prédictibilité des processeurs multi-cœurs 39
3.2.2 Gestion du manque d'information sur les processeurs COTS 40
3.2.3 Techniques d'analyse d'interférences 41
3.3 Alternatives pour l'évaluation du WCET sur des processeurs multi-cœurs 45
3.3.1 Approches par pénalité d'interférences globale 45
3.3.2 Approches par processeurs déterministes 46
3.3.3 Approches par logiciel déterministe 47
3.4 Synthèse 49

Dans les systèmes temps réels durs, le logiciel embarqué doit s'exécuter en respectant certaines échéances. Le non respect d'une échéance constitue une défaillance simple qui, même si elle ne peut avoir à elle seule des conséquences catastrophiques comme la perte de l'avion (cf section 2.1.3), fait courir un risque d'une défaillance de plus haut niveau dans le système associé. La notion de calcul du pire temps d'exécution –ou WCET– d'un logiciel a été introduite pour rationaliser ce risque.

Définition 3.1. *Le WCET d'une tâche logicielle se définit comme une durée au delà de laquelle on estime, avec un risque acceptable, qu'elle aura terminé son exécution quels que soient ses paramètres d'entrée et la configuration initiale du matériel.*

Avec l'arrivée de processeurs multi-cœurs dans les équipements avioniques, la question s'est posée de la possibilité et de la pertinence à réutiliser les techniques existantes pour évaluer des WCET, ou à développer de nouvelles méthodes. Cette question n'est actuellement pas tranchée, ni dans la communauté académique, ni dans la communauté industrielle. L'objectif de cette section est donc de présenter les différentes pistes, puis de définir une orientation dans le cadre de cette thèse.

Nous développons dans la section 3.1 les méthodes de WCET développées dans la littérature. Nous abordons dans la section 3.2 la question de leur application sur des processeurs multi-cœurs, qui est rendue difficile par la présence d'interférences. Nous développons enfin dans la section 3.3 des approches alternatives développées spécifiquement pour les processeurs multi-cœurs.

3.1 Méthodes existantes dans un contexte mono-cœur

Les méthodes de calcul de WCET existantes dans la littérature, dont un recensement a été effectué en 2008 par Wilhelm [100], et dans l'industrie sont assez diverses. Elles sont habituellement regroupées en trois familles :

Les méthodes analytiques dites aussi **statiques.** Ces méthodes mettent en œuvre des techniques d'analyse statique effectuées sur le programme et éventuellement un modèle du processeur.

Les méthodes empiriques dites aussi **dynamiques.** Ces méthodes sont basées sur des mesures de temps d'exécution du logiciel dans un ensemble de configurations considéré comme représentatif.

Les méthodes hybrides qui combinent les deux techniques précédentes.

Ces méthodes visent à retourner une évaluation du WCET, et lorsque c'est possible les conditions –paramètres d'entrées et état du processeur– dans lesquels ce WCET a été obtenu. Cette évaluation vise à être la plus *précise* possible, dans le sens où le WCET évalué et le temps d'exécution mesuré dans ces conditions, ou à défaut le pire temps mesuré lorsque ces dernières sont réalisables, doivent être les plus proches possibles. La précision d'une méthode de WCET dépend d'un certain nombre de facteurs portant sur la structure du logiciel qui peut être difficile à analyser, le matériel qui peut avoir un comportement impactant fortement le WCET, même si très improbable, et enfin la méthode qui peut faire des approximations trop pessimistes.

Une autre propriété appréciable pour une méthode de calcul de WCET est la *sûreté*. Une méthode sûre, dont les hypothèses de validité sont vraies, retourne par construction un majorant du pire temps d'exécution réel. Le risque sur le résultat est donc nul, seul demeure le risque sur la méthode.

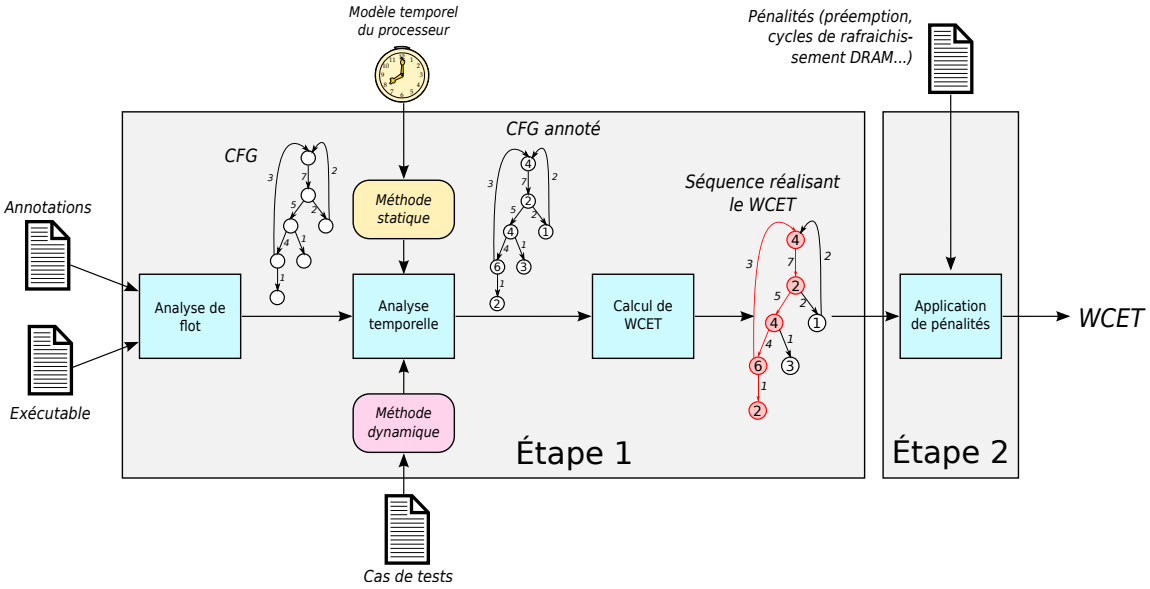


FIGURE 3.1 – Vue d'ensemble des différentes étapes du calcul d'un WCET

3.1.1 Vue d'ensemble

Si on considère une tâche intégrée au sein d'un système multi-tâches comme l'est un système IMA, l'évaluation de son WCET est effectuée en deux étapes, illustrées sur la figure 3.1 :

- L'évaluation du pire temps d'exécution de cette tâche en supposant que son exécution sera ininterrompue. Il s'agit d'une hypothèse simplificatrice. Cette tâche sera effectivement interrompue lors de son exécution, que ce soit par le système d'exploitation qui est préemptif, par l'expiration d'un timer, ou encore par une interruption externe.
- L'application de pénalités et d'éventuelles marges de sûreté couvrant le coût temporel des interruptions, ainsi que certains phénomènes matériels dont la prise en compte dans l'étape précédente est compliquée. C'est le cas par exemple des cycles de rafraichissement d'une DRAM.

Les méthodes que l'on rencontre dans la littérature portent essentiellement sur la première étape, qui est la plus complexe d'un point de vue exploratoire. Celle-ci demande en effet d'analyser l'espace d'états du processeur, par exemple le contenu du pipeline et des différents niveaux de cache, ainsi que les chemins d'exécution possibles à l'intérieur de la tâche. Cette première étape est constituée des phases d'analyses suivantes :

Analyse de flot. (Control Flow Analysis). Cette première étape vise à retrouver le graphe de contrôle, ou CFG (Control Flow Graph) à partir du code binaire du logiciel à analyser. Ce graphe décrit les différentes exécutions possibles du logiciel. Chaque nœud de ce graphe, appelé *bloc de base*, correspond à une fonction ou à une boucle. Un bloc de base peut être appelé un certain nombre de fois, dans des contextes différents.

Une fois le CFG défini, il est nécessaire d'explorer les différents chemins d'exécution pour déterminer le nombre d'itération de chaque boucle, ainsi que d'éventuelles branches infaisables. On parle d'***analyse de valeurs*** (Value Analysis). Cette étape soulève des problèmes d'explosion combinatoire lors de l'exploration du binaire [57]. Ces problèmes sont abordés par des techniques d'analyse statique.

Cette première phase d'analyse est effectuée sur le code binaire du logiciel de manière à se rendre indépendant de la chaîne de compilation.

Analyse temporelle. (Timing analysis). Cette deuxième étape vise à évaluer le pire temps d'exécution de chaque bloc de base. Cette analyse peut éventuellement être effectuée sur un bloc une fois par contexte d'appel pour tirer parti de la connaissance de l'historique d'exécution. C'est le cas par exemple dans l'outil aiT développé par AbsInt [55].

Dans cette étape, les méthodes statiques et dynamiques diffèrent. Les méthodes statiques effectuent l'analyse temporelle en simulant au cycle près l'état d'un modèle du processeur exécutant le code du bloc de base. On parle ***d'analyse matérielle***. Ce modèle décrit en général l'état du pipeline et des caches, et intègre des pires temps d'accès aux autres ressources matérielles comme la mémoire principale.

Dans les méthodes dynamiques, le temps d'exécution d'un bloc de base est mesuré sur chaque chemin d'exécution possible. Cette mesure demande de placer le processeur dans la configuration la plus défavorable avant exécution du bloc de base, par exemple en invalidant les caches.

A la fin de cette étape, le CFG est annoté avec le pire temps d'exécution de chaque bloc de base, éventuellement par contexte d'appel.

Estimation du temps d'exécution. (Estimate Calculation.) Cette dernière étape vise à analyser le CFG annoté afin d'identifier les chemins de plus longue durée. Il existe diverses approches pour identifier ce plus long chemin, qui mettent en œuvre des techniques d'exploration de graphe ou de programmation linéaire. Nous nous référons à la classification donnée par Wilhelm [100, Section 3.4]

Ce schéma menant à l'évaluation du WCET est aujourd'hui développé dans différents outils académiques ou commerciaux. Parmi les plus connus on peut citer aiT [55] (AbsInt, commercial), RapiTime (Rapita Systems, commercial) Otawa [17] (IRIT Toulouse, open source), Heptane (IRISA). Nous nous référons au recensement de Wilhelm [100] pour une liste plus exhaustive.

Les sections suivantes développent les spécificités des méthodes statiques et des méthodes dynamiques, puis abordent la question de l'intégration d'une méthode d'évaluation de WCET dans un processus industriel de type IMA.

3.1.2 Spécificités des méthodes statiques

Lors de la phase d'analyse temporelle, les méthodes statiques se basent sur l'analyse de modèles de processeurs. Pour des raisons historiques, ces modèles sont très détaillés

au niveau du cœur, avec une description fine du pipeline et des caches [38]. D'une part ces éléments sont dimensionnants pour le WCET du logiciel embarqué, et d'autre part ils sont à l'origine d'*anomalies temporelles* (timing anomalies) [97], parfois appelées *effets dominos* [60] dans la littérature. Ces phénomènes correspondent à des situations où le comportement en pire cas d'un composant ne peut être déduit du comportement en pire cas de chacun de ses sous-composants. Une analyse globale, plus complexe, est nécessaire. A contrario, ces modèles sont peu détaillés pour décrire le comportement des périphériques. Ils détaillent la durée dans le pire cas de chaque opération vers les périphériques. Ainsi, on mentionnera des données comme le pire temps de lecture dans la mémoire principale, que l'on reportera systématiquement dans l'analyse temporelle.

La définition du modèle de processeur est importante dans l'usage d'une méthode statique car cette dernière hérite de ses propriétés. Ainsi, lorsque le modèle est sûr, i.e. décrit correctement les mécanismes matériels implémentés dans le processeur, le WCET retourné par la méthode est un majorant du WCET réel. La méthode est donc sûre. À l'inverse, certaines approches, basées sur des modèles statistiques de processeurs, retournent des estimations probabilistes du WCET [21; 29]. Ces méthodes se revendiquent comme plus précises, ce qui est le cas lorsque les situations de pire cas au niveau matériel sont très improbables et peuvent être relâchées.

La justesse[25] et la précision d'une méthode statique dépend donc de celle du modèle de processeur. À ce titre, l'utilisation de processeurs COTS représente une difficulté supplémentaire du fait du manque d'informations communiquées par le fabricant. Ce manque d'informations se ressent principalement sur les périphériques et la mémoire principale, vis-à-vis desquels les modèles de processeurs sont en général minimalistes, i.e. réduits à la liste des temps d'accès.

3.1.3 Spécificités des méthodes dynamiques

Les méthodes dynamiques, largement employées dans l'industrie, ont fait l'objet d'une littérature moins abondante que les méthodes statiques. La problématique principale consiste à automatiser les mesures de temps d'exécution de segments de code [102]. Les approches portent sur les techniques permettant de collecter des temps d'exécution par des techniques non intrusives.

Les méthodes dynamiques ne sont pas présentées comme sûres. Cela vient du fait qu'il est difficile d'assurer à la fois une couverture complète du code et des états du processeur. Toutefois, certaines approches [30] permettent d'obtenir une évaluation probabiliste sur le WCET. D'une manière générale, l'usage de méthodes dynamiques sur un processeur est approprié lorsque le processeur a un comportement reproductible du point de vue temporel, ce qui limite le nombre d'échantillons à collecter pour garantir une couverture statistique correcte.

25. Soundness

3.1.4 Utilisation dans un processus industriel

Dans les méthodes rencontrées dans la littérature et les pratiques industrielles, toutes les étapes de l'évaluation du WCET ne sont pas nécessairement portées à un niveau de maturité permettant de les automatiser. De plus, dans un processus industriel tel que l'IMA, le logiciel exécuté est fourni par différents acteurs, qui par nature partagent une quantité limitée d'informations. Par exemple un fournisseur d'application sera amené à évaluer le WCET d'une tâche qu'il aura développée, en sachant que cette tâche sera contrôlée par le système d'exploitation, dont il n'a pas les sources, et que ce système d'exploitation exercera ce contrôle à des dates et dans des circonstances qu'il ne maîtrise pas.

En conséquence, le processus IMA [5] prévoit que le fournisseur de plateforme est tenu de diffuser toutes les informations nécessaires, ainsi que des plateformes matérielles représentatives, pour que le WCET des applications embarquées puisse être évalué en tenant compte des caractéristiques de la plateforme et du système d'exploitation.

L'évaluation du WCET est un champ de recherche vaste, qui s'est beaucoup développé ces dernières années en suivant l'évolution des processeurs. On dispose aujourd'hui d'outils matures pour une utilisation industrielle, ces outils supportant un certain nombre de processeurs COTS. L'arrivée de processeurs multi-cœurs soulève la question de la possibilité d'appliquer ces méthodes avec peu ou pas de changements. Nous développons cet aspect dans la section suivante.

3.2 Applicabilité des méthodes mono-cœurs dans un cadre multi-cœurs

Le calcul de WCET dans les processeurs multi-cœurs est un problème posé depuis le début des années 2000. Les approches proposées dans la communauté reprennent le schéma présenté dans la section 3.1, à savoir une phase d'analyse effectuée individuellement sur chaque tâche suivie d'une phase d'application de pénalités. Le problème porte plus précisément sur la possibilité de réutiliser des méthodes actuellement appliquées sur les processeurs mono-cœurs dans un cadre multi-cœurs.

Nous considérons la première phase de l'évaluation du WCET, qui est la plus développée dans la littérature. Nous cherchons à calculer le pire temps d'exécution d'une tâche séquentielle et ininterrompue. L'application d'une méthode statique demandera un modèle de processeur, qui si on se réfère aux modèles habituels est détaillé au niveau du cœur. Les cœurs de processeurs multi-cœurs sont d'un niveau de complexité semblable aux cœurs de processeurs mono-cœurs. À l'inverse, la complexité des interconnexions et autres ressources communes est nettement plus élevée sur un processeur multi-cœurs. De ce fait, la principale difficulté est de satisfaire la propriété de prédictibilité, définie ci-après.

Définition 3.2 (Prédictibilité d'un processeur multi-cœurs)**.** *Un processeur multi-cœurs est prédictible lorsque pour toute configuration atteignable sur le processeur, la durée de chaque opération effectuée d'un cœur vers une ressource partagée est bornée.*

De la même manière, l'application d'une méthode dynamique demandera d'effectuer une série de mesures devant couvrir un espace d'état dont la taille augmente exponentiellement. En pratique, la propriété de prédictibilité telle que définie ci-dessus est incontournable car elle permet d'éviter cette explosion combinatoire.

In fine, méthodes statiques et dynamiques peuvent s'appliquer sur un processeur à condition qu'il soit prédictible, pour des raisons théoriques dans le premier cas –la construction d'un modèle de processeur correct suppose la prédictibilité–, et des raisons pratiques dans le second –la prédictibilité permet de réduire le nombre de cas de tests. Or assurer la propriété de prédictibilité constitue une exigence dans les processeurs multi-cœurs. Nous détaillons ces aspects dans la section suivante.

3.2.1 Prédictibilité des processeurs multi-cœurs

La prédictibilité d'un processeur, selon la définition 3.2, est obtenue lorsque l'on dispose d'une borne sur le temps d'accès de chaque cœur à chaque périphérique et mémoires partagées. Dans les processeurs multi-cœurs, la principale difficulté vient de la présence d'interférences, pour lesquelles nous proposons la définition suivante :

Définition 3.3 (Interférences)**.** *La présence d'interférences est définie comme l'augmentation de la durée d'une opération effectuée par un périphérique, par exemple un cœur, sur une ressource commune due à la présence d'une ou plusieurs opérations effectuées par d'autres périphériques.*

L'impact des interférences s'observe en général au niveau macroscopique par une augmentation du temps d'exécution du logiciel, mais également une augmentation de sa variabilité. Toute méthode d'évaluation de WCET doit prendre en compte cet impact à travers une *pénalité*, qui sera appliquée dans l'une des deux phases d'évaluation du WCET.

Comme illustré sur la figure 3.2, une opération sur une ressource commune correspond électroniquement à la transmission et au traitement d'une ou plusieurs requêtes. Les causes des interférences sont à chercher à la fois dans les composants qui propagent les requêtes, c'est-à-dire les interconnexions, et ceux qui les traitent, c'est-à-dire les contrôleurs de mémoire et de périphériques. Il est nécessaire d'analyser le comportement de chaque composant pour identifier et quantifier l'impact des interférences. Nous parlons par la suite d'***analyse d'interférences***.

Dans notre contexte, marqué entre autres par l'utilisation de processeurs COTS, nous rencontrons plusieurs obstacles à la faisabilité d'une analyse d'interférences, obstacles que nous détaillons dans la suite de cette section.

Exemple 3.1 – Exemple de séquence d'instructions en assembleur PowerPC

```
li      r0,1
stw     r0,60(r1)
lwz     r0,64(r1)
stb     r0,8(r1)
addi    r1,r1,56
```

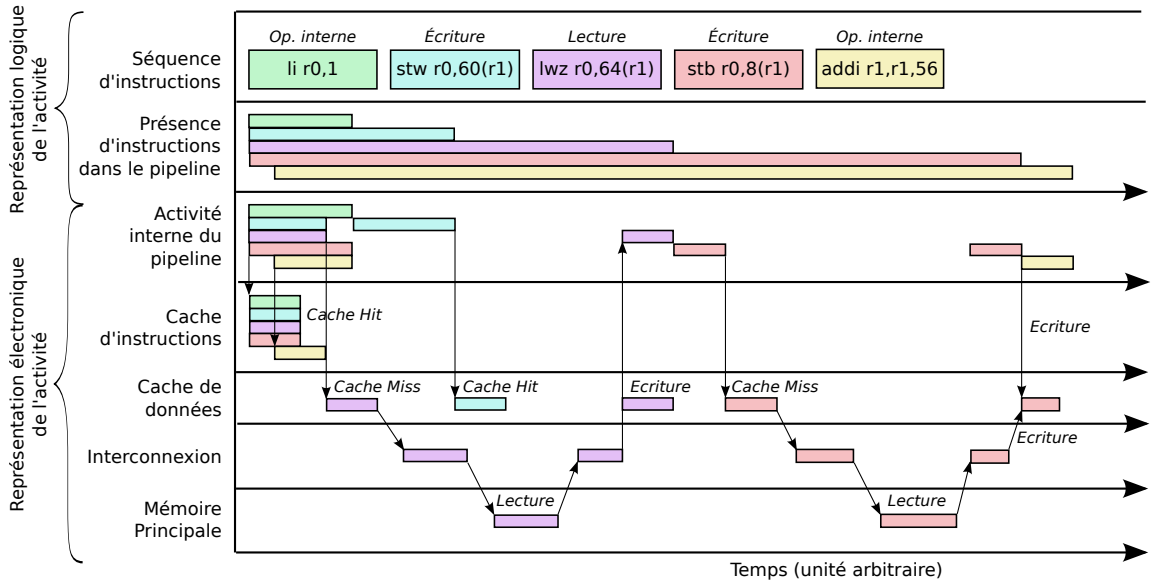


FIGURE 3.2 – Chronogramme représentant l'activité interne d'un processeur PowerPC exécutant une séquence d'instructions simple

3.2.2 Gestion du manque d'information sur les processeurs COTS

Le premier obstacle est lié à la contrainte d'utilisation de processeurs COTS. Il s'agit du manque d'informations communiquées par les fabricants sur leurs processeurs.

L'architecture des cœurs est en général correctement documentée pour permettre le développement de compilateurs optimisés. À l'inverse, la documentation du reste du processeur est en général restreinte à une description fonctionnelle des périphériques. Cette description peut être volontairement rendue incomplète par le fabricant pour éviter de communiquer des secrets industriels. C'est le cas notamment pour les interconnexions, qui sont dimensionnantes pour les performances du processeur. Une bonne connaissance de l'architecture d'une interconnexion est pourtant nécessaire pour y mener une analyse d'interférences.

Pour pouvoir définir correctement l'espace d'état d'une interconnexion, il est nécessaire de connaître avec un niveau de détails suffisant son architecture, les politiques d'arbitrages supportées, ainsi que son interface de configuration. Lorsque le processeur contient plusieurs niveaux d'interconnexions, cette exigence s'applique à chaque niveau.

Or, on ne dispose que rarement de la totalité de ces informations. On rencontre les situations suivantes :

Architecture non documentée. Parfois, l'interconnexion est une boîte noire, comme CoreNet, qui équipe les processeurs de la gamme QorIQ développée par Freescale, à laquelle l'industrie aéronautique s'intéresse. Seul le protocole de communication est décrit dans un brevet de Freescale [33].

Politique d'arbitrage non documentée. On peut relever les processeurs DSP [26] de la gamme TMS320 fabriquée par Texas Instruments [93]. Pour cette famille de processeurs, l'interconnexion, appelée TeraNet, est décrite dans le manuel de référence, incluant la matrice d'interconnexion et les registres de priorité. Toutefois, la politique d'arbitrage, qui se base sur des priorités statiques, n'est pas précisée.

Description limitée à certaines IP. On dispose de spécifications assez détaillées de certaines IP comme CoreLink [15], développée par ARM. Toutefois, les processeurs de la famille ARM qui l'embarquent contiennent également un bus d'interconnexion appelé *Snoop Control Unit* (SCU) qui est dépendante de l'implémentation et non documenté par ARM.

En pratique, l'architecture d'un processeur est rarement une boîte noire dont on n'a aucune information, mais plutôt une boîte plus ou moins grise. Il est possible d'effectuer des analyses d'interférences en utilisant des méthodes prenant en compte ce manque d'informations. Roger et Brindejonc [78] ont proposé le modèle Initiateur-Cible (initiator-target), à partir duquel ils identifient les cas de tests nécessaires pour une analyse d'interférences. Cette approche n'aborde cependant pas la question du nombre d'itérations à effectuer sur un cas de test donné pour obtenir une couverture correcte des interférences, qui à notre connaissance est toujours ouverte.

La gestion du manque d'information représente donc un premier obstacle, pour lequel la seule approche recommandée dans la littérature consiste à réduire la couverture de tests à un nombre atteignable, et faire en sorte que l'utilisation du processeur reste toujours couverte par ces tests.

3.2.3 Techniques d'analyse d'interférences

Supposons le point précédent résolu, et considérons un processeur pour lequel nous disposons d'un bon niveau d'informations sur l'architecture. Nous disposons des données permettant d'effectuer l'analyse d'interférences.

Pour effectuer une analyse d'interférences sur le processeur, on peut évaluer pour chaque composant la variabilité du temps de traitement –ou de traversée dans le cas d'une interconnexion– d'une requête, en faisant varier son état interne et les types de requêtes entrantes. L'objectif est d'identifier le pire cas d'interférences pour chaque composant.

Nous détaillons ci-après différents types d'analyses d'interférences, ou assimilées, que nous avons rencontré dans l'état de l'art.

26. Digital Signal Processing

Approches empiriques par benchmarks ciblés

La plupart des approches ciblant les processeur COTS sont empiriques, et visent à stresser un composant en particulier avec différents types de benchmarks pour observer les perturbations dues aux interférences. Elles doivent adresser les difficultés suivantes :

- Il est difficile de cibler un composant du processeur en particulier dans un benchmark de stress. Ce dernier est limité par la possibilité d'un cœur, qui exécute le benchmark, à initier un stress efficace [24].
- Il faut distinguer les interférences de la variabilité naturelle du temps de traitement des requêtes entrantes par un composant. Or on observera systématiquement la somme des deux.

D'une manière générale, atteindre avec des benchmarks de stress la pire situation d'interférences dans un composant est un problème complexe, et non résolu dans la majeure partie des cas [75]. Toutefois, certaines approches rencontrées dans la littérature proposent des méthodes permettant d'estimer des pénalités d'interférences réalistes :

- Bin [24] a proposé la notion de signature d'une application, visant à quantifier le stress que celle-ci exerce sur chaque composant. Cette signature est obtenue par des benchmarks stressant de manière ciblée chaque composant. La combinaison des signatures de plusieurs applications permet d'estimer la charge globale exercée sur chaque composant du processeur, et d'identifier les composants à l'origine d'interférences. L'expérimentation de cette méthode est effectuée sur un P4080, qui est un processeur octo-cœurs fabriqué par Freescale.
- Radojkovic [75] propose une démarche proche de celle de Bin, expérimentée sur un processeur Intel Atom.
- Nowotsch [65] propose la notion de *WCET sensible aux interférences* (isWCET) dans laquelle chaque application dispose d'un budget en terme de stress exercé sur chaque ressource commune. Le WCET est calculé en appliquant une pénalité d'interférences déterminée à partir des budgets alloués aux différentes applications. À l'exécution, le système d'exploitation vérifie que chaque application respecte son budget.

Analyse d'interférence dans l'interconnexion

L'interconnexion est l'élément central dans un processeur multi-cœurs. Elle assure la propagation de requêtes, souvent appelées *transactions* dans la littérature. Le temps de traversée d'une interconnexion est la combinaison d'une latence technologique et d'une durée d'arbitrage, qui correspond au temps où une transaction est mise en attente pour obtenir l'autorisation d'être propagée. Selon la manière dont l'interconnexion est conçue, l'arbitrage peut être centralisé ou distribué. Une transaction peut même se faire arbitrer plusieurs fois au cours de sa traversée.

Les analyses d'interférences effectuées sur l'interconnexion portent principalement sur l'analyse des politiques d'arbitrage. Un recensement effectué par Bourgade [28, section 2.3] montre que certaines politiques n'amènent pas à une pénalité d'interférences bor-

née. C'est le cas des protocoles à priorités fixes, et de certaines politiques "First In First Out", pour lesquelles le temps de mise en attente dépend de la charge en nombre de transactions émise par chaque cœur. À l'inverse, l'auteur relève des familles de politiques d'arbitrage qui permettent d'obtenir une pénalité d'interférences acceptable. Ces politiques sont basées sur un ordonnancement statique des accès de chaque cœur aux ressources communes. On parle souvent de politique TDMA [27].

Le protocole d'interconnexion peut également être à l'origine d'interférences. Une étude de ce type a été effectuée par Shah [88] sur le protocole AMBA, qui est répandu dans les architectures ARM. Il a mis en évidence des phénomènes de verrouillage de bus (bus locking) qui peuvent entraîner un ralentissement entre des transactions successives, après la phase d'arbitrage. L'auteur relève que ce type d'interférence est souvent négligé dans les analyses menées sur l'interconnexion.

La politique d'arbitrage ainsi que celle du protocole d'interconnexion sont des éléments déterminants de l'analyse d'interférences. Sur des architectures simples on dispose de solutions qui sont prometteuses. La difficulté consiste à les appliquer sur des processeurs COTS, sur lequels la politique d'arbitrage est complexe, optimisée pour certaines opérations, et surtout non documentée. À notre connaissance, ce problème n'a pas été résolu dans la littérature.

Interférences causées par la cohérence de cache

Le trafic lié à la cohérence de caches est une source d'interférences connue dans les processeurs multi-cœurs, au point que divers travaux [45; 56] recommandent d'éviter son usage en désactivant les mécanismes associés. En effet, la cohérence de cache entraîne un trafic qui est propagé vers tout ou partie des cœurs, et qui augmente de manière quadratique avec le nombre de cœurs. C'est un des facteurs limitant pour l'augmentation du nombre de cœurs dans les processeurs qui offrent ce mécanisme.

L'exemple le plus frappant a été décrit par Nowotsch et Paulitsch [66] sur un processeur P4080, qui est un octo-cœurs fabriqué par Freescale. Les auteurs ont montré que dans des conditions où tous les cœurs initient un trafic important dans des plages mémoire proches, ce qui stresse les mécanismes de cohérence de caches, le trafic global est ralenti d'un facteur 20.

Dans un contexte IMA où le logiciel est fonctionnellement indépendant sur chaque cœur, la cohérence de caches n'est pas nécessaire au bon fonctionnement du logiciel embarqué, dans la mesure où il n'y a pas de partage de ressources entre les cœurs. Nous considérons donc dans nos travaux la question des interférences dues au trafic de cohérence de caches comme secondaire.

27. Time Division Multiple Access

Interférences sur les caches partagés

La plupart des processeurs multi-cœurs possèdent un niveau de cache partagé. Du point de vue d'un cœur, son contenu est susceptible d'être altéré de manière arbitraire, selon l'exécution du logiciel sur les autres cœurs. Divers travaux dans la littérature visent à borner cet impact, par exemple l'approche développée par Hardy [50] ou encore par Li [58].

Par ailleurs, il est souvent possible de partitionner ces caches, ce qui a pour finalité de simuler des caches privés, l'accès restant concurrent. Sur la plupart des processeurs, les caches partagés peuvent également être configurés comme des mémoires statiques, dont le contenu est déterminé à l'avance. Dans un contexte IMA où le logiciel concurrent n'est pas toujours connu, cette approche est recommandée.

Interférences sur les contrôleurs DRAM

Les contrôleurs de mémoires de type DRAM effectuent de nombreux types d'opérations. Ils gèrent une mémoire organisée en banques, dont ils contrôlent l'état ouvert ou fermé, avec un nombre limité de banques ouvertes simultanément. Lorsqu'une banque est ouverte, son contenu peut être lu et modifié. La banque est organisée en lignes, que le contrôleur charge, et en colonnes qui sont toutes accessibles dans la ligne ouverte. Les ouvertures de lignes et ouvertures de banques entraînent des retards dans la réponse des contrôleurs de mémoire. De plus, ces derniers gèrent des files d'attente de requêtes entrantes qu'ils réordonnent pour minimiser les ouvertures de banques et/ou de lignes. Ces mécanismes sont à l'origine de nombreuses interférences.

Ainsi, Moscibroda a montré [63] que des applications peuvent interférer jusqu'à ralentir d'un facteur 3 lorsqu'elles utilisent le même contrôleur mémoire. L'auteur insiste sur le faible niveau de stress exercé par ses applications sur l'interconnexion, ce qui ramène toutes les interférences sur le contrôleur de mémoire.

On rencontre des méthodes d'analyse d'interférences sur les contrôleurs DRAM. Ainsi, Wu [101] propose d'intégrer une analyse d'interférences sur le contrôleur DRAM au cours de l'évaluation du WCET. Les résultats obtenus permettent d'améliorer les WCET obtenus par rapport à des contrôleurs de mémoire déterministes. À notre connaissance, cette méthode n'a pas été étendue à des contrôleurs de processeurs COTS.

Synthèse

Il existe un ensemble assez vaste de techniques d'analyses d'interférences que l'on peut appliquer sur un processeur, en supposant que l'on dispose d'informations sur son architecture et son comportement. Ces techniques sont pour la plupart locales à un composant, comme une interconnexion, un cache ou un contrôleur de mémoire, ou à des mécanismes, par exemple la cohérence de caches.

À notre connaissance, ces techniques n'ont pas encore été mises en œuvre sur des processeur COTS et si c'était le cas, le résultat serait incertain. Dans certains cas, l'analyse d'interférences ne passe pas à l'échelle sur des processeurs COTS dont l'architecture est complexe. Elle peut également être trop pessimiste car reposant sur des approximations du matériel. Certaines analyses ont également mis en évidence des cas d'interférences très importantes, qu'il faudrait pourtant intégrer systématiquement dans le calcul de la pénalité d'interférences. Cela rend les processeurs en question inutilisables pour une utilisation temps réel dure.

Nous détaillons dans la section suivante diverses techniques alternatives, qui visent à contourner les limitations des analyses d'interférences présentées précédemment.

3.3 Alternatives pour l'évaluation du WCET sur des processeurs multi-cœurs

Le problème consistant à évaluer l'impact des interférences sur les temps d'accès des cœurs aux ressources partagées dans un processeur multi-cœurs COTS est reconnu comme difficile. Sa résolution constitue l'approche la plus directe pour permettre de réutiliser les méthodes d'évaluation de WCET existantes dans un cadre multi-cœurs.

Compte tenu des obstacles que nous avons décrits dans la section précédente, nous considérons qu'une approche "classique" visant à résoudre ce problème n'est pas pertinente. Ce constat, partagé dans la communauté, a motivé différentes approches alternatives, que nous classons dans trois catégories :

Pénalité globale. Les approches à pénalité globale visent à évaluer une pénalité d'interférences à partir des profils d'accès aux ressources pour chaque tâche embarquée. Elles demandent dont de connaître la totalité des tâches déployées sur le processeur.

Processeur déterministe. Ces approches visent à rendre le processeur prédictible par construction. Leur apport principal consiste à introduire des mécanismes d'arbitrage matériels permettant d'isoler temporellement l'activité électronique émise par chaque cœur. Cependant, ces approches n'ont, à notre connaissance, pas encore été suivies par les fabricants de processeurs COTS.

Logiciel déterministe. Les approches par logiciel déterministe visent à contraindre le logiciel à respecter un domaine d'usage où il est démontré que le processeur est prédictible.

Nous détaillons ces trois familles d'approches dans la suite de cette section.

3.3.1 Approches par pénalité d'interférences globale

Les approches par pénalité globale partent du principe que l'analyse d'interférences n'aboutit pas dans un cas général car il faut atteindre les pires cas d'interférences sur les

ressources communes, ce qui est impossible sur un processeur COTS. Toutefois les tâches, au cours de leur exécution, ne réalisent pas le pire cas d'interférences. L'enjeu est donc d'extraire des tâches embarquées les informations concernant leur usage des ressources communes, puis de déterminer une pénalité d'interférences globale qui s'applique à chaque tâche.

L'approche la plus aboutie dans ce sens a été proposée par Sha [86], et porte un concept de "machine virtuelle équivalente à un mono-cœur" (Single-Core Equivalent Virtual Machine). Cette approche combine des travaux visant à réduire les interférences par une utilisation fine de ressources comme les caches partagés et les contrôleurs mémoire, et à allouer des budgets à chaque tâche en termes d'utilisation des ressources. Chaque accès incrémente un compteur matériel, et une tâche de contrôle est chargée de vérifier que chaque tâche ne consomme pas plus d'accès que ce que son budget lui autorise.

Cette approche est prometteuse, mais elle ne vérifie pas au sens strict la propriété de Partitionnement Robuste. La validité du WCET obtenu dépend du fait que toutes les tâches respectent leur budget. Or, si une tâche ne respecte pas son budget –ce qui est une défaillance–, elle risque de ralentir les autres tâches plus que prévu. Ces dernières risquent donc de dépasser leur WCET, ce qui correspond également à une défaillance. Il y aura eu propagation de défaillances entre les tâches, et donc invalidation de la propriété de Partitionnement Robuste. Cette limitation est toutefois à nuancer, la solution prévoit un mécanisme de détection de violation du partitionnement robuste, cette violation pouvant être latente durant une période de la tâche de contrôle au plus. Cette considération demande toutefois un développement plus poussé, qui sort du cadre de nos travaux.

Nous présentons dans la section suivante des approches par élimination des interférences.

3.3.2 Approches par processeurs déterministes

Les approches de type processeurs déterministes visent à définir des architectures de processeurs prédictibles par construction, c'est-à-dire sans interférences. Les approches que l'on peut rencontrer se concentrent principalement sur le développement d'interconnexions et de contrôleurs mémoire déterministes. Certaines ont abouti à des prototypes de processeurs synthétisables sur cible FPGA.

Nous avons recensé cinq prototypes de processeurs, développés dans le cadre des projets Merasa [94], PRET [59], CompSOC [49], ACROSS [82] et JOP [83]. Le point commun de ces processeurs est de proposer une interconnexion implémentant une politique d'arbitrage de type TDMA [28]. Cette politique d'arbitrage consiste à allouer des fenêtres temporelles disjointes à chaque cœur, dans lesquelles ils ont un accès exclusif aux ressources communes. Ainsi, l'activité électronique liée à chaque cœur est isolée et ne peut pas interférer. La configuration peut être équitable, c'est-à-dire donner des fenêtres de tailles égales à chaque cœur, ou favoriser un cœur par rapport à d'autres. Ainsi, Schoeberl propose une méthode itérative [83] pour optimiser le dimensionnement des

28. Time Division Multiple Access

fenêtres temporelles de telle sorte à favoriser les cœurs qui hébergent les tâches qui ne respectent pas leurs échéances.

Un inconvénient venant avec la politique TDMA est la faiblesse des performances des processeurs, ce qui explique qu'aucun fabricant ne l'aie adoptée jusqu'à présent. Pour palier ce manque de performances, Shah propose [87] une politique de priorité tournante selon laquelle chaque cœur aura de manière exclusive la priorité maximale dans une fenêtre temporelle donnée. Cette politique apporte des garanties de bande passante à chaque cœur tout en permettant de récupérer au profit des cœurs la bande passante non utilisée par les cœurs de haute priorité. Cependant, à notre connaissance, elle n'est pas non plus implémentée dans des processeurs COTS.

D'autres approches se sont tournées vers la définition de contrôleurs mémoire déterministes. Akesson a proposé en 2007 Predator [10], un contrôleur mémoire prédictible supportant le standard DDR2, plus tard étendu pour supporter le standard DDR3. Ce contrôleur est basé sur la mise en œuvre de motifs d'accès à la mémoire (memory patterns), avec pour chacun d'eux un pire temps de réponse associé. Un arbitre interne est en charge d'entrelacer les requêtes entrantes selon un algorithme à budget de priorité [11], qui offre des garanties en termes de bande passante et de latence. Un contrôleur de mémoire similaire a été développé par Paolieri [67] en 2009.

Les approches à base de processeurs déterministes portent des concepts d'arbitrage prédictible, couramment utilisés en ordonnancement dans les systèmes temps réels, au niveau électronique. Même si aucun processeur COTS n'a pour le moment repris ces principes, ces derniers constituent des solutions matures.

3.3.3 Approches par logiciel déterministe

Les approches de type logiciel déterministe se basent sur un domaine d'usage dans lequel le processeur est prédictible. Le logiciel est en charge de respecter ce domaine d'usage. On distingue dans les approches existantes celles où le logiciel est conçu pour respecter de lui-même le domaine d'usage –on parle de logiciel auto-contenu–, et celles pour lesquelles un logiciel dédié, que nous appelons *logiciel de contrôle* dans la suite de ce manuscrit, force le reste du logiciel à respecter le domaine d'usage.

Approche par logiciel auto-contenu

Dans les approches que l'on peut rencontrer dans la première catégorie, le logiciel est décomposé en phases d'exécution interne depuis une mémoire locale, et en phases de communication, lors desquelles il effectue les lectures et écritures sur les entrées/sorties et d'éventuels transferts de mémoire. Le logiciel est configuré pour ne pas communiquer sur l'interconnexion lors des phases d'exécution, si bien qu'il ne peut générer aucune interférence durant cette phase. Un ordonnancement statique garantit qu'il y a toujours au plus un cœur qui se trouve dans une phase de communication.

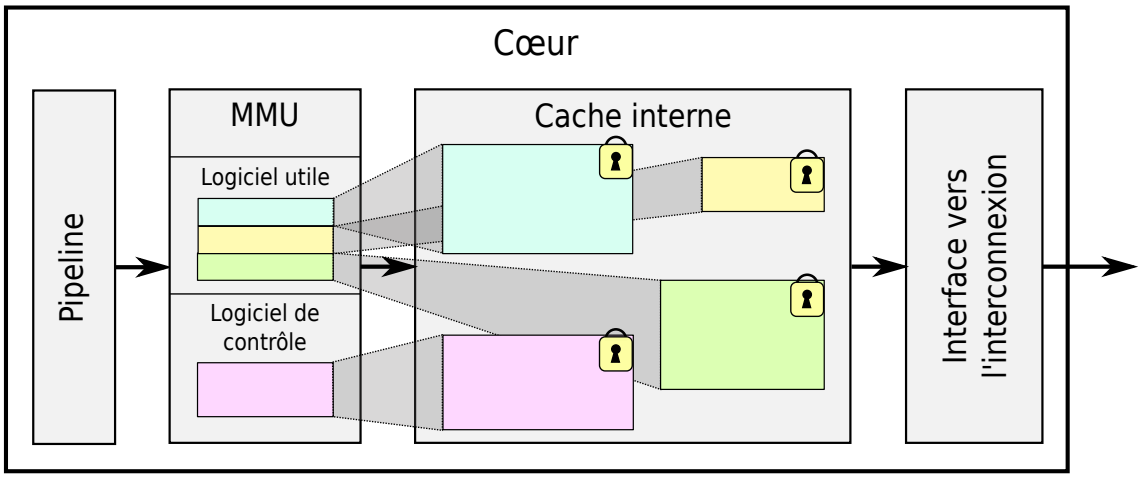


FIGURE 3.3 – Vue d'ensemble de la configuration d'un cœur dans l'approche par logiciel de contrôle de Jegu [54]

Un exemple d'implémentation de ce concept, appelé AER (Acquisition, Execution, Restitution) a été développé par Durrieu et al. [34] et expérimenté sur un Flight Management System déployé sur un processeur COTS. Cet exemple montre que les principes d'isolation temporelle des phases de communication permettent d'utiliser un processeur COTS dans un domaine d'usage où il est prédictible.

Approche par logiciel de contrôle

Jegu et al. proposent, dans un brevet déposé par Airbus [54], une approche reprenant les principes exposés ci-dessus. Ces principes sont implémentés dans un logiciel dédié, que nous appelons "logiciel de contrôle". Ce logiciel, qui est déployé sur tous les cœurs, va réaliser lui-même les phases de communication pour le compte du logiciel qu'il contrôle, que nous appellerons "logiciel utile" par la suite.

Le logiciel utile est conçu en une succession de phases de communication (fetch, flush) et d'exécution (execute). L'activité des phases de communication est décrite dans une table de données que le logiciel de contrôle lit pour réaliser les communications à proprement parler à des dates maîtrisées. Il s'occupe également de charger les données et instructions nécessaires à la phase d'exécution, de telle sorte que cette dernière ait lieu depuis une mémoire cache privée. Enfin, comme illustré sur la figure 3.3, il garantit que le logiciel utile ne peut émettre d'accès durant sa phase d'exécution en configurant la MMU pour ne rendre accessible que des données présentes dans le cache.

Ce brevet s'accompagne d'une publication [26] expliquant comment construire les tables de données pour le logiciel de contrôle à partir d'une application décrite comme une séquence de traitements, comme peut l'être un modèle synchrone. Cette approche semble la plus prometteuse parmi celles que nous avons recensées, dans la mesure où elle vise les processeurs COTS, puis se place dans un domaine d'usage qui reprend les

principes d'isolation temporels validés dans les approches à processeurs déterministes. Elle s'accompagne toutefois des limitations suivantes :

- Le logiciel doit être développé selon un modèle d'exécution contraint, puis analysé pour obtenir les tables de configuration du logiciel de contrôle. Cela empêche toute réutilisation du logiciel existant.
- À notre connaissance, aucune implémentation de cette approche n'a été réalisée. Nous ne connaissons pas l'impact sur le temps d'exécution du logiciel.

Nous revenons sur cette approche dans le chapitre 4, où nous définissons notre problématique de thèse.

3.4 Synthèse

Nous avons donné une vue d'ensemble des méthodes d'évaluation du WCET, et de leur application potentielle dans un contexte d'utilisation d'un processeur multi-cœurs. La principale difficulté, qui fait consensus dans les communautés industrielles et scientifiques, porte sur l'évaluation des perturbations engendrées sur une tâche par les tâches concurrentes exécutées sur les autres cœurs. Ces perturbations résultent de la présence d'interférences entre les activités électroniques concurrentes, qui surviennent lorsque les cœurs accèdent à des ressources matérielles communes. On cherche à borner ces perturbations en appliquant des pénalités d'interférences.

Le problème vient du fait qu'on ne connaît pas aujourd'hui de méthode pour évaluer une pénalité d'interférences sur un processeur COTS sans connaître l'intégralité des tâches embarquées, ou sans poser des restrictions sur l'utilisation des ressources communes par les cœurs. Or, les méthodes mono-cœurs n'abordent pas ces questions.

Nous nous sommes donc intéressés à des méthodes alternatives, qui étendent les méthodes mono-cœurs pour traiter la question des interférences. Ces approches visent soit à obtenir une pénalité d'interférences globale à partir du profil d'utilisation des ressources communes par toutes les tâches embarquées, soit à éliminer par construction les interférences en posant des restrictions sur l'usage des ressources communes par les cœurs.

Les approches par pénalité globale sont intéressantes, mais inapplicables dans notre contexte car elles invalident la propriété de partitionnement robuste, contrairement aux approches par élimination des interférences. Ces dernières mettent en œuvre des principes d'isolation temporelle des activités concurrentes émises par les différents cœurs. Dans la plupart des approches, ces principes sont implémentés dans le matériel, ce qui a abouti à des prototypes de processeurs adaptés pour le calcul du WCET. Ces approches ne sont à l'heure actuelle pas suivies par les fabricants de processeurs. D'autres approches, par logiciel déterministe, s'intéressent à la manière d'organiser le logiciel embarqué pour que l'activité qu'il génère s'insère dans des fenêtres temporelles disjointes. Ces approches sont au cœur de notre problématique de thèse, que nous développons dans le chapitre suivant.

Troisième partie

Démarche scientifique

Chapitre 4

Problématique

Sommaire

4.1 Problème général 53
4.2 Résumé des approches existantes 54
4.3 Problématique de thèse 57
4.4 Synthèse 58

4.1 Problème général

Le problème général que nous abordons dans cette thèse porte sur la capacité à évaluer des pires temps d'exécution (WCET) sur un ensemble de tâches déployées sur un processeur multi-cœurs. Cette maîtrise est une condition nécessaire pour effectuer par la suite des tests d'ordonnancement, et ainsi garantir que chaque tâche respecte ses échéances. Nous cherchons à appliquer des méthodes de calcul de WCET issues de l'état de l'art, qui ont été éprouvées dans un contexte mono-cœur, et qui sont reconnues comme sûres. La famille des méthodes par analyse statique, implémentées dans des outils comme Otawa, aiT, BoundT ou encore RapiTime [100] est un bon candidat. Les méthodes appartenant à cette famille sont connues pour être sûres, i.e. retournent des majorants des WCET réels.

Quand on cherche à calculer le WCET d'une tâche déployée sur un processeur multi-cœurs, la principale difficulté consiste à savoir à quel point cette tâche sera ralentie par des tâches exécutées sur les autres cœurs. Ces ralentissements surviennent lorsque le cœur effectue des accès vers des ressources matérielles partagées entre tous les cœurs. Lorsqu'une ressource n'a pas la capacité de traiter simultanément des accès provenant de différents cœurs, il s'ensuit des mises en attente de certains d'entre eux, que l'on qualifie d'***interférences***. La présence de ces interférences constitue le principal verrou pour le calcul du WCET du logiciel embarqué. Les méthodes par analyse statique supposent en effet que les accès effectués par le cœur à destination des ressources partagées ont une

durée bornée. En l'absence de borne sur leur impact, la présence d'interférences remet en cause cette hypothèse.

Nous abordons ce problème dans un contexte industriel, que nous avons décrit dans le chapitre 2. Nous nous intéressons à des systèmes de type IMA, qui font intervenir une plateforme composée d'un processeur multi-cœurs et d'un logiciel système, ainsi qu'un ensemble de partitions ARINC 653. Les partitions sont déployées dans une configuration de type AMP, ce qui signifie que chaque cœur exécute un ensemble de partitions qui lui sont affectées statiquement. Vis-à-vis de notre problème, cela se traduit par les contraintes suivantes :

(i) Le processeur sur lequel sera exécuté l'ensemble de tâches est un processeur COTS. Nous ne maîtrisons pas son design, et le niveau de détails de sa spécification peut être faible.

(ii) L'ensemble de tâches est déployé dans un environnement respectant la propriété de Partitionnement Robuste. En conséquence, la méthode de calcul de WCET doit être applicable modulairement sur chaque tâche, quelles que soient les tâches exécutées sur les autres cœurs.

(iii) Nous ne pouvons pas influer sur le design d'une application avionique, et seulement de manière marginale sur celui d'un système d'exploitation. En conséquence nous souhaitons être rétro-compatible vis-à-vis des applications existantes. Sur le système d'exploitation, un effort de portage similaire à celui d'un changement de processeur mono-cœur est acceptable.

À notre connaissance ce problème, abordé sous l'angle de ces contraintes, est ouvert. Aucune approche n'a levé le verrou posé par la présence d'interférences sans devoir relâcher une de ces contraintes. Nous développons ce point dans la section suivante.

4.2 Résumé des approches existantes

Les méthodes de calcul du WCET par analyse statique employées sur des processeurs mono-cœurs sont développées depuis une vingtaine d'années. Elles sont aujourd'hui considérées comme matures. Elles reposent sur des hypothèses qui s'appliquent au processeur ainsi qu'au logiciel embarqué. On suppose par exemple que le logiciel s'exécute en une suite finie d'instructions. Côté matériel, on demande que le traitement d'une séquence d'instructions dans le pipeline d'un cœur soit effectué en un temps borné. Il en va de même pour les opérations effectuées par ce cœur à destination des ressources communes. Le passage au processeurs multi-cœurs, avec la prise en compte des interférences, rend cette dernière hypothèse plus difficile à satisfaire.

L'approche la plus directe consiste à vouloir appliquer les mêmes méthodes de calcul de WCET sans altérer leurs hypothèses de validité. La prise en compte des interférences est effectuée en introduisant une pénalité sur la durée de chaque accès. Cette pénalité

s'obtient dans la pire configuration dans laquelle cet accès peut être effectué, en considérant toutes les possibilités d'accès concurrents. Comme développé dans la section 3.2.1, nous considérons que, avec nos contraintes, cette approche a peu de chances d'aboutir. Cela demanderait en effet de résoudre les verrous suivants :

1. Les processeurs COTS (cf. contrainte (i)) disposent de composants dont le niveau de documentation communiquée par le fabricant est faible, voire nul. Les pistes –empiriques ou analytiques– pour évaluer la pénalité d'interférences reposent donc sur des hypothèses qu'il est difficile de vérifier pour un équipementier faute d'informations, et sur lesquelles les fabricants de processeur ne s'engagent pas. Construire soi-même un modèle du processeur suffisamment détaillé pour étudier les interférences est donc une tâche incertaine.
2. L'architecture des processeurs COTS est complexe. Ces derniers sont optimisés pour avoir de bonnes performances moyennes au détriment de leur comportement en pire cas. Les quelques approches qui ont voulu analyser des modèles de processeurs ont mis en évidence une explosion combinatoire du nombre de situations à prendre en compte [48]. A supposer que l'on ait résolu le premier verrou et que l'on dispose d'un modèle du processeur dans lequel on puisse analyser les interférences, une telle analyse serait compliquée.
3. Certaines études ont montré des comportements de processeurs COTS qui mettent en évidence des cas d'interférences dont la pénalité est grande devant les temps d'accès aux ressources, par exemple d'un facteur 20 pour un processeur octo-cœurs [66]. A supposer que l'on ait résolu le deuxième verrou et que la pénalité d'interférence soit bornée, elle serait probablement grande devant les temps d'accès aux ressources et donc peu exploitable dans un calcul de WCET.

Du fait de la présence d'interférences que l'on n'arrive pas à analyser correctement, il ne semble pas réaliste de calculer un WCET sur un processeur multi-cœurs sans remettre en cause certaines pratiques existantes. On peut par exemple introduire des hypothèses supplémentaires pour contourner la complexité de l'analyse du comportement du processeur dans un espace de configuration peu contraint. Ces hypothèses peuvent restreindre le comportement du processeur, celui du logiciel, ou altérer la méthode de calcul de WCET. C'est l'objet de diverses approches, dont certaines sont arrivées à un stade de maturité avancé. Ces approches se classent dans les catégories suivantes :

Pénalité globale. La méthode de calcul de WCET est altérée pour effectuer dans un premier temps un calcul provisoire qui ne prend pas en compte les interférences. Dans un second temps, elle applique à chaque tâche une pénalité d'interférence globale calculée en prenant en compte le détail des opérations effectuées par tous les cœurs. Cela demande une connaissance fine de toutes les tâches exécutées sur le processeur et de l'activité que chacune génère. Cette approche relâche la contrainte (ii), car elle demande de connaître toutes les tâches déployées sur les différents cœurs, ce qui n'est pas compatible avec la propriété de Partitionnement Robuste.

Processeur déterministe. En altérant l'architecture du processeur, il est possible de garantir par construction l'absence d'interférences. Les approches de cette catégorie proposent des processeurs pour lesquels les accès aux ressources communes sont gérés par une politique d'arbitrage time-triggered, en général TDMA. Les activités concurrentes sont ainsi effectuées dans des fenêtres temporelles disjointes. Cette famille d'approche demande de relâcher la contrainte (i), car elle n'est pas supportée par les fabricants de processeurs COTS.

Logiciel déterministe. Les tâches embarquées sont organisées de telle sorte à assurer que les opérations qu'elles génèrent n'interfèrent pas. En général, l'exécution d'une telle tâche est organisée en phases de communication avec les ressources communes et en phases d'exécution interne qui ne génère pas d'activité dans les ressources communes. Les phases de communication ont lieu dans des fenêtres temporelles disjointes. Cette approche demande une connaissance fine des tâches embarquées afin de connaître précisément leur besoin en ressource. Elle demande également de concevoir le logiciel de manière adéquate pour qu'il s'organise selon ces phases de communication et d'exécution. Dans les approches existantes, la rétrocompatibilité vis-à-vis du logiciel existant n'est pas assurée. Cela demande de relâcher la contrainte (iii).

Ces approches éliminent les interférences, ou bornent leur impact sur le temps d'exécution du logiciel, mais elles s'appliquent dans le cas où on relâche une des contraintes de départ. Toutefois certaines sont arrivées à un stade de maturité avancé, et les concepts qu'elles portent sont de bons candidats pour résoudre notre problème, à condition qu'on puisse les adapter pour qu'ils supportent nos contraintes. Nous retenons en particulier les points suivants :

Stratégie d'utilisation. Il est possible de calculer des WCET sur des processeurs multi-cœurs, y compris COTS, quand ils sont utilisés selon certaines restrictions, que nous désignons sous le terme de *stratégie d'utilisation*. Par exemple, on peut interdire qu'un nombre trop important de cœurs accède à certaines ressources en même temps afin de ne pas les saturer. Dans les approches rencontrées, la stratégie d'utilisation est *time-triggered*, en général basée sur des politiques d'arbitrage TDMA. On élimine donc les interférences en isolant temporellement les activités concurrentes émises par les cœurs.

Stratégie de contrôle. La plateforme d'exécution, composée du processeur et du logiciel système, a l'obligation d'assurer la stratégie d'utilisation indépendamment du logiciel applicatif. Dans le cas contraire, la propriété de Partitionnement Robuste serait invalidée. Par exemple, on ne peut demander à une application de séquencer elle-même des phases d'exécution et des phases de communication. Il est donc nécessaire de définir une *stratégie de contrôle*, qui sera implémentée dans la plateforme.

La résolution de notre problème demande donc de choisir une famille d'approche qui dispose de ces deux stratégies. L'utilisation de processeurs multi-cœurs COTS (cf contrainte (i)) présente plusieurs difficultés, que nous détaillons ci-après et qui orientent notre problématique de thèse.

4.3 Problématique de thèse

Comme expliqué ci-dessus, la maîtrise des interférences sur les processeurs multi-cœurs, nécessaire pour calculer un WCET, demande de définir une stratégie d'utilisation ainsi qu'une stratégie de contrôle sur le processeur.

Les politiques d'arbitrage TDMA constituent une stratégie d'utilisation applicable sur des processeurs COTS. Cette solution n'est sûrement pas la plus efficace, dans la mesure où elle interdit tout parallélisme alors que les processeurs COTS peuvent effectuer un certain nombre de traitements en parallèle. Néanmoins elle résout le problème de la stratégie d'utilisation.

À l'inverse, le problème de la stratégie de contrôle est ouvert. En effet, on sait que le processeur ne permet pas d'offrir un niveau de contrôle adéquat sans une intervention du logiciel, que nous appelons *logiciel de contrôle*. On ne sait pas non plus comment concevoir un tel logiciel, ni comment ce dernier doit utiliser les ressources matérielles à sa disposition.

Nous nous intéressons à l'approche présentée par Jegu et al. dans un brevet d'Airbus [54], et résumée dans la section 3.3.3. Les auteurs présentent un concept de logiciel de contrôle, qui a vocation à être déployé sur chaque cœur du processeur et à exécuter une tâche par cœur en contrôlant ses accès aux ressources communes. Chaque tâche est découpée en une séquence de phases d'exécution interne et de phases de communication avec les ressources communes. Durant une phase d'exécution, les données et instructions nécessaires sont stockées dans une mémoire cache privée. Les phases de communication sont réalisées par le logiciel de contrôle à des dates définies statiquement, lors desquelles il effectue les transferts utiles et charge les données et instructions pour la phase d'exécution suivante. Ainsi, on a bien une stratégie de contrôle mise en œuvre par du logiciel, puisque les dates de tous les accès aux ressources communes sont maîtrisées.

Bien que cette piste soit un candidat sérieux pour résoudre notre problème, elle présente plusieurs limites, qui portent sur l'absence de rétrocompatibilité avec le logiciel existant (cf. contrainte (iii) page 54), ainsi que l'absence d'implémentation de ce logiciel de contrôle. Notre problématique de thèse porte donc sur l'opportunité de mettre en place une stratégie de contrôle du processeur mettant en œuvre un logiciel de contrôle, tout en assurant la rétrocompatibilité avec le logiciel applicatif existant. Nous nous intéressons à deux aspects du logiciel de contrôle, portant respectivement sur sa faisabilité sur un processeur donné, et sur l'existence d'une implémentation efficace dans un certain domaine d'applications.

4.4 Synthèse

La problématique que nous abordons dans cette thèse détaille un problème de haut niveau : la maîtrise des WCET d'un ensemble de tâches déployées sur un processeur multi-cœurs COTS. Cette maîtrise est actuellement impossible à cause d'interférences dans le processeur dont on ne connait ni les circonstances dans lesquelles elles se manifestent, ni leur impact sur le temps d'exécution du logiciel. Nous cherchons à améliorer une approche existante, quoique peu développée, à base de logiciel de contrôle, qui vise à éliminer les interférences. Cette approche adresse ce problème dans un contexte proche du nôtre, sans la contrainte de rétrocompatibilité du logiciel existant.

Notre problématique de thèse s'articule autour des deux questions suivantes :

- Comment concevoir et valider un logiciel de contrôle sur un processeur donné, tout en préservant la rétrocompatibilité du logiciel existant ?
- Pour un processeur donné, existe-t-il une implémentation efficace du logiciel de contrôle sur un domaine d'application représentatif de l'avionique ?

Nous présentons dans le chapitre suivant une vue d'ensemble de l'approche que nous adoptons pour traiter cette problématique.

Chapitre 5

Approche

Sommaire

5.1 Définition et existence du logiciel de contrôle 59
5.2 Prototypage et étude de l'efficacité du logiciel de contrôle 61
5.2.1 Vue d'ensemble du prototype 61
5.2.2 Efficacité du logiciel de contrôle 62

Nous avons présenté dans le chapitre précédent notre problématique de thèse, qui porte sur les points suivants :

- la définition et l'étude de l'existence d'un logiciel de contrôle sur un processeur donné.
- l'existence d'une implémentation efficace du logiciel de contrôle sur un certain type d'applications.

Nous distinguons dans la suite du manuscrit le ***logiciel de contrôle*** du ***logiciel utile***, qui fait l'objet du contrôle. Le logiciel utile sera selon les utilisations du logiciel applicatif, mais également un système d'exploitation.

Cette section présente l'approche que nous avons suivie dans cette thèse.

5.1 Définition et existence du logiciel de contrôle

Le logiciel de contrôle est en charge de mettre en œuvre une stratégie d'utilisation du processeur basée sur une politique d'arbitrage TDMA. À ce titre, son rôle majeur est d'effectuer lui-même, à des dates maîtrisées, toutes les opérations sur les ressources communes dont le logiciel utile a besoin pour s'exécuter correctement. Cela concerne les données et instructions du logiciel utile, qui sont stockées dans la mémoire principale, mais également les lectures et écritures sur les entrées/sorties.

De plus, nous attendons du logiciel de contrôle qu'il soit rétro-compatible vis-à-vis du logiciel avionique existant.

Pour définir le logiciel de contrôle, nous effectuons une analogie avec le cadre de virtualisation de Popek et Goldberg [69]. Dans ce cadre, les auteurs introduisent un logiciel appelé gestionnaire de machines virtuelles, ou hyperviseur dans la littérature plus récente. Cet hyperviseur est une forme de logiciel de contrôle, dans le sens où il se substitue au logiciel utile –appelé *logiciel invité*– pour effectuer toutes les opérations d'allocation de ressources matérielles. Cela concerne par exemple l'allocation d'espace mémoire, ou encore l'allocation de droits d'accès à un périphérique. En se substituant au logiciel utile, il peut effectuer un certain nombre d'opérations, par exemple vérifier que ce dernier a le droit de s'allouer ces ressources.

Popek et Goldberg définissent un hyperviseur comme un logiciel vérifiant les propriétés suivantes :

Contrôle de ressources. Le logiciel de contrôle intervient dans la totalité des cas où le logiciel invité tente de s'allouer des ressources. Il est en mesure d'effectuer correctement les opérations d'allocation pour le compte du logiciel invité, de telle sorte que ce dernier puisse s'exécuter correctement.

Équivalence. À l'exception de son temps d'exécution, le logiciel invité ne peut distinguer une exécution sous contrôle de l'hyperviseur d'une exécution sans hyperviseur, au cours de laquelle il s'alloue lui-même les ressources matérielles.

Efficacité. Toute opération neutre, i.e. non sensible vis-à-vis de l'allocation des ressources ne fait pas l'objet d'un contrôle par l'hyperviseur.

Dans notre cas, le contrôle ne porte pas sur l'allocation de ressources mais sur les accès à ces mêmes ressources. Nous définissons notre logiciel de contrôle sur la base des deux premières propriétés, à savoir le contrôle des ressources et l'équivalence. La notion d'efficacité selon Popek et Goldberg est trop restrictive pour notre contexte, mais nous en adoptons une définition proche. Mis à part ces détails, les propriétés que nous attendons de notre logiciel de contrôle sont les mêmes. De plus, la propriété d'équivalence assure que le logiciel de contrôle préserve la rétrocompatibilité sur le logiciel existant. L'analogie entre les deux approches est donc pertinente.

Popek et Goldberg introduisent un théorème de virtualisation, qui conditionne l'existance d'un hyperviseur vérifiant les trois propriétés ci-dessus. Nous cherchons à appliquer ce théorème, ou une version proche, pour notre logiciel de contrôle.

Dans notre contribution, développée dans le chapitre 6, nous montrons que le théorème de virtualisation de Popek et Goldberg ne permet pas d'assurer l'existence d'un logiciel de contrôle sur les accès aux ressources communes. Nous proposons à cet effet une généralisation de ce théorème, que nous appelons *théorème de contrôlabilité*. Le reste de cette contribution porte sur l'application du théorème de contrôlabilité, dans un premier temps sur un modèle simple de processeur, puis sur un processeur COTS. Il s'agit du P5040, de la gamme QorIQ commercialisée par Freescale, gamme à laquelle l'industrie avionique s'intéresse.

Cette contribution se termine par la spécification d'un logiciel de contrôle, dont nous étudions l'efficacité dans la seconde partie de la contribution.

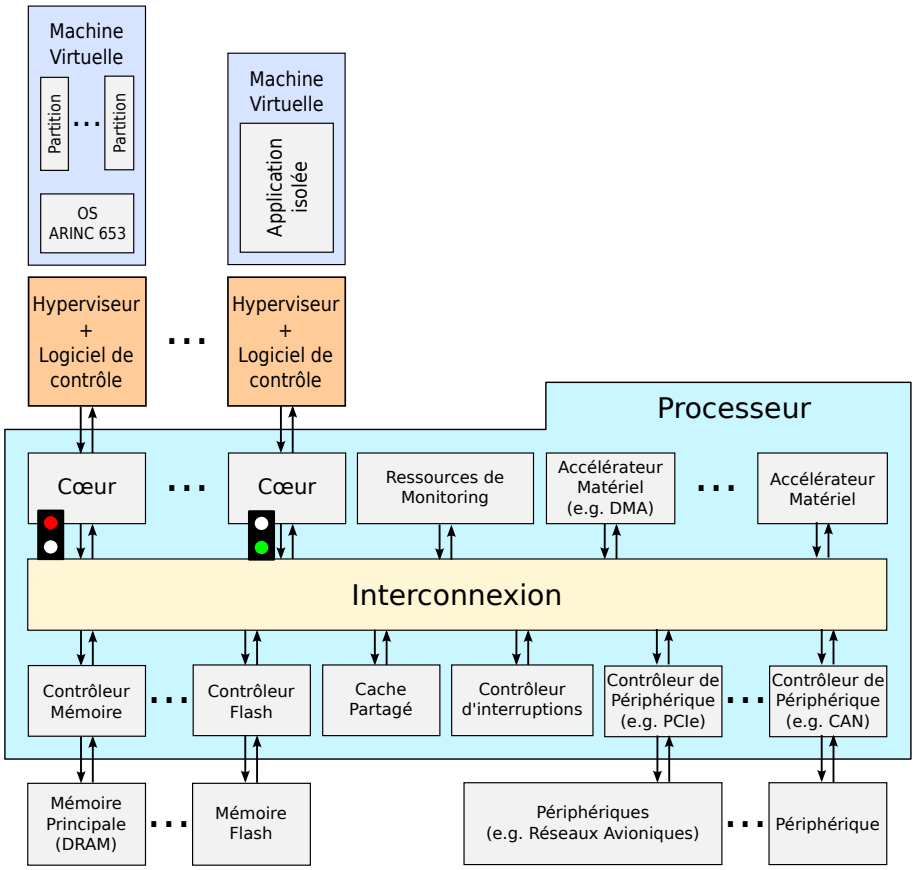


FIGURE 5.1 – Vue d'ensemble du déploiement du prototype sur un processeur générique

5.2 Prototypage et étude de l'efficacité du logiciel de contrôle

L'application du théorème de contrôlabilité nous amène à spécifier les mécanismes à implémenter dans un logiciel de contrôle. Cette spécification a abouti à la réalisation d'un prototype, présenté sur la figure 5.1.

5.2.1 Vue d'ensemble du prototype

Notre prototype de logiciel de contrôle est implémenté au sein d'un petit hyperviseur, que nous avons réalisé. Ce choix vient de plusieurs facteurs, mais n'est toutefois pas corrélé avec l'utilisation du cadre de virtualisation de Popek et Goldberg, que nous avons abordé précédemment. Il aurait par exemple été possible de l'intégrer directement dans les couches basses d'un système d'exploitation.

D'une part nous souhaitons déployer sur chaque cœur un système ARINC 653, qui sera composé d'un système d'exploitation et d'applications. Intégrer le logiciel de contrôle

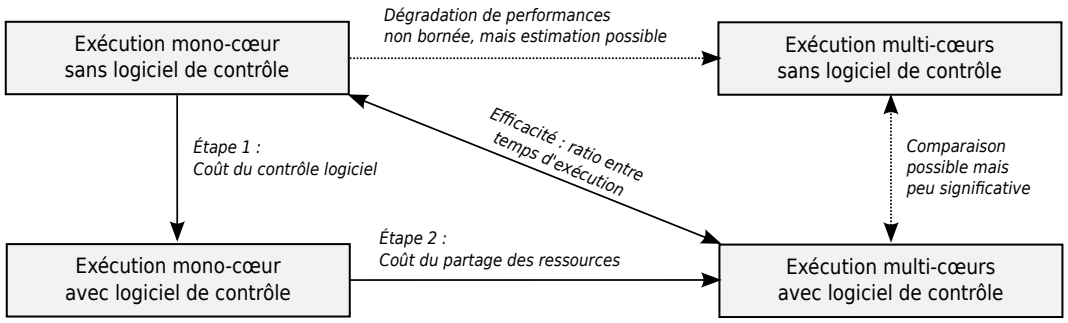


FIGURE 5.2 – Vue d'ensemble de la démarche d'évaluation de l'efficacité du prototype

dans un hyperviseur va nous permettre de contrôler des systèmes d'exploitation de la même manière que des applications, i.e. sans à avoir à les modifier. D'autre part, cela réduit les contraintes de déploiement, dans la mesure où le logiciel de contrôle et son conteneur sont chargés de manière permanente dans les caches de chaque cœur. À ce titre, disposer d'un hyperviseur minimaliste nous permet de limiter l'espace occupé dans le cache et donc de maximiser celui qui est disponible pour le logiciel utile.

5.2.2 Efficacité du logiciel de contrôle

Une fois déployé, le logiciel de contrôle va altérer le temps d'exécution –en moyenne et en pire cas– du logiciel utile. On dit que le logiciel de contrôle est ***efficace*** si le ratio du temps d'exécution avec et sans logiciel de contrôle est en dessous d'une certaine valeur. Un ratio égal à deux sur un processeur à quatre cœurs nous semble un bon compromis entre la dégradation individuelle du temps d'exécution de chaque tâche et le gain apporté par le logiciel de contrôle en terme de prédictibilité sur le processeur, combiné à la possibilité d'utiliser plusieurs cœurs.

Comme illustré sur la figure 5.2, on peut étudier l'efficacité du logiciel de contrôle en comparant les temps d'exécution dans des configurations mono-cœurs et/ou multi-cœurs. Nous décomposons cette étude en deux étapes, portant respectivement sur :

- Le coût du logiciel de contrôle. Nous évaluons le coût minimal lié à l'utilisation d'un logiciel de contrôle. Pour cela, nous considérons une configuration mono-cœur sans logiciel de contrôle, que nous comparons à une configuration où un cœur dispose de l'intégralité des plages d'accès aux ressources communes.
- Le coût du partage des ressources. Nous évaluons ici la sensibilité d'une application au fait de concentrer ses accès aux ressources communes dans des fenêtres temporelles dont la taille et la période varient.

Ce chapitre résume notre approche vis-à-vis de la problématique que nous abordons dans cette thèse. La partie suivante développe ces éléments dans quatre chapitres portant respectivement sur la stratégie de contrôle, son instanciation dans le cadre d'un cas d'étude basé sur un processeur COTS, l'architecture du logiciel de contrôle dans ce cas d'étude et l'étude d'efficacité du logiciel de contrôle.

Quatrième partie

Contributions

Chapitre 6

Stratégie de contrôle du processeur

Sommaire

6.1 Cadre de virtualisation de Popek et Goldberg 66
6.1.1 Définition d'un hyperviseur 66
6.1.2 Théorème de virtualisation de Popek et Goldberg 68
6.2 Cadre analogue pour le logiciel de contrôle 71
6.2.1 Définition du logiciel de contrôle 71
6.2.2 Théorème de contrôlabilité 74
6.3 Application du théorème de contrôlabilité 76
6.3.1 Vue d'ensemble de la démarche 76
6.3.2 Modèle de cœur considéré 78
6.3.3 Définitions des évènements sensibles 79
6.3.4 Classification des évènements sensibles 81
6.3.5 Construction d'un invariant 83
6.3.6 Construction des séquences de contrôle 87
6.4 Synthèse 90

Nous avons expliqué précédemment qu'il est possible de rendre un processeur multi-cœurs prédictible en restreignant l'utilisation que l'on en fait selon une *stratégie d'utilisation*. Nous reprenons la stratégie de partage TDMA, qui est répandue dans les solutions rencontrées dans l'état de l'art. Nous supposons donc que chaque cœur dispose d'une fenêtre temporelle dans laquelle il peut accéder aux ressources communes de manière exclusive. Pour mettre en place cette stratégie d'utilisation, nous introduisons la notion de *stratégie de contrôle*, définie ci-après.

Définition 6.1 (Stratégie de contrôle)**.** *La stratégie de contrôle se traduit par l'ensemble des mécanismes mis en œuvre, au niveau matériel et/ou logiciel, pour assurer le respect de la stratégie d'utilisation.*

Nous cherchons à implémenter la stratégie de contrôle dans un logiciel dédié, que nous appelons ***logiciel de contrôle***, par opposition au ***logiciel utile***, qui subit ce contrôle.

Pour le logiciel de contrôle, implémenter une stratégie TDMA signifie qu'il faut maîtriser les dates d'émission de chaque accès des cœurs aux ressources partagées. Dans ce chapitre, nous étudions la question de l'existence d'un tel logiciel de contrôle.

Nous développons le problème de l'existence du logiciel de contrôle à travers une analogie avec le cadre de virtualisation de Popek et Goldberg [69]. Ces derniers ont défini le concept de *gestionnaire de machines virtuelles*, souvent appelé *hyperviseur* dans la littérature plus récente. Popek et Goldberg ont défini les propriétés qu'ils attendent d'un hyperviseur, qui est une forme de logiciel de contrôle, le contrôle portant sur l'allocation des ressources à des machines virtuelles, qui sont l'analogue du logiciel utile. Ils présentent ensuite un théorème d'existence d'un hyperviseur vérifiant ces propriétés. Les hypothèses d'application de ce théorème portent sur des propriétés du processeur.

Nous développons dans la section 6.1 l'analogie avec le cadre de virtualisation de Popek et Goldberg. Nous définissons sur la base de cette analogie notre concept de logiciel de contrôle dans la section 6.2, puis nous montrons que le théorème de virtualisation ne permet pas de garantir l'existence du logiciel de contrôle. Nous proposons par la suite une généralisation de ce théorème, appelée "théorème de contrôlabilité". La section 6.3 détaille une méthode permettant l'application du théorème de contrôlabilité.

6.1 Cadre de virtualisation de Popek et Goldberg

Nous résumons dans cette section le cadre de virtualisation de Popek et Goldberg [69], dans lequel nous disposons d'un théorème d'existence d'un hyperviseur sur un processeur donné. Cette approche fait toujours l'objet de travaux, par exemple sur le jeu d'instruction ARM [68]. L'objectif est de montrer que l'analogie est correcte, et que l'on peut donc étudier l'existence de notre logiciel de contrôle dans ce cadre.

6.1.1 Définition d'un hyperviseur

Popek et Goldberg définissent un hyperviseur comme une forme de logiciel de contrôle. Ce contrôle s'applique sur du logiciel dit *logiciel invité*, exécuté dans un environnement appelé *machine virtuelle*, qui simule le comportement d'une machine réelle. Le contrôle porte sur l'allocation de ressources matérielles aux machines virtuelles. Il revient donc à l'hyperviseur de s'exécuter lorsque le logiciel invité tente d'effectuer une opération de nature à s'allouer des ressources matérielles.

Selon les auteurs, l'allocation de ressources matérielles au logiciel invité a lieu lors de l'exécution de certaines instructions sur le processeur. Ces instructions sont dites *sensibles*[29], par opposition aux instructions *neutres*[30]. L'hyperviseur doit empêcher le logiciel invité d'exécuter lui-même toute instruction sensible.

Pour Popek et Goldberg, les propriétés attendues d'un hyperviseur sont les suivantes :

29. sensitive
30. innocuous

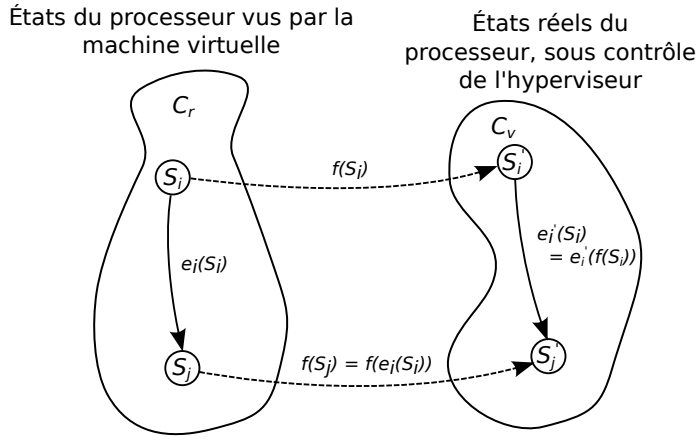


FIGURE 6.1 – Illustration de la propriété d'équivalence [69]

Contrôle des ressources. Le logiciel invité ne peut avoir accès autrement que par émulation des instructions sensibles par l'hyperviseur à des moyens d'allocation de ressources matérielles.

Équivalence. À l'exception d'une altération de son temps d'exécution, le logiciel invité ne peut distinguer une exécution dans une machine virtuelle d'une exécution sur une machine réelle. En d'autres termes, l'hyperviseur doit donner au logiciel invité l'illusion qu'il exécute lui-même les instructions sensibles.

Les auteurs formalisent cette notion à travers la définition d'une fonction de virtualisation [31] correspondant à la différence entre l'état du processeur que verrait le logiciel invité s'il était exécuté seul, et l'état du processeur dans lequel le logiciel invité est exécuté sous contrôle de l'hyperviseur. Considérons les notations suivantes :

- $S_i \in C_r$ l'espace d'états du processeur tel que le logiciel invité croit le voir, i.e. tel qu'il le verrait s'il n'y avait pas d'hyperviseur.
- $S_i' \in C_v$ l'espace d'états du processeur en présence d'un hyperviseur.
- $e_i \in I$ l'instruction e_i appartenant au jeu d'instructions I, et $e_i(S_i)$ l'état du processeur dans un état initial S_i après exécution de l'instruction e_i.

La fonction de virtualisation, est notée $f : C_r \longmapsto C_v : f(S_i) = S_i'$. La propriété d'équivalence, illustrée sur la figure 6.1 s'écrit alors :

$$\forall S_i \in C_r, \forall e_i \in I, \exists e_i' \in I, f(e_i(S_i)) = e_i'(f(S_i))$$

Efficacité. L'hyperviseur n'intervient pas lorsque le logiciel invité exécute des instructions neutres. Dans le cadre de Popek et Goldberg, cette propriété est définie au sens strict.

31. Virtual Machine Map

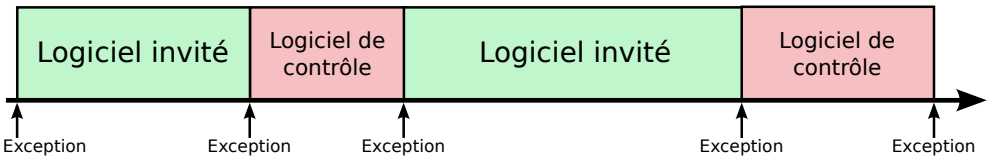


FIGURE 6.2 – séquences d'exécution du logiciel de contrôle et du logiciel invité

Les propriétés de contrôle des ressources, d'équivalence et d'efficacité sont les mêmes que celles que nous attendons de notre logiciel de contrôle. L'analogie est donc pertinente. Nous présentons dans la section suivante le théorème de virtualisation de Popek et Goldberg, qui donne une condition suffisante pour l'existence d'un hyperviseur vérifiant ces propriétés. Cette condition se traduit en propriétés à vérifier sur le processeur.

6.1.2 Théorème de virtualisation de Popek et Goldberg

Dans le cadre de virtualisation de Popek et Goldberg, nous disposons d'un ensemble de propriétés attendues d'un hyperviseur, qui s'appliquent également à notre logiciel de contrôle. Ces propriétés portent sur le contrôle d'opérations interdites pour le logiciel invité, appelées instructions sensibles. Comme illustré sur la figure 6.2, l'exécution du logiciel invité est interrompue par des phases d'exécution de l'hyperviseur. Pour empêcher le logiciel invité d'exécuter des instructions sensibles, l'hyperviseur doit donc se reposer sur la configuration du processeur.

À cet effet, Popek et Goldberg supposent que le processeur dispose de deux niveaux de privilèges : superviseur et utilisateur. Au niveau superviseur, toutes les instructions sont exécutées normalement. Au niveau utilisateur, certaines instructions, dites *privilégiées*, ne sont pas autorisées. Si le logiciel tente malgré tout d'en exécuter une, le processeur lève une exception appelée *piège*[32]. Cette exception s'accompagne d'un changement de privilège vers le niveau superviseur, et est dirigée vers l'hyperviseur. L'ensemble des instructions privilégiées est une propriété du processeur. Nous n'avons pas, ou à la marge, la possibilité d'influer sur cet ensemble.

Popek et Goldberg introduisent le théorème suivant, qui est une condition suffisante pour l'existence d'un hyperviseur :

Théorème 6.1 (Théorème de virtualisation [69])**.** *Un hyperviseur vérifiant les propriétés d'efficacité, de contrôle des ressources et d'équivalence peut être implémenté si chaque instruction sensible est privilégiée.*

Démonstration :

Nous reprenons ici les grandes étapes de la preuve du théorème de virtualisation, exposée dans l'article original. Les auteurs admettent que l'hypothèse suivante est vraie :

32. trap

Hypothèse 6.1 (Jeu d'instruction émulable)**.** *Pour chaque instruction privilégiée, il existe une séquence d'instructions qui, lorsqu'elles sont exécutées au niveau de privilège superviseur, reproduisent le comportement de ladite instruction, en assurant les propriétés d'équivalence et de contrôle des ressources.*

Autrement dit, au terme de cette séquence d'instruction, l'altération de l'état du processeur (registres et mémoire) provoquée par une instruction privilégiée peut être reproduite explicitement par une séquence d'instructions. Dans le contexte dans lequel se placent les auteurs, on peut légitimement admettre cette hypothèse, pour deux raisons :

- D'une part la description d'un jeu d'instructions fournit les informations suffisantes pour connaître exactement l'état du processeur après exécution d'une instruction. On peut prendre pour exemple les jeux d'instructions PowerPC [73] et ARM [16].
- les vecteurs d'exception disposent des structures de données nécessaires pour manipuler directement l'état du processeur, c'est-à-dire l'état des registres avant exécution de l'instruction privilégiée. Ils sont également capables de restaurer un état altéré par du logiciel. Il est donc possible de reproduire l'état du processeur après exécution de l'instruction, mais aussi d'altérer la manière dont les ressources sont effectivement allouées, tout en donnant l'illusion au logiciel invité qu'il a correctement exécuté son instruction.
  On vérifie ainsi la propriété de contrôle des ressources, ainsi que la propriété d'équivalence pour les instructions privilégiées.

La partie principale de la preuve du théorème de virtualisation vise à montrer la propriété d'équivalence, les propriétés de contrôle de ressources et d'efficacité étant supposées par les auteurs comme induites par la définition d'une machine virtuelle. Pour montrer la propriété d'équivalence pour toute séquence finie d'instructions, les auteurs raisonnent par récurrence.

La condition initiale consiste à montrer la propriété d'équivalence pour toute instruction. Les auteurs distinguent le cas d'une instruction neutre [69, Lemme 1] de celui d'une instruction sensible [69, Lemme 2] :

- Dans le cas d'une instruction neutre, si cette instruction n'est pas privilégiée, elle n'est pas interrompue et la propriété d'équivalence est implicitement vérifiée. Si elle est privilégiée, la propriété d'équivalence découle de l'hypothèse d'émulation du jeu d'instructions.
- Dans le cas d'une instruction sensible, cette instruction est par définition privilégiée. La propriété d'équivalence découle également de l'hypothèse d'émulation du jeu d'instructions.

En supposant la condition initiale vraie, les auteurs montrent que si une séquence d'instructions vérifie la propriété d'équivalence, cette séquence augmentée d'une instruction la vérifie toujours [69, Lemme 3]. Reprenons les notations des auteurs, à savoir S_i l'état du processeur vu du logiciel invité, $S_j = f(S_i)$ l'état réel du processeur, e_i une séquence d'instructions vérifiant la propriété d'équivalence, t une instruction simple, qui par hypothèse vérifie la propriété d'équivalence, e'_i et t' les séquences d'instructions équivalentes, et te_i et $t'e'_i$ la concaténation de t et t' avec les séquences e_i et e'_i.

Nous avons les relations $f(t(S_i)) = t'(f(S_i))$ et $f(e_i(S_i)) = e'_i(f(S_i))$. Cela implique les égalités suivante :

$$\begin{aligned} f(te_i(S_i)) &= f(t(e_i(S_i))) \\ &= t'(f(e_i(S_i))) \\ &= t'(e'_i(f(S_i))) \\ &= t'e'_i(f(S_i)) \end{aligned}$$

Ainsi la séquence te_i a pour séquence équivalente $t'e'_i$. Elle vérifie ainsi la propriété d'équivalence, ce qui prouve l'hypothèse de récurrence. □

Discussion : On pourra noter que la notion d'efficacité est revendiquée dans le théorème, mais n'est pas abordée dans la preuve. Les auteurs admettent cette propriété dans la définition d'une machine virtuelle. Au sens strict, cette propriété signifie qu'une instruction neutre n'est pas privilégiée, car dans le cas inverse, l'hyperviseur serait contraint de l'émuler à l'identique. Or le théorème de virtualisation ne suppose pas cette assertion. C'est ce qui a motivé la définition de Smith et Nair [90], présentée précédemment, qui considèrent l'efficacité non pas comme une exigence mais comme une bonne propriété, ce qui permet d'appliquer le théorème de virtualisation dans un plus grand nombre de cas. En pratique, sur les processeurs actuels, le problème se pose peu. Les instructions sensibles sont connues des fabricants. Ce sont les mêmes pour un système d'exploitation que pour un hyperviseur, seule la manière de les émuler est différente. Par conséquent, seules celles-ci sont privilégiées. La définition exacte de Popek et Goldberg coïncide avec la définition moins restrictive de Smith et Nair, avec éventuellement quelques écarts mineurs.

Dans notre cas, le type de contrôle est différent, et n'a pas été prévu par le fabricant de processeurs. Par conséquent il faut s'attendre à ce que la propriété d'efficacité, telle que définie par Popek et Goldberg, ne soit pas vérifiée. Autrement dit, le logiciel de contrôle interviendra à certains moments dans des situations où le logiciel utile n'aurait pas émis d'accès. Ce type de situation est indésirable, mais pas rédhibitoire pour que le logiciel de contrôle fonctionne correctement. Il est donc pertinent de considérer la notion d'efficacité selon la définition de Smith et Nair. 8.

Synthèse : Nous avons résumé le cadre dans lequel Popek et Goldberg ont défini la notion d'hyperviseur, qui est une forme de logiciel de contrôle portant sur l'allocation de ressources matérielles à des machines virtuelles. Nous effectuons une analogie entre notre logiciel de contrôle, qui porte sur les accès à ces ressources, et la notion d'hyperviseur. Cette analogie se justifie par la similarité des propriétés attendues des logiciels de contrôle, à savoir le fait de contrôler l'intégralité des opérations sensibles, tout en étant aussi peu intrusif que possible.

L'analogie avec les concepts manipulés est pertinente. Dans la section suivante, nous nous appuyons sur ces notions pour définir un cadre dans lequel nous précisons la notion d'évènement sensible, qui fait l'objet du contrôle, et les propriétés attendues du logiciel de

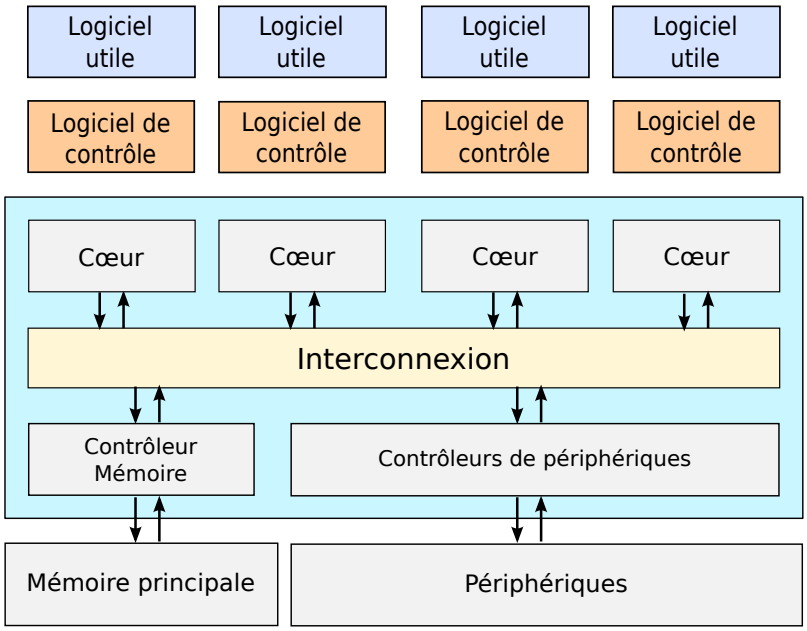


FIGURE 6.3 – Déploiement du logiciel de contrôle

contrôle. Nous abordons ensuite la question de l'adaptation du théorème de virtualisation de Popek et Goldberg à notre cadre.

6.2 Cadre analogue pour le logiciel de contrôle

Nous reprenons dans cette section certaines notions développées dans la section précédente, en les adaptant aux spécificités de notre logiciel de contrôle. Dans un premier temps, nous raffinons la définition d'évènement sensible –analogue des instructions sensibles chez Popek et Goldberg– ainsi que les propriétés attendues de notre logiciel de contrôle. Nous abordons ensuite la question de la validité du théorème de virtualisation dans notre contexte.

6.2.1 Définition du logiciel de contrôle

Comme décrit dans notre approche (cf chapitre 5), nous mettons en œuvre un logiciel qui contrôle les accès des cœurs aux ressources communes sur un processeur multi-cœurs. Conformément à la stratégie d'utilisation, les cœurs disposent de fenêtres temporelles dans lesquelles ils ont le droit d'accéder aux ressources. Le logiciel de contrôle doit forcer le logiciel utile à respecter ces fenêtres.

L'hyperviseur de Popek et Goldberg porte sur le contrôle d'instructions sensibles, qui peuvent amener le logiciel invité à s'octroyer le niveau de privilège superviseur et/ou à s'allouer certaines ressources matérielles. Notre logiciel vise à contrôler les dates d'accès

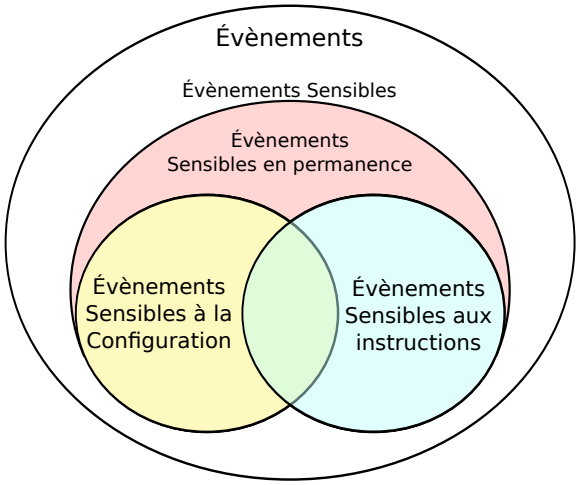


FIGURE 6.4 – Classification des évènements sensibles

TABLE 6.1 – Terminologies analogues

Cadre de Popek et Goldberg [69]	**Logiciel de contrôle**
Hyperviseur, ou Gestionnaire de Machines Virtuelles	Logiciel de contrôle
Logiciel invité	Logiciel utile
Instructions sensibles	Évènements sensibles
Instructions neutres	Évènements neutres

du logiciel utile à ces ressources. Certains accès ont lieu lors de l'exécution de certaines instructions. D'autres accès peuvent avoir lieu sur des évènements matériels autres que l'exécution d'instructions, par exemple les chargements (fetch) d'instructions effectués par le cœur. Dans les deux cas, nous parlons *d'évènements sensibles*, comme étant l'analogue des instructions sensibles de Popek et Goldberg.

Définition 6.2 (Évènement sensible)**.** *Un évènement matériel est sensible s'il est suivi de l'émission d'un ou plusieursaccès aux ressources communes.*

Parmi les évènements sensibles que l'on peut rencontrer sur un processeur, nous retenons les cas particuliers suivants, illustrés sur la figure 6.4 :

Sensibilité à une instruction. Un évènement est sensible à une instruction si et seulement s'il correspond à l'exécution d'une instruction, au sens électronique, c'est-à-dire lors de son passage dans l'étage correspondant du pipeline. Lorsqu'un évènement est sensible à une instruction, on parle également d'instruction sensible.

Sensibilité à la configuration. Un évènement est sensible à la configuration si et seulement s'il existe un état du cœur dans lequel cet évènement est sensible, et un autre

état dans lequel cet évènement est neutre. Le chargement d'une donnée stockée dans une mémoire est un exemple d'évènement sensible à la configuration, selon la présence –ou non– de cette donnée dans un cache local.

Sensibilité permanente. Un évènement est sensible en permanence s'il n'appartient à aucune des deux catégories précédentes.

Les propriétés de contrôle des ressources et d'équivalence que nous attendons du logiciel de contrôle sont les mêmes que celles définies dans le cadre de virtualisation de Popek et Goldberg. Ainsi le logiciel utile ne doit avoir aucun moyen d'accéder aux ressources communes sans passer par le logiciel de contrôle, et l'effet du logiciel de contrôle doit être invisible pour le logiciel utile, à l'exception d'une altération de son temps d'exécution. La propriété d'équivalence assure entre autre la rétrocompatibilité vis-à-vis du logiciel existant, ce qui était un de nos objectifs.

Comme expliqué dans la section précédente, nous retenons la définition de la propriété d'efficacité selon Smith et Nair [90], moins restrictive que celle de Popek et Goldberg. Dans notre cas, elle signifie que le logiciel de contrôle intervient le moins possible dans les cas où le logiciel utile n'aurait pas accédé aux ressources communes. Cette propriété fait l'objet d'un développement spécifique dans le chapitre 8.

Définition 6.3 (Évènement contrôlable)**.** *Un évènement est dit contrôlable lorsque son occurrence peut être systématiquement évitée grâce à la levée d'une exception dirigée vers le logiciel de contrôle.*

Définition 6.4 (Évènement partiellement contrôlable)**.** *Un évènement est dit* partiellement contrôlable *lorsqu'il existe un état du cœur où cet évènement déclenche une exception dirigée vers le logiciel de contrôle, et un état du cœur où aucune exception n'est levée.*

Lorsqu'un évènement sensible est contrôlable, au sens de la définition ci-dessus, l'exception levée par le cœur donne la possibilité au logiciel de contrôle de reproduire l'effet de cet évènement conformément à la propriété de contrôle des ressources. Pour cela, il embarque des séquences de contrôle, telles que définies ci-après.

Définition 6.5 (Séquence de contrôle)**.** *Une séquence de contrôle est une suite d'instructions neutres exécutées par le logiciel de contrôle qui a pour but de reproduire l'effet d'un évènement intercepté par le logiciel de contrôle. Nous distinguons deux types de séquences de contrôle :*

Émulation *Une séquence de contrôle par émulation s'applique à un évènement sensible à une instruction. Elle cherchera à reproduire les effets d'une instruction sensible sans que celle-ci soit réellement exécutée. L'exécution du logiciel utile reprend après l'instruction émulée. On retrouve ce type de séquence de contrôle chez Popek et Goldberg.*

Neutralisation *Une séquence de contrôle par neutralisation s'applique sur un évènement sensible à la configuration. Elle visera à placer le processeur dans un état où cet évènement n'est plus sensible, pour ensuite le faire rejouer par le logiciel utile.*

Pour un évènement sensible donné, on parlera de stratégie de contrôle par émulation ou par neutralisation selon le type de séquence de contrôle utilisé. De la même manière, on dira qu'un évènement est contrôlable par émulation et/ou par neutralisation selon le type de séquence de contrôle disponible.

Pour respecter la propriété de contrôle des ressources, une séquence de contrôle devra notamment garantir que les accès pour le compte du logiciel utile seront émis dans des fenêtres temporelles maîtrisées. Cela suppose que chaque cœur ait un moyen de mesurer une date de référence, sans pour autant accéder aux ressources communes. Nous introduisons à cet effet la propriété de synchronisabilité.

Définition 6.6 (Synchronisabilité)**.** *Un cœur est synchronisable s'il existe une séquence d'instructions neutres qui retourne la valeur d'une horloge de référence, commune entre tous les cœurs.*

La nature du contrôle d'accès apporte également certaines contraintes au niveau du déploiement de ce logiciel, qui sont spécifiques à notre cadre, et n'ont donc pas été abordées par Popek et Goldberg. Comme illustré sur la figure 6.3, le contrôle est effectué au niveau des cœurs par un logiciel stocké dans une mémoire locale, par exemple un cache. Les données nécessaires à la configuration du logiciel de contrôle sont répliquées. Ainsi, l'instance locale du logiciel de contrôle sur chaque cœur est autonome pour contrôler l'activité de ce cœur. Autrement dit, le logiciel de contrôle n'émet pas d'accès autres que ceux, contrôlés, qu'il effectue à la place du logiciel utile à destination des ressources communes. Cette particularité de notre logiciel de contrôle a des conséquences sur la validité du théorème de virtualisation, que nous développons dans la section suivante.

6.2.2 Théorème de contrôlabilité

Le théorème de virtualisation de Popek et Goldberg donne une condition suffisante pour l'existence d'un hyperviseur vérifiant les propriétés d'efficacité, de contrôle des ressources et d'équivalence. Cette condition consiste à vérifier que toute tentative du logiciel invité d'exécuter une instruction sensible, qui pourrait lui permettre de s'allouer des ressources, génère une exception dirigée vers l'hyperviseur.

On peut définir la propriété analogue, à savoir que toute tentative du logiciel utile d'effectuer une opération impliquant un évènement sensible sur le cœur déclenche une exception dirigée vers le logiciel de contrôle. On pourrait conjecturer que cette condition est suffisante pour assurer l'existence d'un logiciel de contrôle. Ce n'est pas le cas, à cause d'une différence entre le cadre de Popek et Goldberg et le nôtre sur la notion d'évènement (resp. d'instruction) sensible.

De plus, dans le cadre de Popek et Goldberg, une instruction sensible n'a de sens que lorsqu'elle est exécutée au niveau de privilège utilisateur, car elle peut altérer le niveau de privilège ou allouer des ressources au logiciel invité. Dans notre cas, un évènement est sensible lorsqu'il génère un accès, indépendamment du niveau de privilège du cœur. Ainsi, le logiciel de contrôle pourrait au cours de son exécution générer des accès non

maîtrisés à destination des ressources communes. Nous devons faire en sorte que, par construction, cette situation n'arrive jamais.

Nous proposons à travers le théorème de contrôlabilité, présenté ci-après, une adaptation du théorème de virtualisation de Popek et Goldberg, dans laquelle les conditions d'application ont été généralisées :

Théorème 6.2 (Contrôlabilité)**.** *Un logiciel de contrôle vérifiant les propriétés de contrôle des ressources et d'équivalence peut être construit sur le cœur d'un processeur donné si ce dernier vérifie les propriétés suivantes :*

(i) *Tout évènement sensible (cf. définition 6.2) est soit contrôlable (cf. définition 6.3), soit rendu impossible.*

(ii) *Tout évènement contrôlable dont l'occurrence est possible peut être reproduit par une ou plusieurs séquences de contrôle.*

On pourra noter que l'hypothèse (i) reprend la condition d'application du théorème de virtualisation de Popek et Goldberg.

En comparaison avec le théorème de virtualisation de Popek et Goldberg, la preuve de ce théorème est plus concise. Pour cause, l'hypothèse 6.1, portant sur la possibilité d'émuler un jeu d'instruction, n'est plus admise. Elle fait désormais parti des conditions d'application du théorème (hypothèse (ii)). Toutefois, le schéma de notre preuve reste le même que pour le théorème de virtualisation.

La propriété de contrôle des ressources se déduit de l'hypothèse (ii). Si on considère un évènement sensible, l'application de la séquence de contrôle correspondant à cet évènement assure que le logiciel de contrôle maîtrise les dates des accès qu'il effectue à destination des ressources communes, et qu'il n'en effectue pas pour sa propre exécution. Il suffit alors que le logiciel de contrôle fasse coïncider ces dates avec des fenêtres temporelles où les accès en question sont autorisés.

La propriété d'équivalence se montre par récurrence, en suivant un schéma analogue à celui de Popek et Goldberg, que nous avons détaillé précédemment.

Nous disposons à travers ce cadre d'une définition du logiciel de contrôle qui lui confère de bonnes propriétés :

- La propriété de contrôle des ressources apporte l'assurance que le logiciel de contrôle interviendra pour empêcher le logiciel utile d'accéder directement aux ressources communes. Le logiciel de contrôle effectuera lui-même ses accès à des dates maîtrisées, en accord avec la stratégie d'utilisation.
- Grâce à la propriété d'équivalence, le logiciel utile n'est pas conscient de l'intervention du logiciel de contrôle, hormis une altération de son temps d'exécution. Cette propriété nous assure la rétro-compatibilité vis-à-vis du logiciel existant.

Nous disposons d'un théorème qui s'il est applicable garantit l'existence de notre logiciel de contrôle sur un processeur donné. Les conditions d'application de ce théorème

se traduisent en propriétés à vérifier sur le processeur. Nous nous intéressons dans la section suivante à la manière d'appliquer ce théorème sur des processeurs COTS.

6.3 Application du théorème de contrôlabilité

Nous avons introduit dans la section précédente le théorème de contrôlabilité, qui conditionne l'existence d'un logiciel de contrôle portant sur les accès du logiciel utile aux ressources communes d'un processeur multi-cœurs. Nous proposons à présent une démarche visant à appliquer ce théorème sur un processeur donné. Nous illustrons cette démarche sur un modèle simplifié de processeur.

6.3.1 Vue d'ensemble de la démarche

Les conditions d'application du théorème de contrôlabilité correspondent à des propriétés à vérifier sur les cœurs du processeur. Ces propriétés portent sur le comportement du cœur lors d'évènements matériels dits sensibles, que nous avons définis dans la section 6.2.1 comme étant l'origine d'un accès d'un cœur vers une ressource commune.

Ces évènements sensibles sont liés à l'exécution du logiciel, parfois à l'exécution d'une instruction en particulier. Le logiciel de contrôle doit garantir d'une part que le logiciel utile ne sera jamais à l'origine d'un évènement sensible, sans connaître a priori ce logiciel utile. C'est la propriété de contrôle des ressources. D'autre part, il doit lui permettre de s'exécuter correctement, là encore sans en connaître le détail. C'est la propriété d'équivalence. Nous considérons ici que le logiciel utile est une suite quelconque d'instructions exécutées dans le niveau de privilège utilisateur.

Nous proposons dans cette section une méthodologie visant à établir les conditions d'applications du théorème de contrôlabilité. Cette méthode est illustrée dans cette section sur un modèle de processeur simplifié. Elle a toutefois vocation à passer à l'échelle pour être appliquée sur des cœurs de processeurs COTS.

Identification des évènements sensibles

La première opération consiste à identifier les évènements sensibles sur le cœur, et à formaliser les relations entre évènements, sous la forme *d'arbres d'activités*.

Un exemple d'arbre d'activités est donné sur la figure 6.5. On peut constater que les nœuds correspondent à des composants du cœur, ici le cache L2. Certaines feuilles sont colorées en vert pour signifier que l'évènement final n'est pas sensible, en rouge quand l'évènement final l'est, et en orange (non représenté sur la figure) quand une exception est levée, et donc que l'évènement a été contrôlé. Les transitions sont définies avec une garde, qui est une condition sur l'état courant du cœur, et/ou un ou plusieurs évènements émis suite au franchissement de la garde. Par construction, toutes les transitions pouvant être franchies le sont.

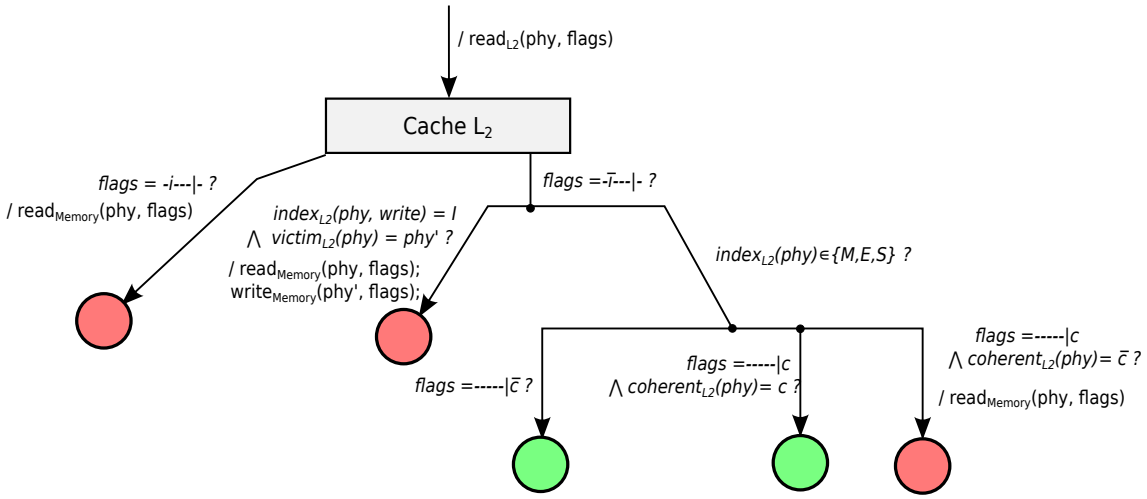


FIGURE 6.5 – Exemple d'arbre d'activité d'une opération de lecture sur un cache

Au terme de cette étape, nous disposons de la liste des évènements et des relations entre ces évènements, ces relations pouvant dépendre de propriétés sur l'état du cœur.

Classification et traitement des évènements sensibles

La seconde étape vise à classer les évènements selon leur type de sensibilité et leur contrôlabilité. Nous proposons à cet effet des règles de classification permettant de typer la sensibilité et la contrôlabilité d'un évènement. Pour cela, nous étudions les arbres d'activités à la recherche d'évènements sensibles et/ou d'exceptions.

La première hypothèse d'application du théorème de contrôlabilité précise que tout évènement sensible doit être contrôlable ou bien rendu impossible. Une fois la classification établie, nous nous intéressons aux évènements non contrôlables et aux évènements partiellement contrôlables. Pour les cas non contrôlables, nous devons éliminer les occurrences d'évènements sensibles. En d'autres termes, un évènement non contrôlé doit forcément être non sensible.

Comme illustré sur la figure 6.6, on cherchera à éliminer les évènements identifiés précédemment en introduisant une propriété qui va restreindre l'espace d'états du cœur. On parle *d'invariant*. Un exemple d'invariant pourrait être : "Pour toute donnée référencée par la MMU, l'état de cette donnée est 'valide' dans le cache L_2." Le logiciel de contrôle est en charge de mettre en place l'invariant. Il convient donc de construire ce dernier de telle sorte qu'il soit implémentable dans le logiciel de contrôle. Nous proposons par la suite des règles de construction de l'invariant.

Lorsqu'un évènement ne peut être éliminé par un invariant, nous cherchons à l'éliminer en introduisant des *règles de conception*, qui peuvent s'appliquer au logiciel de contrôle et/ou au logiciel invité. Un exemple de règle de conception pourrait être "Le logiciel invité n'utilise pas l'instruction *dcbf* (data cache block flush)". On pourra noter que lorsqu'une règle de conception s'applique au logiciel utile, celle-ci empeche la réutilisation de tout

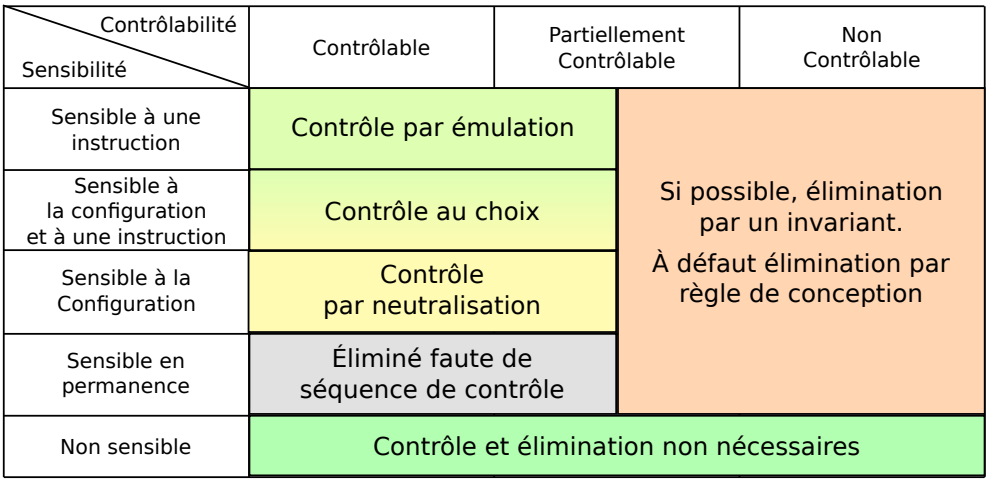


FIGURE 6.6 – Vue d'ensemble des traitements à effectuer selon la sensibilité et la contrôlabilité des évènements

logiciel existant qui ne la vérifie pas. Il convient donc de minimiser les règles de conception applicables au logiciel utile.

Au terme de cette étape, nous avons un ensemble d'évènements classés selon leur type de sensibilité et leur contrôlabilité. Les évènements non contrôlables et/ou sensibles en permanence sont éliminés. Nous vérifions donc la première hypothèse du théorème de contrôlabilité.

Construction des séquences de contrôle

La construction des séquences de contrôle est propre à chaque évènement sensible. Un évènement sensible à une instruction aura vocation à être associé à une séquence de contrôle par émulation, tandis qu'un évènement sensible à la configuration aura vocation à être émulé par une séquence de contrôle par neutralisation. Lorsqu'un évènement est sensible aux deux (cf figure 6.6), il peut disposer de plusieurs séquences de contrôle, et l'utilisateur a le choix de celle qu'il utilisera.

La construction des séquences de contrôle valide la seconde hypothèse d'application du théorème de contrôlabilité, ce qui nous permet de conclure. Nous développons cette méthode dans la suite de la section.

6.3.2 Modèle de cœur considéré

Nous souhaitons déterminer des éléments de la stratégie de contrôle qui puissent s'appliquer à des classes de processeurs larges. Les évènements matériels sensibles, générant des accès vers les ressources communes sont propres à chaque processeur. Toutefois, pour un certain nombre de composants, les comportements que l'on peut observer sont proches. Par exemple un cache, lorsqu'il reçoit une requête de lecture d'une donnée à

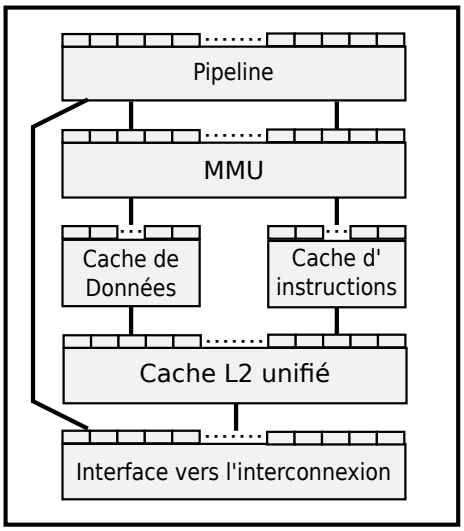


FIGURE 6.7 – Modélisation d'un cœur à deux niveaux de caches internes

une certaine adresse, va renvoyer la donnée contenue à cette adresse s'il la contient, et propagera automatiquement la requête à la mémoire principale dans le cas inverse. Dans le premier cas, on peut s'attendre à ce que l'évènement ne soit pas sensible, i.e. qu'aucun accès ne soit émis vers la mémoire principale. Dans le second cas, l'évènement sera toujours sensible. Nous nous intéressons ici aux comportements des caches et de la MMU, qui sont des composants que l'on rencontre dans la plupart des processeurs utilisés dans l'avionique.

Les caches, comme la MMU, ont une fonction clairement identifiée, qui varie peu d'un processeur à l'autre. On peut donc s'attendre à ce que les évènements sensibles qui leur sont associés soient à peu près les mêmes. Nous cherchons donc à identifier et classifier les évènements sensibles liés aux caches et à la MMU, à partir de modèles de cœurs. Cette classification a pour finalité de déterminer des invariants sur l'état d'un cœur applicables à une large classe de processeurs. Plus précisément, nous visons une architecture à deux niveaux de caches, telle que décrite sur la figure 6.7

6.3.3 Définitions des évènements sensibles

La première étape de cette méthode vise à identifier les évènements sur le processeur. Dans notre cas, il n'est pas réellement question d'identifier mais plutôt de définir les évènements supportés par notre modèle.

Nous utilisons dans la suite de ce chapitre les notations suivantes :

$@ \in$ ***A.*** Espace d'adressage effectif, vu et manipulé par le logiciel.

phy $\in$ ***Phy.*** Espace d'adressage physique, vu et manipulé dans les caches et ressources communes. Le lien entre les deux espaces d'adressages est effectué par la MMU.

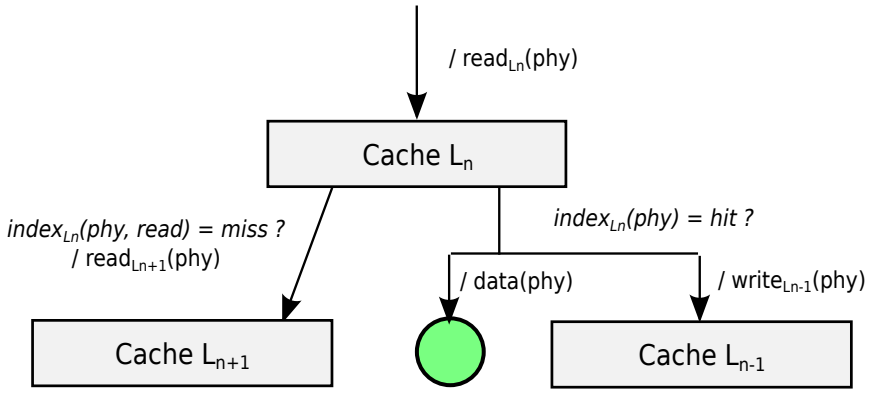


FIGURE 6.8 – Activité d'un cache sollicité par une requête en lecture

L₁, L₂. Les deux niveaux de cache supportés par le cœur. Nous ne dissocions pas le niveau L_1 pour les données et les instructions.

Définition des évènements

Nous considérons les évènements suivants :

load/store/fetch(@). Lecture, écriture ou chargement d'instruction à une adresse effective donnée.

read/write(phy). Lecture ou écriture d'une donnée à l'adresse *phy*.

data(phy). Transmission des données contenues à l'adresse $phy \in Phy$ au pipeline. Cet évènement termine l'activité, i.e. ne peut être suivi d'autres évènements.

Exception. Levée d'une exception dirigée vers le logiciel de contrôle.

Sur un processeur COTS, cet ensemble d'évènements doit être défini après une analyse du comportement du cœur, ainsi que du jeu d'instructions.

Caractérisation du comportement des éléments du cœur

Nous définissons le comportement des caches et de la MMU comme suit :

MMU. L'état de la MMU est représenté par une fonction de *pagination*, qui associe à une adresse effective une adresse physique, ou retourne une erreur en cas de défaut de page :

Map : $A \times \{read, write, fetch\} \longmapsto \{Phy\ ,\ miss\}$

Caches. L'état d'un cache est représenté par :

- La fonction *d'indexation* : ***index*** : $Phy \longmapsto \{hit, miss\}$, qui définit la présence ou non d'une donnée dans un cache.
- La fonction de *remplacement* : ***victim*** : $Phy \longmapsto Phy$, qui associe à l'adresse d'une donnée nouvelle celle d'une donnée "victime", qui sera par la suite écrite dans le cache de plus bas niveau, ou à défaut vers la mémoire principale.

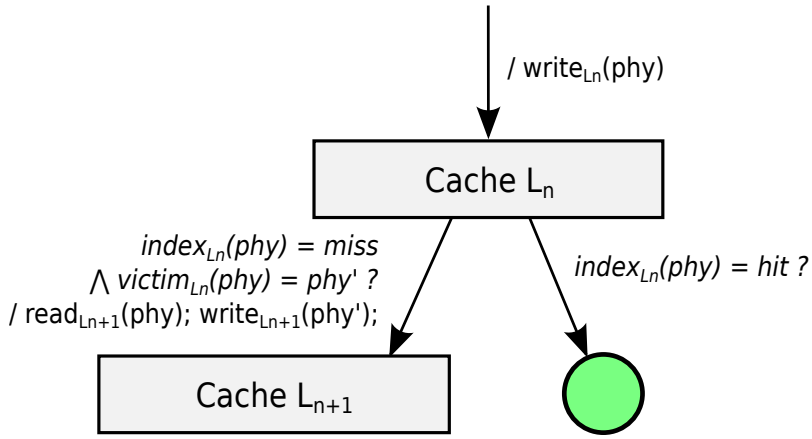


FIGURE 6.9 – Activité d'un cache sollicité par une requête en écriture

Le comportement du cache est illustré sous forme d'arbres d'activités sur les figures 6.9 et 6.8. La composition des arbres d'activités permet de définir des arbres d'activités sur l'intégralité du cœur, tel qu'illustré sur la figure 6.10. À partir de ces définitions, nous pouvons passer à la deuxième étape de la méthodologie, à savoir la classification des évènements sensibles.

6.3.4 Classification des évènements sensibles

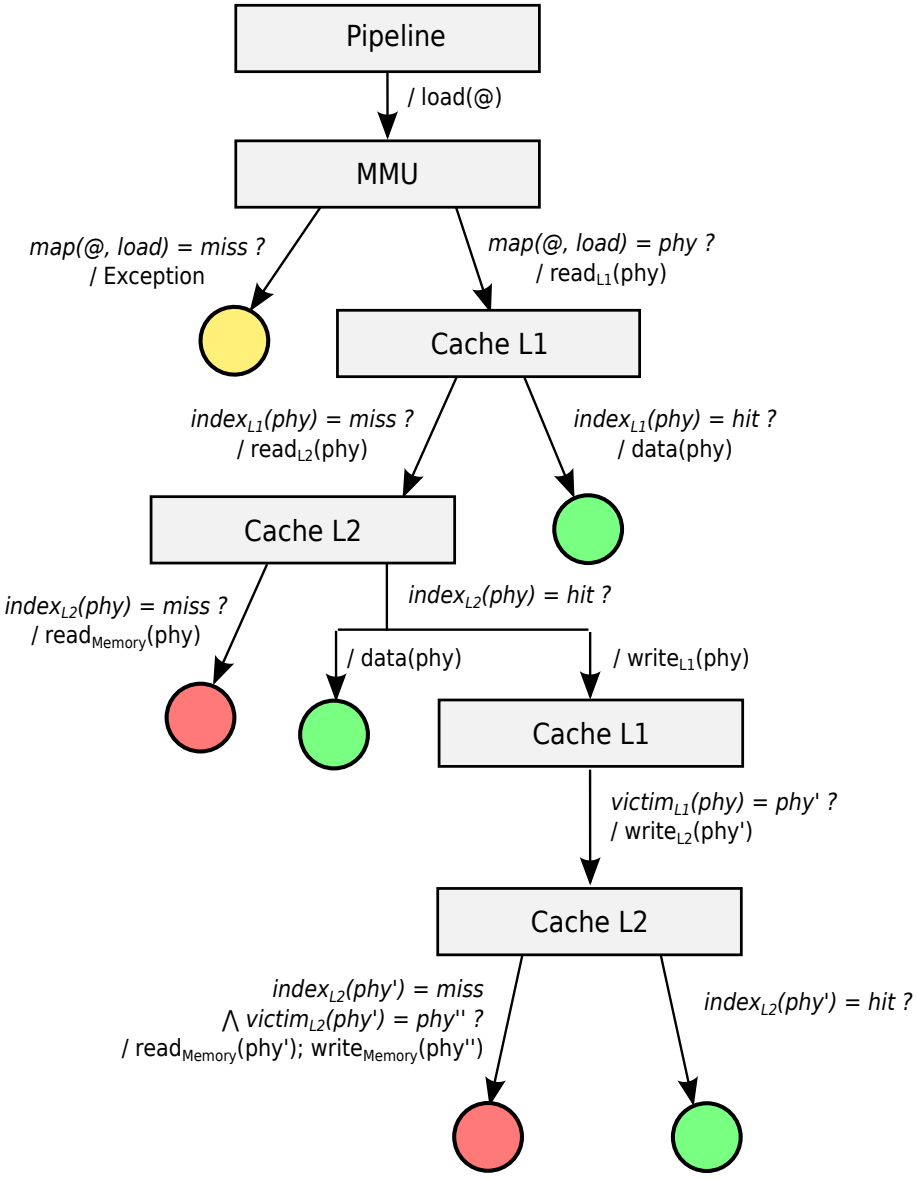
La classification des évènements sensibles s'effectue selon les règles suivantes :

- Les évènements sensibles issus de l'analyse du jeu d'instructions du cœur sont par définition sensibles aux instructions. Dans notre cas, nous pouvons supposer que c'est le cas pour les évènements de type *load* et *store*.
- Un évènement est sensible à une instruction si tous ses prédécesseurs possibles sont sensibles à une instruction.
- Un évènement est sensible à la configuration s'il possède parmi ses descendants au moins un évènement sensible (rouge) et un évènement non sensible (vert).
- Un évènement est sensible en permanence si tous ses descendants sont sensibles (rouges).
- Un évènement est non sensible si tous ses descendants sont non sensibles (verts).

De la même manière, nous proposons les règles de classification suivantes pour la contrôlabilité :

- Un évènement est contrôlable si tous ses descendants lèvent une exception (orange).
- Un évènement est partiellement contrôlable si au moins un de ses descendants lève une exception (orange).
- À défaut, un évènement est non contrôlable.

L'application de ces règles permet d'obtenir une classification des évènements applicable à notre modèle de processeur. Cette classification est détaillée sur la table 6.2.



Note : Une transition dans l'arbre d'activité est conditionnée par une garde (en italique), portant sur un état du cœur. Il s'accompagne de l'occurrence d'un ou plusieurs évènements, notés après le "/"

FIGURE 6.10 – Arbre d'activités associé à un évènement de type *load*

TABLE 6.2 – Classification des évènements définis sur le modèle

Évènement	Sensible à une instruction	Sensible à la configuration	Sensible en permanence	Non sensible	Contrôlable	Partiellement contrôlable	Non contrôlable
Évènements associés au pipeline							
load(@)	✓	✓				✓	
store(@)	✓	✓				✓	
fetch(@)		✓				✓	
Évènements associés aux caches							
$read_{L1}(phy)$		✓					✓
$write_{L1}(phy)$		✓					✓
$read_{L2}(phy)$		✓					✓
$write_{L2}(phy)$		✓					✓
$read_{Memory}(phy)$			✓				✓
$write_{Memory}(phy)$			✓				✓
data(phy)				✓	N/A		

Nous pouvons constater que les évènements associés au pipeline sont partiellement contrôlables, et que les autres ne le sont pas. Il est alors nécessaire de configurer le cœur pour éliminer les cas où les évènements non contrôlables sont sensibles. C'est l'objet de la section suivante, qui porte sur la construction d'un invariant.

6.3.5 Construction d'un invariant

La majorité des évènements que nous avons définis se trouvent être non contrôlables. Il est alors nécessaire de prévenir leurs occurrences dans des conditions où ils sont sensibles. Nous proposons à cet effet de construire des propriétés invariantes sur l'état des ressources du cœur, dans notre cas la MMU et les caches. Nous parlons par la suite *d'invariants*.

L'invariant doit être correct, i.e. éliminer toutes les occurrences sensibles d'évènements non contrôlables. Il doit également être mis en place par le logiciel de contrôle. Par

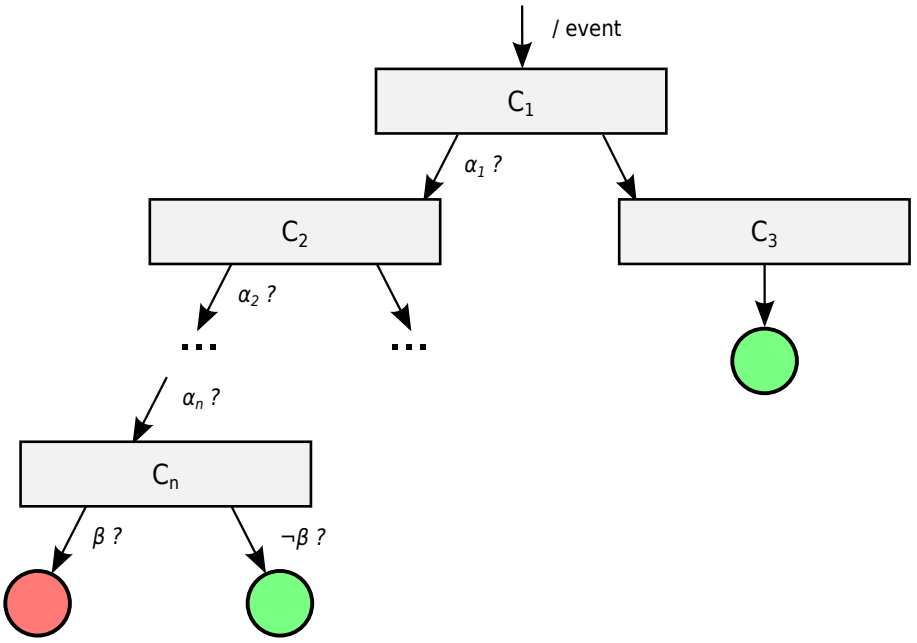


FIGURE 6.11 – Construction de l'invariant à partir de l'arbre d'activités

conséquent il doit porter sur des propriétés que ce dernier peut manipuler. Nous proposons une méthode de construction de l'invariant en deux étapes, portant respectivement sur la construction d'un invariant à partir des arbres d'activités, et la réduction de cet invariant.

Construction de l'invariant à partir des arbres d'activités

Dans un premier temps on construit un invariant correct en reprenant directement les arbres d'activités. Pour chaque feuille de l'arbre d'activités correspondant à un évènement sensible (rouge), on écrit une clause composée de la conjonction des gardes de la racine de l'arbre à cette feuille, à l'exception de la dernière dont on prend la négation. Cela donne sur l'exemple de la figure 6.11 une clause de la forme $\alpha_1 \wedge ... \wedge \alpha_n \Rightarrow \neg\beta$.

Répétée sur chaque évènement sensible, cette procédure produit un ensemble de clauses sur le même modèle que ci-dessus. Appliquée à l'arbre d'activités que nous avons construit, représenté sur la figure 6.10 à la page 82, nous obtenons les clauses suivantes :

$$\forall @ \in A,\ op \in \{load,\ store,\ fetch\},\ map(@, op) = phy \ \wedge\ index_{L1}(phy) = miss \Rightarrow index_{L2}(phy) = hit \tag{1.1}$$

$$\forall @ \in A,\ op \in \{load,\ store\},\ map(@, op) = phy \ \wedge\ index_{L1}(phy) = miss \wedge index_{L2}(phy) = hit \ \wedge\ victim_{L1}(phy) = phy' \Rightarrow index_{L2}(phy') = hit \tag{1.2}$$

Informellement, la clause (1.1), signifie que dès lors qu'une donnée est référencée par la MMU et absente du cache L_1, celle-ci est contenue dans le cache L2. La clause (1.2)

signifie que toute donnée sélectionnable comme victime par le cache L_1 est également présente dans le cache L2.

Le logiciel de contrôle devra mettre en place cet invariant avant de lancer le logiciel utile. Cependant, on n'a pas de garantie que l'invariant puisse être implémenté. Certaines clauses portant sur l'état de certains composants peuvent en effet être difficiles à implémenter dans le logiciel de contrôle. L'état et le contenu d'un cache, par exemple, est complexe à maîtriser. On peut s'appuyer sur des instructions de verrouillage de cache quand elles sont proposées par le jeu d'instruction –c'est le cas pour PowerPC et ARM– et supportées par le cœur. Encore faut-il mettre en place la logique permettant de maintenir l'état du cache et la politique de remplacement.

Il est donc pertinent de considérer des invariants réduits pour que le logiciel de contrôle puisse les implémenter de manière plus simple. C'est l'objectif de la seconde phase, dite de réduction de l'invariant.

Réduction de l'invariant

Comme expliqué précédemment, il est intéressant de réduire un invariant à un ensemble de propriétés sur des ressources que le logiciel de contrôle pourra effectivement manipuler. En considérant que l'invariant est constitué d'un ensemble de clauses de la forme $\bigwedge_{i\in[1\cdots n]}\{\alpha_i\} \Rightarrow \beta$. Nous proposons les règles de réduction suivantes :

- On peut retirer certains termes d'une clause lorsqu'ils portent sur des propriétés non implémentables par le logiciel de contrôle. La clause obtenue est de la forme $\bigwedge_{i\in I}\{\alpha_i\} \Rightarrow \beta$ pour $I \subset [1\cdots n]$. Les variables libres, s'il y en a, deviennent universellement quantifiées.
- Si pour un terme α_i donné on dispose d'un terme γ tel que la propriété $\alpha_i \Rightarrow \gamma$ soit vraie, alors γ peut être substitué à α_i dans toute clause où ce dernier apparaît. De même, si la propriété $\gamma \Rightarrow \beta$ est vraie, alors γ peut être substitué à β.

L'application de ces règles de réduction se fait au cas par cas, selon la manière dont on interprète une clause de l'invariant. Il y a là un compromis à trouver, compromis qui entraîne des choix de conception sur le logiciel de contrôle.

En effet, réduire un invariant en invoquant la première règle de réduction permet d'éviter d'avoir à mettre en œuvre une propriété sur un composant que l'on ne maîtrise pas, mais aussi sur un composant que l'on n'a pas envie de maîtriser. Cela peut s'expliquer par un souci de limiter la complexité du logiciel de contrôle, ou d'améliorer son efficacité. À l'inverse, la réduction d'un invariant élargit l'ensemble des configurations matérielles écartées. Le risque est d'éliminer des configurations où des ressources auraient pu être utilisées sans que cela ne génère d'évènements sensibles. Le compromis doit être adopté lors de la phase de conception du logiciel de contrôle.

On peut illustrer l'argument d'amélioration de l'efficacité sur la clause (1.1). Cette clause signifie informellement que "toute requête traduite par la MMU et non servie par le cache L_1 est servie par le cache L_2." Son implémentation supposerait que le logiciel de contrôle maîtrise le contenu de la MMU et des caches L_1 et L_2. Si la maîtrise de la

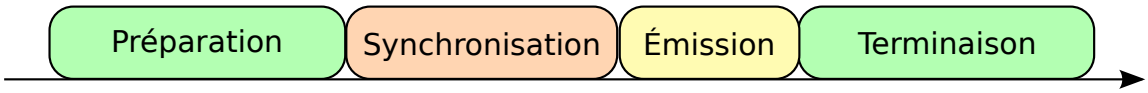


FIGURE 6.12 – Vue d'ensemble d'une séquence de contrôle

MMU ne pose pas de problèmes, celle du contenu d'un cache va au contraire en poser. Il faut implémenter une logique pour le chargement, le remplacement et le déchargement du cache. Dès lors, il est intéressant de limiter cette logique au cache L_2 et de laisser au matériel la gestion du cache L_1. Nous proposons donc la clause (2.2), qui est une réduction de la clause (1.1) dont on a retiré la référence au contenu du cache L_1. En définissant cette clause, nous avons volontairement éliminé certaines configurations, comme celle où le cache L_1 contient des données verrouillées que le cache L_2 ne contient pas.

De la même manière, la clause (1.2), qui signifie que "toute donnée sélectionnable comme victime par le cache L_1 est incluse dans le cache L_2, peut être réécrite en enlevant les premiers termes, qui sont contenus dans la première clause. Nous obtenons la clause (2.2), définie ci-dessous :

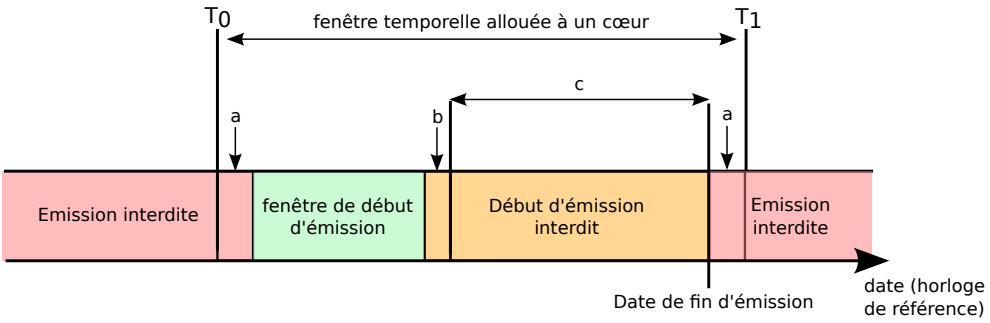
$$\forall @ \in A,\ op \in \{load,\ store,\ fetch\},\ map(@, op) = phy \Rightarrow index_{L2}(phy) = hit \quad (2.1)\ (\text{anc. } 1.1)$$

$$\forall phy \in Phy, victim_{L1}(phy) = phy' \Rightarrow index_{L2}(phy') = hit \quad (2.2)\ (\text{anc. } 1.2)$$

L'invariant final est composé des clauses (2.1) et (2.2).

La phase de réduction de l'invariant est importante car elle impacte la conception du logiciel de contrôle, ainsi que son efficacité. Les choix effectués lors de cette phase sont des compromis entre la complexité des mécanismes à prévoir dans le logiciel de contrôle, les ressources effectivement utilisées et l'efficacité du système. Dans notre exemple, l'invariant que nous avons proposé répond à ce compromis en ignorant volontairement les contraintes sur le contenu du cache L_1 pour ne pas intervenir dans la gestion qui en est faite par le matériel. L'effet attendu est une amélioration de l'efficacité globale.

Quel que soit le résultat de la phase de réduction de l'invariant, ce dernier répond à l'objectif qui lui était fixé, à savoir éliminer les situations où des évènements non contrôlables sont sensibles. La première hypothèse d'application du théorème de contrôlabilité est atteinte. Nous abordons dans la section suivante la question de la construction des séquences de contrôle, qui complète notre démarche.



a : Précision de la mesure de la date de référence
b : Durée entre la fin de la phase de synchronisation et le début de l'émission
c : Durée de la séquence d'émission
(T_0, T_1) : Fenêtre temporelle d'émission allouée au cœur.

FIGURE 6.13 – Processus de synchronisation sur une fenêtre temporelle

6.3.6 Construction des séquences de contrôle

La dernière étape de notre méthode consiste à construire les séquences de contrôle. Nous avons défini dans la section 6.2.1 deux types de séquences de contrôles :

Émulation. Une séquence de contrôle par émulation s'applique à un évènement sensible à une instruction. Elle aura pour objectif de reproduire les effets de l'instruction sur les ressources internes du cœur et sur les ressources distantes, sans toutefois exécuter l'instruction en question.

Neutralisation. Une séquence de contrôle par neutralisation s'applique à un évènement sensible à la configuration. Elle n'a pas pour objectif de reproduire l'évènement mais plutôt de placer le cœur dans une configuration où ce dernier n'est plus sensible, pour ensuite laisser le logiciel utile le réémettre.

Comme illustré sur la figure 6.6, page 78, lorsque l'évènement est sensible à la fois à une instruction et à la configuration, on peut lui associer plusieurs séquences de contrôle. On pourra également constater qu'un évènement contrôlable mais sensible en permanence ne peut être reproduit. Si ce type d'évènement est possible sur le cœur, il convient de poser une limitation d'usage demandant d'éviter les opérations et mécanismes associés. À titre d'exemple, on peut citer le chargement d'une instruction depuis une mémoire non cachable.

Nous proposons un schéma de séquence de contrôle en quatre phases, représentées sur la figure 6.12 :

Phase de préparation. Planification des accès à réaliser et préparation des ressources internes du cœur. Cette préparation peut invalider l'invariant vis-à-vis du logiciel utile. Elle est propre à chaque évènement.

Phase de synchronisation. Le logiciel de contrôle est en attente active jusqu'à la prochaine date d'émission. Cette phase suppose que le processeur est synchronisable, au sens de la propriété 6.6.

Phase d'émission. Émission de la séquence d'accès planifiée. Il est préférable que cette phase soit synchrone, i.e. termine avec la certitude que l'accès a bien été réalisé.

Phase de terminaison. Terminaison de l'accès et rétablissement de l'invariant s'il a été invalidé lors de la phase de préparation.

Nous développons dans la suite de cette section un exemple de séquence de contrôle par neutralisation pour un évènement de type *load(@)*. Nous supposons donc que nous venons d'intercepter l'exécution du logiciel utile par l'exception de défaut de page. Ce dernier était sur le point d'émettre une requête en lecture qui n'était pas présente dans le cache. La séquence de contrôle aura pour objectif de charger cette donnée dans le cache, puis de configurer la MMU pour référencer cette donnée, ce qui neutralisera l'évènement *load(@)*.

Nous proposons la séquence de contrôle suivante :

Phase de préparation

L'objectif de cette phase sera de préparer les ressources internes du cœur en vue du chargement. Elle commence par une analyse de l'accès intercepté comme suit :

- Récupération de l'adresse effective (@) visée la requête à l'origine de l'exception, et du type d'accès (ici *load*).
- Calcul de l'adresse physique visée par l'accès, que nous notons *phy*.
  Cela suppose que le logiciel de contrôle est capable de se substituer à une MMU pour effectuer une traduction d'adresse. Il doit donc disposer des informations nécessaires pour effectuer cette traduction. Nous proposons d'embarquer dans le logiciel de contrôle une ***table des pages virtuelles***, dont le contenu est alimentée par le logiciel de contrôle sur la base de la configuration de la MMU voulue par le logiciel utile. Il s'agit d'une application du principe de page fantôme [33], qui est une technique que l'on rencontre pour virtualiser des périphériques.
- Sélection d'un emplacement dans le cache qui va accueillir la donnée à l'adresse *phy*. Cette donnée sera chargée avec l'intégralité de la page mémoire la contenant. Il s'agit d'une conséquence de l'invariant, qui dit que toute donnée référencée par la MMU doit être présente dans le cache L_2. Si la donnée à l'adresse *phy* est référencée, toute donnée référencée par la même page l'est également. Elle doit par conséquent être chargée dans le cache L_2. La granularité des échanges entre le cache et la mémoire principale est par conséquent la taille d'une page dans la MMU.
  La sélection d'un emplacement pour la page à charger demandera éventuellement de choisir une page victime. Pour effectuer cette sélection, le logiciel de contrôle doit être capable de connaître l'état des données présentes dans le cache L_2. Nous

33. Shadow Page

proposons à cet effet d'embarquer dans le logiciel de contrôle une table de données contenant des ***descripteurs de pages*** présentes dans le cache. Chaque descripteur de page doit contenir les informations permettant au logiciel de contrôle de décharger la page correspondante du cache, et de la déréférencer dans la MMU. La sélection de l'emplacement dans le cache doit enfin tenir compte de sa structure, en particulier de son degré d'associativité. Nous organisons la table des descripteurs de pages selon la même structure et le même degré d'associativité que le cache physique.

Pour garantir la conformité avec l'invariant, les données appartenant à la page victime contenues dans le cache L_1 devront être écrites dans le cache L_2.

Phase de synchronisation

Nous mettons en place une phase de synchronisation telle qu'illustrée sur la figure 6.13. Cette phase de synchronisation prend en compte les paramètres suivants :

- L'imprécision de la mesure de l'horloge de référence (a).
- Le délai entre la fin de la phase de synchronisation et le début de la phase d'émission à proprement parler (b).

En outre elle garantit que la phase d'émission commence à une date où elle aura le temps de terminer dans sa fenêtre.

Phase d'émission

Cette phase met en œuvre une séquence d'instructions qui charge une page mémoire complète dans le cache L_2. Sur un processeur réel, il faudrait identifier la séquence d'instructions correspondante.

Phase de terminaison

La phase de terminaison va chercher à finaliser la séquence de contrôle, par exemple en verrouillant la page nouvellement chargée dans le cache L_2 afin de la rendre résidente. Elle pourra également mettre à jour des statistiques d'utilisation des ressources communes.

Au terme de cette séquence, le logiciel de contrôle peut effectuer un retour sur exception, qui entraîne la reprise de l'exécution du logiciel utile. Ce dernier émettra à sa requête en lecture, qui réémettra l'évènement *load(@)*, cette fois ci dans une configuration où il est neutre.

Nous avons correctement neutralisé l'évènement intercepté, et la séquence de contrôle n'a pas invalidé l'invariant. En définissant d'une manière analogue des séquences de contrôle pour les autres évènements, nous pouvons conclure sur la deuxième hypothèse d'application du théorème de contrôlabilité.

6.4 Synthèse

Nous avons présenté dans ce chapitre une formalisation d'un logiciel de contrôle d'évènements sensibles sur un processeur donné. Dans notre cas, ces évènements sensibles correspondent à des accès des cœurs aux ressources communes d'un processeur multi-cœurs. Nous considérons une configuration où le logiciel de contrôle est instancié sur chaque cœur. L'instance locale exerce un contrôle de manière autonome sur un logiciel utile, qui est déployé sur ce cœur.

Le logiciel de contrôle doit vérifier les deux propriétés suivantes, inspirées du cadre de virtualisation de Popek et Goldberg [69] :

Contrôle des ressources. Aucun accès ne doit être émis en dehors de fenêtres temporelles définies statiquement, dans lesquelles l'émission d'accès est autorisée.

Équivalence. Le logiciel de contrôle doit donner l'illusion au logiciel utile, qui subit le contrôle, qu'il est le seul à s'exécuter sur un cœur donné.

En outre, il est attendu du logiciel de contrôle qu'il soit le plus efficace possible, c'est-à-dire qu'il n'exerce un contrôle sur le logiciel utile que lorsque c'est nécessaire. L'efficacité est ici vue comme une bonne propriété plus qu'une exigence.

Nous avons proposé un théorème qui généralise celui de Popek et Goldberg, et qui conditionne l'existence de ce logiciel de contrôle à deux hypothèses, portant respectivement sur la contrôlabilité d'évènements matériels sensibles, et l'existence de séquences de contrôle reproduisant ces évènements.

Nous avons enfin décrit une démarche permettant d'appliquer ce théorème. Celle-ci est constituée des opérations suivantes :

- L'identification des évènements sensibles sur le cœur.
- La classification des évènements sensibles selon leur type de sensibilité et leur contrôlabilité. Il convient d'éliminer toute configuration où un évènement sensible n'est pas contrôlable. À cet effet, nous construisons un invariant sur l'état du cœur. Si l'invariant ne couvre pas tous les cas visés, ou n'est pas directement implémentable, nous introduisons des règles de conception à faire valoir tant sur le logiciel de contrôle que sur le logiciel utile. Au niveau du logiciel utile, ces règles sont acceptables si elles sont compatibles avec la majorité du logiciel existant.
- La construction de séquences de contrôles pour les évènements contrôlables. Ces séquences de contrôle visent à émuler ou à neutraliser un évènement sensible selon son type de sensibilité.

Nous avons illustré notre cadre de logiciel de contrôle en nous basant sur un modèle de processeur simplifié, contenant deux niveaux de caches et une MMU. Ce modèle illustre les interactions entre les caches, et la manière de contrôler un logiciel utile qui utilise de la mémoire cacheable. Le chapitre suivant a pour objectif de montrer que cette démarche peut être appliquée à un processeur COTS.

Chapitre 7

Cas d'étude sur un processeur COTS

Sommaire

7.1 Cible matérielle 92
7.1.1 Gamme de processeurs ciblée et choix de la cible 92
7.1.2 Description du cœur e5500 94
7.2 Description du cas d'étude 97
7.3 Identification des évènements sensibles 99
7.3.1 Évènements associés au pipeline 101
7.3.2 Évènements associés à la MMU 103
7.3.3 Évènements associés aux caches 105
7.3.4 Construction des arbres d'activité au niveau du cœur . . 107
7.4 Classification des évènements sensibles 112
7.4.1 Classification des évènements sensibles 113
7.4.2 Construction de l'invariant 114
7.4.3 Robustesse de l'invariant vis-à-vis du logiciel utile 121
7.5 Construction des séquences de contrôle 123
7.5.1 Exemple de séquence de contrôle par neutralisation . . . 125
7.5.2 Exemple de séquence de contrôle par émulation 127
7.6 Synthèse 129

Dans le chapitre précédent, nous avons généralisé le cadre de virtualisation de Popek et Goldberg pour définir notre logiciel de contrôle. Ce dernier est caractérisé par les propriétés de contrôle des ressources et d'équivalence. La première signifie que tout accès électronique d'un cœur vers une ressource partagée a lieu à une date choisie par le logiciel de contrôle. La seconde signifie que le logiciel utile –qui subit le contrôle– a l'illusion d'accéder directement aux ressources sans intervention du logiciel de contrôle.

Nous nous sommes intéressés dans le chapitre précédent aux conditions d'existence d'un logiciel de contrôle vérifiant ces propriétés. Ces conditions ont été réunies dans la définition du théorème de contrôlabilité. Nous avons ensuite considéré un modèle de cœur simplifié, à partir duquel nous avons développé une démarche permettant de vérifier ces

conditions. L'objectif de ce chapitre est d'appliquer cette démarche sur un processeur COTS, plus complexe que le modèle étudié précédemment.

Appliquer la démarche de contrôlabilité ne signifie pas pour autant que le processeur sera contrôlable dans un cas général. Toutefois, il est pertinent de mettre en évidence des cas particuliers dans lesquels le théorème de contrôlabilité s'applique. La mise en place de ces cas particuliers peut demander d'imposer quelques contraintes sur le développement du logiciel utile, lorsque ces dernières sont raisonnables.

Ce chapitre est découpé en cinq sections principales. La section 7.1 donne une vue d'ensemble de la gamme QorIQ et du processeur P5040, puis détaille les caractéristiques du cœur e5500, qui est au centre de l'étude. La section 7.2 donne une vue d'ensemble des attendus du cas d'étude. La section 7.3 vise à caractériser les évènements sensibles sur le cœur e5500, et à formaliser leurs relations à travers des arbres d'évènements. Il s'agit de la première étape de la méthodologie que nous avons définie au chapitre précédent. Les sections 7.4 et 7.5 développent la deuxième et la troisième étape de la méthodologie, à savoir la classification des évènements sensibles, incluant la construction de l'invariant, puis la construction des séquences de contrôle.

7.1 Cible matérielle

7.1.1 Gamme de processeurs ciblée et choix de la cible

Nous proposons un cas d'étude dans lequel nous ciblons la gamme QorIQ fabriquée par Freescale. Ce choix est motivé par des raisons industrielles. Freescale est historiquement un important fournisseur de processeurs pour l'avionique. Il répond à la plupart des critères de sélection de fabricants présents dans l'état de l'art [39], ainsi que ceux que nous avons définis lors de l'étude Mulcors [53]. Par ailleurs, Freescale a affiché sa volonté de collaborer avec l'industrie aéronautique dans le cadre de l'organisation MCFA (Multi-Core For Avionics) [84], afin de faciliter la certification de ses processeurs multi-cœurs. C'est le seul fabricant à avoir effectué cette démarche.

La gamme QorIQ, fabriquée par Freescale, est conçue pour les infrastructures réseaux, en particulier les serveurs et les switchs. Comme illustré sur la figure 7.1, on peut relever les points suivants :

- La plupart des processeurs de cette gamme sont multi-cœurs, les versions mono-cœurs étant des versions dégradées où plusieurs cœurs sont physiquement présents mais seul un est alimenté.
- Cette gamme est historiquement PowerPC, même si depuis 2014 l'offre est plus équilibrée entre PowerPC et ARM. L'industrie avionique s'intéresse en priorité à des processeurs matures, entrés en production, et inscrits dans le programme de longévité de Freescale. A ce titre, Freescale, dans le cadre du MCFA se focalise sur le P4080.
- Il y a eu une évolution de l'architecture des interconnexions, du bus MPX vers CoreNet qui est un réseau switché. Les architectures à venir sont toutes basées

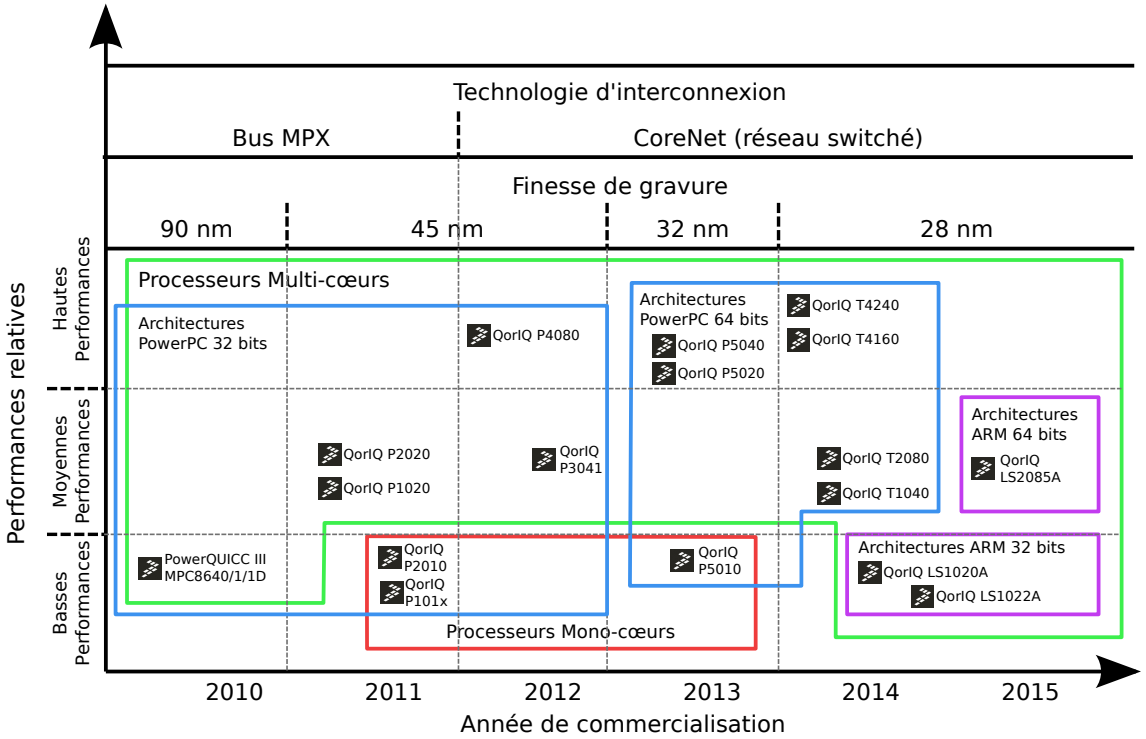


FIGURE 7.1 – Vue d'ensemble de la gamme QorIQ

sur le CoreNet. Il s'agit d'une interconnexion boîte noire sur laquelle Freescale communique peu d'informations.

Nous avons choisi de baser notre prototype sur le processeur P5040 de la gamme QorIQ. Ce choix est motivé par le fait que ce processeur embarque des cœurs e5500, dont le cache L_2 est privé et a une capacité de 512K. Cela permet de stocker un logiciel de contrôle tout en conservant de la place pour le logiciel utile, ce que d'autres processeurs ayant plus de cœurs, en l'occurrence le P4080 dont les cœurs ont 128K de cache L_2, ne permettent pas. Ce choix a également été influencé par des contraintes de disponibilité du matériel pour notre étude.

Le processeur P5040 est un processeur intermédiaire dans cette gamme. Comme illustré sur la figure 7.2, ce processeur contient quatre cœurs e5500 qui sont reliés aux périphériques par une interconnexion principale, appelée CoreNet, sur laquelle Freescale ne communique que très peu d'informations. Le processeur comporte également deux contrôleurs de mémoire de type DDR, qui sont précédés de deux caches de niveau L_3, partagés entre tous les cœurs. En outre, le P5040 embarque une grande variété de périphériques comprenant entre autres :

- Des bus à haut débit, comme le PCIe
- Des bus à bas début tels que CAN, SPI, RS232.

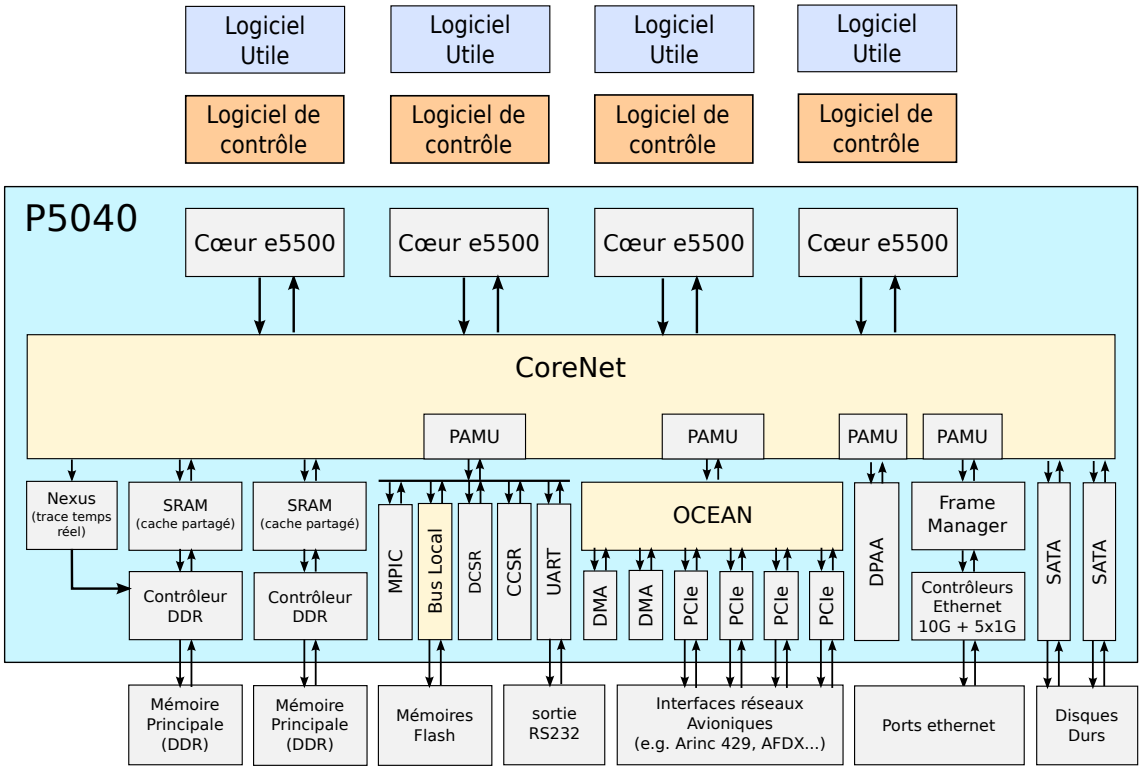


FIGURE 7.2 – Vue d'ensemble du processeur P5040 et des périphériques associés

- Des contrôleurs DMA, qui peuvent être utilisés pour une grande variété d'opérations.

Une description plus complète de l'architecture du P5040 peut être consultée dans l'annexe A. Dans notre chapitre expérimental, nous nous intéresserons principalement au contrôle d'applications qui effectuent des accès dirigés vers la mémoire principale.

L'étude de la contrôlabilité du processeur étant principalement portée sur les cœurs, nous détaillons les aspects architecturaux d'un cœur e5500 dans la section suivante.

7.1.2 Description du cœur e5500

Un P5040 embarque quatre cœurs e5500. Il s'agit d'un modèle de cœur développé par Freescale dans la famille des cœurs e500 [85], qui sont des cœurs PowerPC. Le cœur e5500, illustré sur la figure 7.3, embarque les composants suivants :

Caches L_1. Le cache de niveau L_1 est séparé au niveau données et instructions (architecture Harvard). Les caches L_1D et L_1I ont une capacité de 32K chacuns, et sont organisés en 8 voies de 4K.

Cache L_2. Le cache de niveau L_2 est unifié (architecture Von Neumann) et a une capacité de 512K. Il est organisé en 8 voies de 64K chacune. Il a un comportement

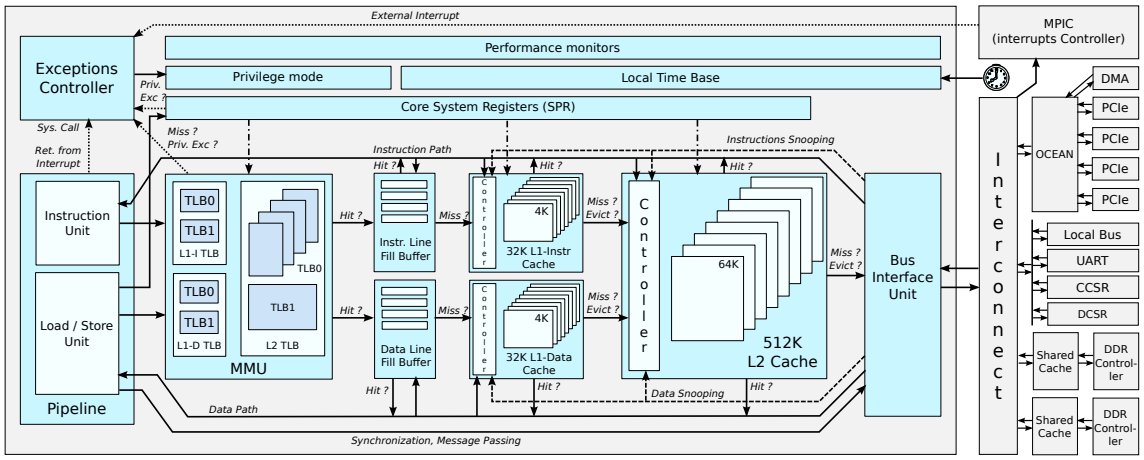


FIGURE 7.3 – Vue d'ensemble du cœur e5500

qualifié par Freescale de "dynamic harvard". Ce comportement se caractérise par deux propriétés :

- Les lignes de cache qui sont stockées sont étiquetées comme "cohérentes" lorsqu'elles sont accédées comme des données, et "non cohérentes" lorsqu'elles sont accédées comme des instructions. Un accès non cohérent à une ligne de cache cohérente est servi, mais pas l'inverse. Ainsi, une fois qu'une ligne de cache a été demandée en tant qu'instruction, elle ne peut plus être retournée en tant que données.
- les règles de rechargement de cache diffèrent entre données et instructions. Les instructions sont recopiées dans les deux caches, L_1 et L_2. Les données sont chargées dans le cache L_1 uniquement, et sont copiées dans le cache L_2 lorsqu'elles sont remplacées dans le cache L_1.

En outre, les caches maintiennent un état de cohérence pour chaque ligne en implémentant le protocole MESI [34]. On pourra également noter que le cache L_1D peut être configuré en mode *Write Shadow*, dans lequel il propage automatiquement au cache L_2 toute requête en écriture.

Niveaux de privilèges Le cœur dispose d'un registre d'état appelé MSR (Machine State Register), décrit dans le manuel de programmation du cœur [44]. Ce registre mentionne en particulier le niveau de privilège courant sur le cœur par le bit MSR[PR]. Le mode utilisateur limite les opérations qui peuvent être effectuées sur le cœur. Ces restrictions portent sur l'exécution de certaines instructions, ainsi que la lecture et/ou l'écriture dans certains SPR. Le changement de niveau de privilège ne peut se faire que lors d'une exception du mode utilisateur vers le mode superviseur, et lors d'un retour sur exception dans le sens inverse.

Registres de configuration (SPR). Le cœur e5500 dispose d'un ensemble de registres de configuration du cœur appelés SPR (Special Purpose Register). L'accès en

34. Modified, Exclusive, Shared, Invalid

lecture et en écriture à ces registres se fait par des instructions assembleur dédiées. L'accès à certains registres est privilégié.

MMU. Le cœur e5500 dispose d'une MMU [35] qui implémente des mécanismes de pagination. Une page est définie par

- Une fenêtre dans l'espace d'adressage effectif vu par le logiciel
- Une fenêtre de même taille dans l'espace d'adressage physique vu du processeur
- Un moyen d'identifier le logiciel qui a le droit d'utiliser cette page. Sur le cœur e5500, il s'agit du numéro de processus et du niveau de privilège.
- Les opérations (lecture, écriture, chargement d'instructions) autorisées dans cette page
- Des attributs, appelés WIMGE :

  W. Write through. Une requête en écriture est automatiquement propagée au cache de niveau inférieur, puis à la mémoire principale, même si elle est servie par un cache. Dans le cas inverse, on parle de configuration *write-back*.

  I. cache Inhibited. Une requête associée à une page non cachable est ignorée par les caches, y compris les caches partagés, et est automatiquement propagée à la mémoire principale ou au périphérique concerné.

  M. Memory coherency required. Une page cohérente entraîne l'activation des mécanismes de cohérence de cache.

  G. Guarded. Une requête en données associée à une page gardée ne peut être émise spéculativement, c'est-à-dire tant que l'exécution de l'instruction concernée dépend encore de la prédiction de branchements.

  E. Endianness.

La MMU dispose de deux niveaux de caches appelées TLB [36], le second étant directement programmable. Ce dernier niveau contient un cache de 512 entrées appelé TLB0, organisé en quatre voies pour les pages de 4K indexé sur l'adresse effective, ainsi qu'un cache de 64 entrées, entièrement associatif, pour les pages de taille variable.

Chaque entrée dans la TLB définit une page. Il n'y a pas de table des pages au sens classique du terme maintenue par le matériel. Un défaut de page entraîne directement une exception dite *Instruction TLB miss* ou *Data TLB miss*. Un défaut de droits d'accès entraîne une exception *Instruction* ou *Data Storage Interrupt*. Dans tous les cas ces exceptions enclenchent le mode superviseur sur le cœur.

Timer interne. Le cœur e5500 dispose de plusieurs registres offrant les fonctions d'un timer. Parmi ceux-ci, nous retenons la ***base de temps*** (Time Base), qui est implémentée comme un registre de 64 bits s'incrémentant selon un circuit d'horloge partagé entre les cœurs, dont on sait qu'il est synchrone, i.e. le déphasage entre

35. Memory Management Unit
36. Translation Lookaside Buffer

les fronts montants sur ce circuit d'horloge est négligeable devant la durée d'un cycle. Le P5040 permet un démarrage synchrone des bases temporelles sur tous les cœurs par un registre commun. Ce mécanisme permet d'assurer la propriété de synchronisabilité, définie dans le chapitre précédent (cf page 74).

Moniteurs de performances. Le cœur e5500 dispose de compteurs matériels qui comptent l'occurrence de certains évènements à des fins statistiques. Ces évènements portent entre autre sur le comportement du matériel. Il est par exemple possible de compter le nombre d'accès qu'un cœur effectue à destination des ressources communes. L'utilisation de ces compteurs peut nous servir à la fois à caractériser finement le comportement du cœur dans des situations bien précises, mais aussi à des fins de débogage sur le logiciel de contrôle une fois fini.

Support pour la virtualisation Le cœur e5500 dispose de support matériel facilitant la mise en place d'une couche de virtualisation. Par exemple, l'information sur le niveau de privilège (MSR[PR]) peut être enrichie par une information de mode logiciel invité ou hôte (MSR[GS]). Cela a pour effet de simuler un troisième niveau de privilège. Le noyau du système d'exploitation est exécuté dans un mode de privilège dégradé qui lui interdit certaines opérations, principalement la configuration de la MMU ainsi que certains SPR, la liste complète étant donnée dans le manuel de référence [43]. Lorsque ces instructions réservées sont exécutées, cela déclenche une exception de privilège qui enclenche le mode hôte (MSR[GS] $= 0$), dans lequel est exécuté l'hyperviseur. Ce dernier peut ainsi reproduire l'effet de ces opérations comme si elles avaient été exécutées par le cœur, en altérant leur sémantique selon les spécifications de la machine virtuelle. Par exemple il peut composer les règles de traduction d'adresse mises en place dans la MMU avec un offset lié à la machine virtuelle. Ce type d'opération est à la base de tous les mécanismes de virtualisation que nous mettons en œuvre dans notre prototype.

7.2 Description du cas d'étude

Nous avons introduit le processeur P5040, puis détaillé l'architecture d'un cœur e5500. Notre objectif est de montrer la propriété suivante :

Propriété 7.1 (Contrôlabilité du cœur e5500)**.** *Un cœur e5500 intégré dans un processeur P5040 satisfait les hypothèses d'application du théorème de contrôlabilité sous réserve que les règles suivantes soient respectées par le logiciel utile :*

- *Le logiciel utile n'utilise pas les instructions sync et mbar, (barrières mémoire), ainsi que dcbf (flush de cache).*
- *Données et instructions ne partagent pas les mêmes lignes de caches, i.e. ne sont pas situées sur le même bloc de 64 octets.*

Nous posons donc des règles de développement applicables au logiciel utile. Nous pensons que leur impact en pratique est très faible sur le logiciel existant.

Les instructions visées par la première règle sont rarement utilisées dans les applications. Elles ne sont d'ailleurs pas supportées par un compilateur comme GCC. Leur

utilisation dans un système d'exploitation est plus courante. Ces instructions sont implémentées dans l'adhérence matérielle, ce qui fait qu'il n'y a qu'une seule section de code à modifier.

On peut également supposer que la seconde règle ne pose pas de difficultés en pratique. Le placement des sections de données dans un binaire est effectué à partir d'un script d'édition de lien. Il est possible de prévoir une contrainte précisant que les sections de données sont alignées sur 64 octets. C'est d'ailleurs le cas pour les scripts d'édition de lien fournis avec GCC.

Nous cherchons à appliquer la méthode définie dans le chapitre précédent et illustrée sur un modèle de cœurs simplifié. Cette méthodologie comprend les étapes suivantes :

1. **Identification des évènements sensibles**.
   La phase d'identification des évènements n'a pas été illustrée sur le modèle de processeurs, dans la mesure où nous étions partis d'un ensemble d'évènements associés au modèle (*load, store, fetch*).
   Dans le cas d'un processeur COTS, on ne connaît pas a priori la liste complète des évènements, même si certains peuvent être intuités. On a également intérêt à avoir une liste d'évènements qui couvre l'activité interne du cœur, mais qui soit aussi réduite que possible.
   L'identification des évènements passe par une analyse du jeu d'instructions, puis des différents composants que l'on rencontre dans le cœur, par exemple les caches et la MMU. Ces analyses aboutissent à la construction d'arbres d'activités pour chaque composant.
2. **Classification et contrôlabilité des évènements sensibles**.
   ▪ La phase de classification des évènements sensibles est similaire à celle qui a été développée sur le modèle de processeur. Nous partons d'arbres d'activités définis sur le cœur, qui sont obtenus en combinant ceux que nous avons défini sur les composants, puis nous appliquons les règles de classification proposées dans le chapitre précédent.
   Cette phase nous permet d'obtenir une liste d'évènements pour laquelle nous évaluons la contrôlabilité, et les différentes pistes pour la stratégie de contrôle.
   ▪ Pour les évènements sensibles qui ne sont pas contrôlables, nous cherchons à définir un invariant qui rende leur occurrence impossible. Nous appliquons dans un premier temps la démarche que nous avons suivie dans le chapitre précédent pour analyser les arbres d'activités et définir des invariants. Dans un second temps, nous montrons qu'un invariant ne peut être invalidé par le logiciel utile. Cela demande une seconde analyse du jeu d'instructions.
3. **Définition des séquences de contrôle**. Nous abordons la question des séquences de contrôle par neutralisation, ainsi que des séquences de contrôle par émulation.

Le jeu d'instructions fait donc l'objet de deux analyses. La première vise à identifier les instructions dont l'exécution est susceptible de déclencher un accès aux ressources

communes. La seconde vise à identifier les instructions qui sont susceptibles de placer –en une ou plusieurs instructions– le cœur dans un état où un invariant n'est plus vérifié. À terme, cela permettrait de déclencher des accès aux ressources communes. Les classes d'instructions considérées dans les deux analyses sont détaillées dans la table 7.1.

Nous détaillons dans la suite de ce chapitre notre méthodologie dans trois sections, correspondant aux trois étapes que nous venons de rappeler.

7.3 Identification des évènements sensibles

Nous développons dans cette section la première phase de notre démarche, qui porte sur l'identification des évènements sensibles. Dans le chapitre précédent, cette phase était peu développée, dans la mesure où nous avons considéré un modèle de cœur accompagné de la liste des évènements (*load, store, fetch*) et de la spécification d'une MMU et de caches simplifiés.

Sur le cœur e5500, cette étape est plus complexe. L'identification des évènements demande en effet une analyse sur chaque composant. Nous considérons en particulier le pipeline, la MMU et les différents niveaux de cache.

À l'issue de cette analyse, nous disposons de la liste des évènements sensibles, ainsi que des arbres d'activités sur chaque composant du cœur.

Notations utilisées

Dans la suite de ce chapitre, nous notons un évènement de la manière suivante : **event(@/phy, flags)**. Les paramètres associés à l'évènement sont optionnels. Ils correspondent aux informations suivantes :

$@ \in$ ***A.*** Espace d'adressage effectif, vu et manipulé par le logiciel.

phy $\in$ ***Phy.*** Espace d'adressage physique, vu et manipulé dans les caches et ressources communes. Le lien entre les deux espaces d'adressages est effectué par la MMU.

flags. L'ensemble des paramètres associés à l'évènement. Ces derniers sont propres à chaque type d'évènements.

Par ailleurs, nous raisonnons sur des attributs qui sont exprimés comme des champs de bits. Par exemple, pour les données entrant dans le cache L_1, le champ ***flags*** est noté ***wimge***, et correspond aux attributs WIMGE associés à l'entrée TLB qui a été utilisée lors de la traduction d'adresses. Dans ce type d'attributs, nous adoptons les notations suivantes :

- w signifie que l'attribut 'W' vaut 1.
- $\overline{w}$ signifie que l'attribut 'W' vaut 0.
- – signifie que la valeur de 'W' est quelconque.

Nous notons Σ l'espace d'états du cœur, et l'état courant est noté ***state*** $\in \Sigma$.

TABLE 7.1 – Classes d'instructions considérées dans les deux analyses

Classe d'instructions	**Première analyse***	**Seconde analyse****	**Remarques**
Opérations arithmétiques et logiques sur des nombres entiers			Le jeu d'instruction PowerPC est de type load/store. Cela signifie que les opérations arithmétiques et logiques ainsi que les instructions de branchement manipulent des données contenues dans les seuls registres internes. Leur exécution est donc entièrement interne au cœur et n'altère que ces registres internes.
Opérations sur des nombres flottants			
Branchements et évaluation de conditions			
Levée d'exceptions (e.g. appels systèmes)		✓	Du point de vue de l'émission d'accès, ces opérations sont assimilées à des branchements et sont donc neutres.
Opérations de débogage			Ces instructions n'ont pas vocation à être utilisées dans notre contexte. Nous les rendons inopérantes par configuration du cœur.
Lectures et écritures dans les registres de configuration (SPR)		✓	Une lecture ou une l'écriture dans un SPR est une opération interne non susceptible de générer un accès aux ressources communes.
Lectures et écritures dans l'espace adressable	✓	✓	
Barrières de synchronisation	✓		Une barrière de synchronisation ne vise qu'à ordonner des lectures et/ou des écritures, et à garantir la terminaison de certaines instructions. Elles n'altèrent pas l'état du cœur, mais peuvent propager la barrière sur l'interconnexion.
Gestion de la MMU	✓	✓	Pour la première analyse, nous considérons certaines instructions d'invalidation d'entrées TLB qui se propagent sur tous les cœurs.
Gestion des caches	✓	✓	
Interruptions inter-cœurs	✓	✓	

* La première analyse ne concerne que les instructions dont l'exécution (et non le chargement) est susceptible d'être un évènement sensible, i.e. qui sont susceptibles d'initier des accès vers les ressources communes.

** La seconde analyse concerne les instructions qui sont susceptibles d'altérer un invariant. Sont concernées les instructions qui sont susceptibles d'altérer l'état des registres de configuration (SPR), de la MMU et des caches, sur lesquels on peut définir un invariant.

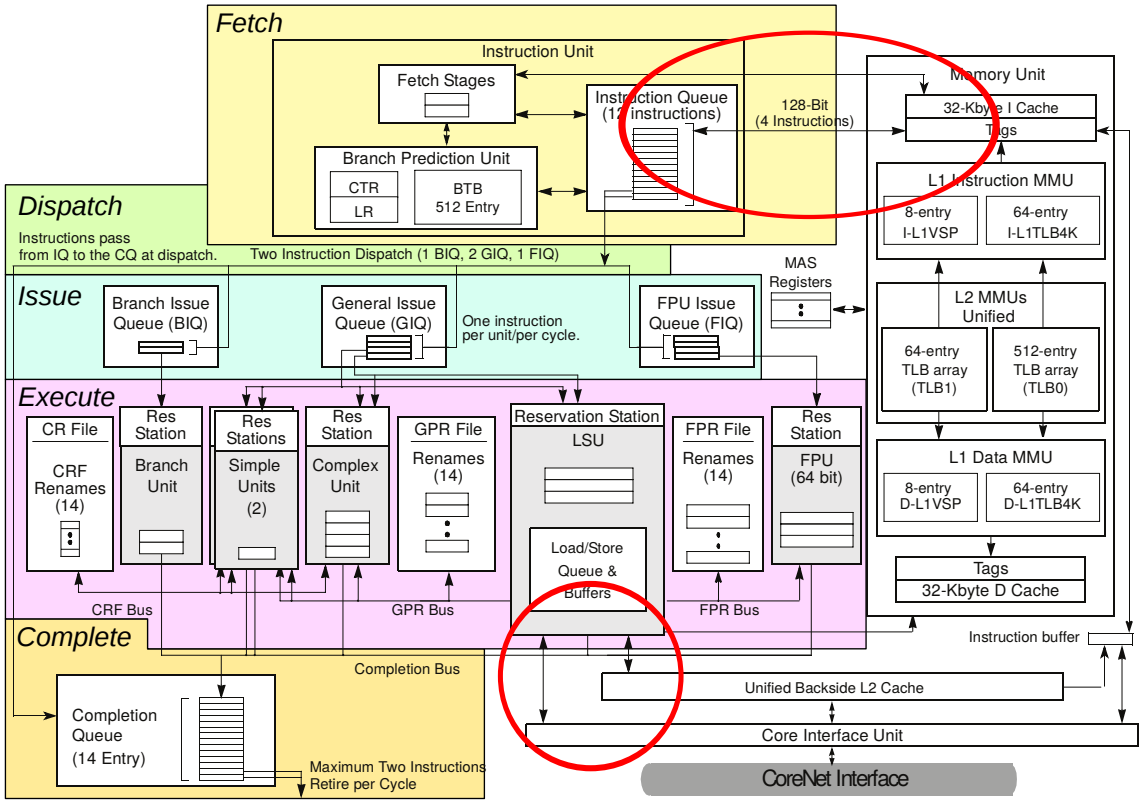


FIGURE 7.4 – Description du cœur e5500 issue du manuel de référence [43]

Enfin, nous regroupons dans certaines figures plusieurs types d'évènements, que nous notons ***event***.

7.3.1 Évènements associés au pipeline

Nous effectuons dans cette section une étude de l'architecture et du comportement du pipeline visant à définir une liste d'évènements qu'il génère. Cette liste doit couvrir les différentes opérations que peut effectuer un pipeline. Toutefois, nous cherchons à la rendre concise pour limiter la complexité des analyses ultérieures.

Nous commençons l'analyse du pipeline par des aspects architecturaux. La figure 7.4, qui reprend le schéma du cœur fourni dans le manuel de référence [43], illustre les différentes unités matérielles que l'on rencontre aux différents étages du pipeline (fetch, dispatch, issue, execute, complete). Les seules unités susceptibles d'interagir avec l'extérieur du pipeline sont la Load/Store Unit (LSU) et la Fetch Unit, qui réalise le chargement des instructions. Nous restreignons notre analyse à ces deux unités.

La "Fetch Unit" est une unité matérielle dont la structure est simple. Elle contient une file d'attente qui est remplie au fur et à mesure que des instructions sont chargées. Elle est proactive, i.e. cherche à charger autant d'instructions que possible dès qu'elle a

de la place dans sa file d'attente. Nous représentons son activité par un seul évènement, noté ***fetch(@)***.

La "Load/Store Unit" est une unité matérielle plus complexe. Elle est en charge de toutes les opérations autres qu'arithmétiques, logiques et d'évaluation de branchements. Pour identifier les différentes activités émises par la LSU, nous effectuons donc une analyse du jeu d'instructions PowerPC. Ce dernier est implémenté dans la version 2.06. Nous nous appuyons sur le standard [72] pour une description générale du jeu d'instructions, le manuel de référence de la famille de cœurs e500 [41, section 6.4], qui liste un sous-ensemble de ce jeu d'instructions supporté par la famille de cœurs e500, ainsi que le manuel de référence du cœur e5500 [43].

Certaines instructions sont purement internes et ne sont donc pas susceptibles d'initier des accès vers les ressources communes. Par exemple, on sait que les instructions portant sur les opérations arithmétiques et logiques ne peuvent pas générer d'accès aux ressources, dans la mesure où le jeu d'instructions PowerPC est de type load/store, ce qui signifie que les accès à l'espace adressable sont réalisés par des instructions dédiées. Nous écartons donc certaines classes d'instructions de notre analyse. La table 7.1 en donne la liste.

Pour les classes d'instructions restantes, nous appliquons la démarche suivante pour définir des évènements :

- Lorsque, de manière non ambigüe, l'instruction est interne, nous ne lui associons pas d'évènement.
- Lorsque, de manière non ambigüe, l'instruction est susceptible d'initier un accès vers les ressources communes, nous lui associons un évènement, qu'il faut définir. Il est intéressant d'associer à un même évènement plusieurs instructions dont le traitement et/ou l'activité électronique générée sont proches. Nous nous appuyons sur les éléments suivants :
  - Le manuel de référence du processeur donne de nombreux détails sur la manière dont certaines instructions sont implémentées. C'est par exemple le cas des instructions de gestion du cache, dont l'effet sur le cœur doit être bien compris par un programmeur pour qu'il puisse les utiliser de manière efficace. Il en va de même pour les barrières mémoire.
  - Certaines instructions sont implémentées de la même manière. Elles peuvent donc être associées au même évènement.
  - La connaissance du fonctionnement de périphériques comme les caches permet de raffiner en amont la définition de certains évènements dans le pipeline. C'est notamment le cas des opérations de gestion du cache.
  - La documentation des moniteurs de performances permet d'avoir des informations sur les opérations réalisées en interne dans le cœur. On peut en déduire les caractéristiques de certains évènements.
- Lorsqu'il y a ambiguïté, nous employons le protocole expérimental suivant :
  - L'intégralité des données et instructions nécessaires sont chargées et verrouillées dans un cache.
  - Un compteur matériel est configuré pour s'incrémenter à chaque phase d'accès sur l'interconnexion.

- Après activation du compteur, une instruction *isync* vient synchroniser le début de la phase de test.
- La phase de test exécute l'instruction en question. La terminaison de cette instruction est assurée par l'instruction *isync*.
- La valeur du compteur matériel indique l'émission ou non d'un accès, donc si l'instruction en question est liée à un évènement sensible.

En suivant cette démarche décrite précédemment, nous proposons les définitions suivantes pour les évènements générés lors de l'exécution d'instructions dans le pipeline :

load/store. Les évènements qui donnent lieu à une activité gérée par la MMU sont de type *load* ou *store*, selon la manière dont ils sont analysés vis-à-vis des droits d'accès. Cette liste est complétée par l'évènement correspondant aux chargements (fetch) d'instructions qui ne relève pas de l'analyse du jeu d'instructions.

snoop. Un évènement de ce type correspond à une activité non filtrée par la MMU. Cette activité se propage sur l'interconnexion dans un domaine de cohérence, qui peut contenir plusieurs cœurs. Ce type d'évènement peut perturber l'activité des cœurs et autres périphériques présents dans ce domaine de cohérence qui devront le traiter, et va également consommer des ressources sur l'interconnexion pour être propagé.

sync. Un évènement de ce type est propagé sur l'interconnexion sans être filtré par la MMU. Il correspond aux situations où une barrière est propagée sur l'interconnexion pour garantir l'ordre de propagation de certaines transactions.

Nous déduisons de cette analyse un ensemble d'évènements, qui est décrit dans la table 7.2. Cette table regroupe les évènements rencontrés dans chaque classe d'instructions. Une table plus complète associant chaque instruction à un évènement est présentée dans l'annexe A.

À partir de ces définitions, et en incluant l'évènement ***fetch(@)***, qui correspond au chargement des instructions, nous disposons d'une liste d'évènements représentant l'activité du pipeline. Nous reprenons ces évènements pour caractériser les activités de la MMU et des caches, présentées ci-après.

7.3.2 Évènements associés à la MMU

La MMU est chargée d'effectuer le lien entre l'espace d'adressage effectif, vu et manipulé par le logiciel, et l'espace d'adressage physique, qui est manipulé par le matériel. Elle s'appuie pour cela sur des mécanismes de traduction d'adresse qui s'appuient sur le principe de pagination. Ces mécanismes permettent également d'associer des droits d'accès en lecture, écriture ou exécution aux pages, puis de vérifier que les accès traités sont compatibles avec ces droits. Nous nous référons à la section 7.1.2 pour une description plus complète des fonctionnalités de la MMU.

TABLE 7.2 – Evènements associés à l'exécution d'instructions PowerPC

Classe d'instructions	Évènements rencontrés
Lectures et écritures dans l'espace adressable	***store(@, write)*** ***load (@, read)***
Barrières de synchronisation	***sync***
Gestion de la MMU	***sync*** ***snoop***
Gestion des caches	***store(@, allocate)*** ***load (@, flush)*** ***load (@, invalidate)*** ***load (@, touch)***
Interruptions inter-cœurs	***snoop***

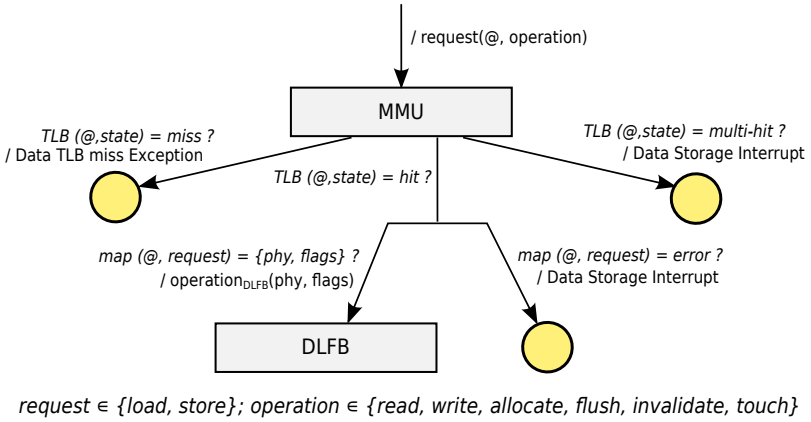


FIGURE 7.5 – Arbre d'activité d'une MMU traitant un évènement de type *load* ou *store*

La MMU est sollicitée par les évènements décrits dans la table 7.2, de types *load* et *store*, ainsi que l'évènement *fetch*. Le champ "operation" se réfère à toutes les opérations supportées par les caches, à savoir *read*, *write*, *allocate*, *flush*, *invalidate*, *touch*.

Nous définissons l'action de la MMU par les éléments suivants :

- Une fonction d'indexation des TLB, notée :
  $\boldsymbol{TLB} : A \times \Sigma \longmapsto \{hit, miss, multi\text{-}hit\}$
  Cette fonction formalise la recherche de règle de traduction valide dans les TLB, qui sont des mémoires caches internes à la MMU. Le mécanisme de traduction d'adresse est décrit dans le manuel de référence [43] à la section 6.2. Le fait d'avoir plusieurs entrées applicables à une adresse est un cas d'erreur, car la MMU n'a pas les moyens de distinguer celle qui doit s'appliquer.

- Une fonction de pagination, notée :
  Map : $A \times \{read, write, fetch\} \longmapsto \{(Phy \times WIMGE), error\}$
  Cette fonction formalise la relation entre l'adresse effective et l'adresse physique. Elle associe à une adresse effective, pour une opération donnée, une adresse physique ainsi que des attributs (flags) correspondant aux bits WIMGE (cf. section 7.1.2), qui seront par la suite associés aux évènements sortants.

La MMU peut lever les exceptions suivantes :

Data/Instruction TLB Miss. En cas d'absence de règle de traduction applicable, aucun mécanisme d'exploration de table des pages n'est prévu par le matériel. Une exception est donc levée vers le système d'exploitation ou l'hyperviseur, qui doit maintenir cette table des pages dans son espace mémoire.

Data/Instruction Storage Interrupt. Cette exception intervient lorsque plusieurs règles de traduction sont applicables, ou lorsqu'il y a une violation de droits d'accès lors de l'application d'une règle de traduction.

L'activité de la MMU est représentée sous forme d'arbres d'activité sur la figure 7.5 pour des évènements de type *load* ou *store*. Cette activité est ensuite propagée dans la mémoire hiérarchique, qui dans le cœur contient plusieurs niveaux de caches.

7.3.3 Évènements associés aux caches

On rencontre deux niveaux de caches (L_1 et L_2) sur le cœur e5500, dont le niveau L_1 qui est coupé en deux pour les données et les instructions. Ces caches sont précédés de deux buffers de données et d'instructions de très petite taille (320 octets chacuns) pour les données et les instructions, appelés ILFB et DLFB [37]. Ces buffers se comportent d'une manière similaire aux caches dans la plupart des cas. Ils contiennent des *lignes de caches*, d'une taille de 64 octets.

Par ailleurs, nous supposons que tous les caches sont activés.

Comme expliqué lors de l'analyse du jeu d'instructions, un cache peut être sollicité pour les opérations suivantes :

Lecture (read), écriture (write). Lecture et écriture de données depuis et dans le cache, à l'initiative du logiciel, i.e. du pipeline du point de vue électronique, ou d'un autre niveau de cache.

Comme illustré sur la figure 7.6, le DLFB est en charge de fusionner les données écrites, qui portent en général sur des tailles allant de 1 à 8 octets, avec les lignes de caches qu'il fait remonter depuis les caches L_1D et L_2, ou à défaut depuis la mémoire principale. Ainsi, les écritures dans les caches L_1D et L_2 ne correspondent qu'à des lignes de caches.

Allocation (allocate). Allocation d'une ligne de cache à une certaine adresse, et mise à zéro du contenu du cache associé.

37. Instruction/Data Line Fill Buffe

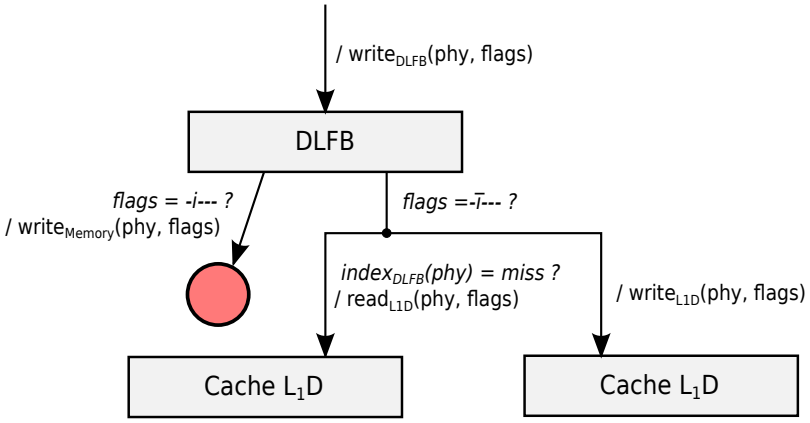


FIGURE 7.6 – Arbre d'activité du DLFB traitant un évènement de type *write*

Flush. Écriture de la ligne de cache visée dans la mémoire principale, suivie de l'invalidation de cette ligne dans tous les niveaux de caches. Les instructions de "flush vers le niveau inférieur" ne sont pas implémentées dans les cœurs e5500.

Pré-chargement (touch). Demande au cache de charger certaines lignes de cache, qui seront probablement utilisées à l'avenir. Ce type d'opération est vu comme un indice (hint) que le logiciel donne au cache pour améliorer ses performances. Les instructions correspondantes terminent sans que l'accès en question ne soit terminé.

Invalidation (invalidate). Invalidation de la ligne de cache correspondante dans tous les niveaux de cache.

Définitions communes à tous les caches

Comme pour le premier cas d'étude portant sur un modèle simple de processeur, nous définissons l'action du cache par les fonctions suivantes :

- Une fonction ***d'indexation***, définie de la manière suivante :
  index : $Phy \longmapsto \{M, E, S, I\}$. Cette fonction associe à toute adresse physique un état dans le cache correspondant au protocole de cohérence MESI, implémenté sur le P5040. Nous assimilons l'absence de la donnée dans le cache à l'état 'I'.
- Une fonction de ***remplacement***, qui définit les lignes de caches sélectionnées en tant que victimes :
  victim : $Phy \longmapsto Phy$.

Définitions spécifiques au cache L1D

L'état du cache L_1D est complété par le mode courant, défini comme suit :

mode $= \{WS, \neg WS\}$. L'abréviation 'WS' signifie 'Write Shadow'. Dans ce mode de fonctionnement, toute écriture est automatiquement propagée vers le cache L_2. Lorsque

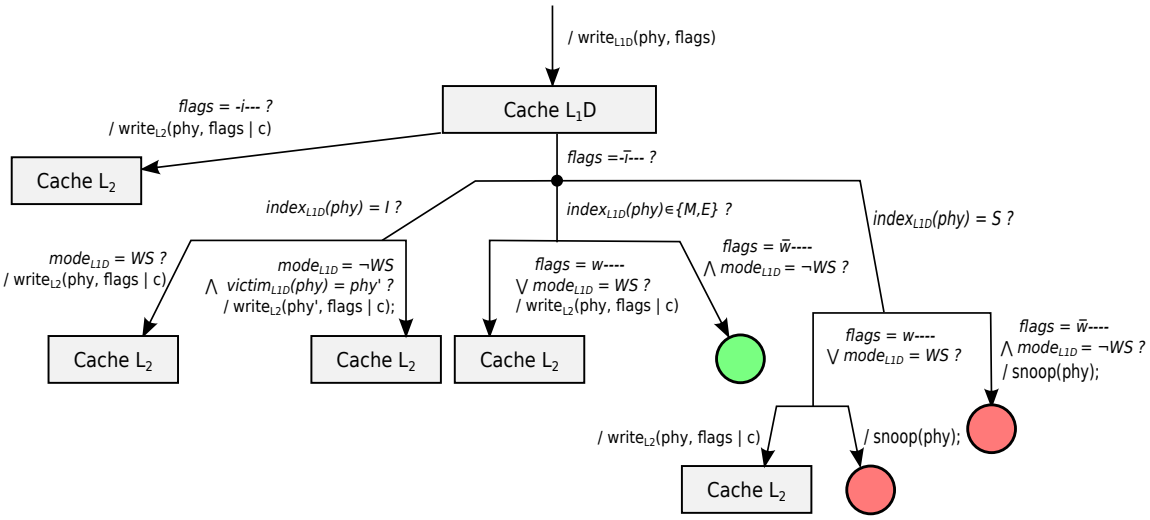


FIGURE 7.7 – Arbre d'activité du cache L_1D traitant un évènement de type *write*

la ligne de cache écrite est présente dans le cache L_1D, celle-ci est mise à jour. Dans le cas contraire, la ligne n'est pas remontée dans le cache L_1D, et est directement écrite dans le cache L_2. Ce phénomène est illustré sur la figure 7.7, qui représente l'arbre d'activité du cache L_1D traitant un évènement de type *write*.

Définitions spécifiques au cache L2

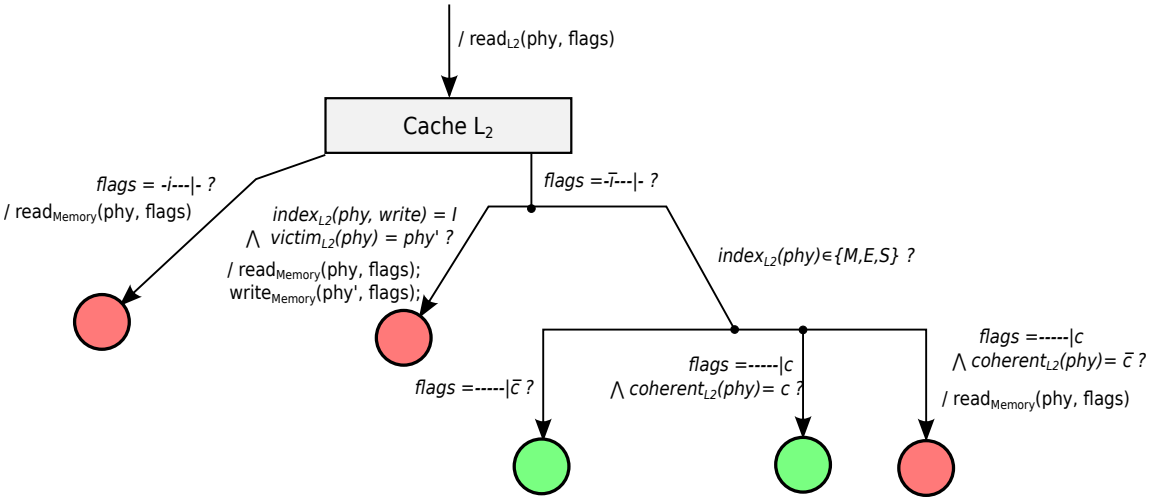
Le cache L_2 est unifié. Toutefois, données et instructions ne sont pas traitées de la même manière. L'architecture du cache L_2 est dite *dynamic harvard*, et à ce titre associe les données à un mode de fonctionnement *cohérent*, et les instructions à un mode *non cohérent*. On peut passer d'un état cohérent à un état non cohérent, mais pas l'inverse.

Nous formalisons cet aspect à travers la fonction $\boldsymbol{coherent_{L2}} : Phy \longmapsto \{c, \overline{c}\}$, qui associe à une ligne de cache un état cohérent ou non cohérent.

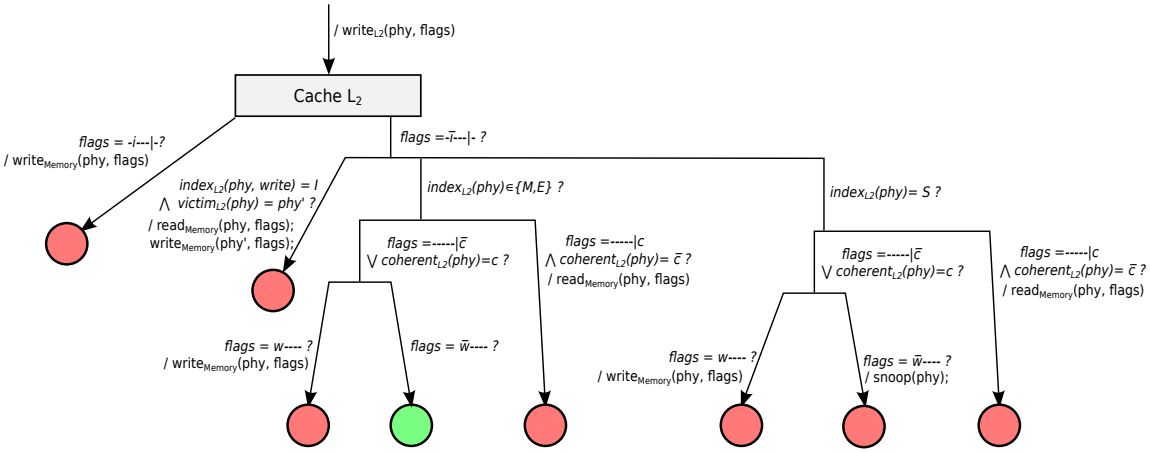
7.3.4 Construction des arbres d'activité au niveau du cœur

Nous disposons d'une caractérisation de chaque composant du cœur sous forme d'arbres d'activités, et des évènements associés. Nous les composons pour construire les arbres d'activités au niveau du cœur.

L'application de cette démarche donne des arbres d'activités complexes, mais sur lesquels de nombreuses branches mènent à des évènements sensibles en permanence. Il est donc intéressant d'imposer des restrictions sur le comportement du logiciel afin de couper les branches les plus simples. L'arbre d'activités peut ainsi être réduit à une forme manipulable. Ces restrictions doivent cependant être assurées par des règles de conception applicables au logiciel de contrôle.



(a) Arbre d'activité du cache L_2 traitant un évènement de type *read*



(b) Arbre d'activité du cache L_2 traitant un évènement de type *write*

FIGURE 7.8 – Principaux arbres d'activité du cache L_2

Nous pouvons constater que certaines configurations amènent systématiquement le cœur à émettre des accès vers les ressources communes. Nous nous intéressons aux cas suivants :

- Les accès en lecture et écriture dans des pages mémoire non cacheables, i.e. pour lesquelles l'attribut 'I' vaut zéro.
- Les accès en écriture dans des pages "write through", i.e. pour lesquelles l'attribut 'W' vaut un.
- Les accès en écriture à des données présentes dans les caches dans l'état 'S' (Shared).

Il est possible de rendre ces situations inatteignables en posant des règles de conception sur le logiciel de contrôle.

Gestion des accès à la mémoire non cacheable

Le cas des accès à des pages non cacheables peut être traité de la manière suivante :

- Les pages mémoires accessibles au logiciel utile sont toujours cacheables.
- Les pages mémoires non cacheables propres au logiciel de contrôle ne sont accédées que dans des fenêtres temporelles prédéfinies.
- Si malgré tout le logiciel utile a besoin d'utiliser de la mémoire non cacheable, le logiciel de contrôle peut lui en mettre à disposition à condition qu'il émule les instructions de lecture et d'écriture dans ces plages mémoires.

Gestion des accès aux pages "write through"

Comme pour la mémoire non cacheable, il est possible de traiter le cas des accès à des pages "write through" de la manière suivante :

- Les pages mémoires dédiées au logiciel utile qui sont configurées en mode "Write Back", i.e. une écriture servie par le cache n'est pas propagée vers la mémoire principale.
- Le logiciel de contrôle n'utilise pas de mémoire "write through" pour sa propre exécution.
- De la mémoire "write through" peut toutefois être proposée au logiciel utile, mais accédée directement en lecture seulement. Les accès en écriture sont effectués par le logiciel de contrôle qui émule les instructions d'écriture.

Gestion des données partagées dans les caches

Le cas des accès en écriture à des données en état 'S' (Shared) peut être également rendu impossible. En effet, une donnée se trouve en cache dans l'état 'S' lorsqu'elle est partagée entre plusieurs cœurs, et que les mécanismes de cohérence de caches sont activés. Les solutions que l'on rencontre dans la littérature vont dans le sens de l'absence, ou de la désactivation des mécanismes de cohérence de caches. Sur le P5040, ces mécanismes peuvent être désactivés de la manière suivante :

- Au niveau du cœur, le bit 'M' (Memory Coherency Required) peut être mis à zéro pour chaque page définie dans les TLB.
- Au niveau du processeur, la cohérence de caches est gérée dans des domaines de cohérences, qui associent à des plages d'adresses physiques des cœurs et périphériques susceptibles de contenir des données en caches. La cohérence de caches est donc restreinte à ces périphériques. Définir des domaines de cohérences vides a pour conséquence d'inhiber les mécanismes de cohérence de caches.

On notera cependant que la désactivation de la cohérence de caches implique que le logiciel doit gérer lui même la cohérence des données partagées entre les cœurs. Le problème se posera sur le logiciel utile si l'on cherche à déployer sur plusieurs cœurs des tâches partageant le même espace d'addressage. Dans notre contexte, notre système est partitionné spatialement, ce n'est pas le cas. Le logiciel utile est indépendant sur chaque cœur. Toutefois, la question de la parallélisation du logiciel utile est une perspective intéressante.

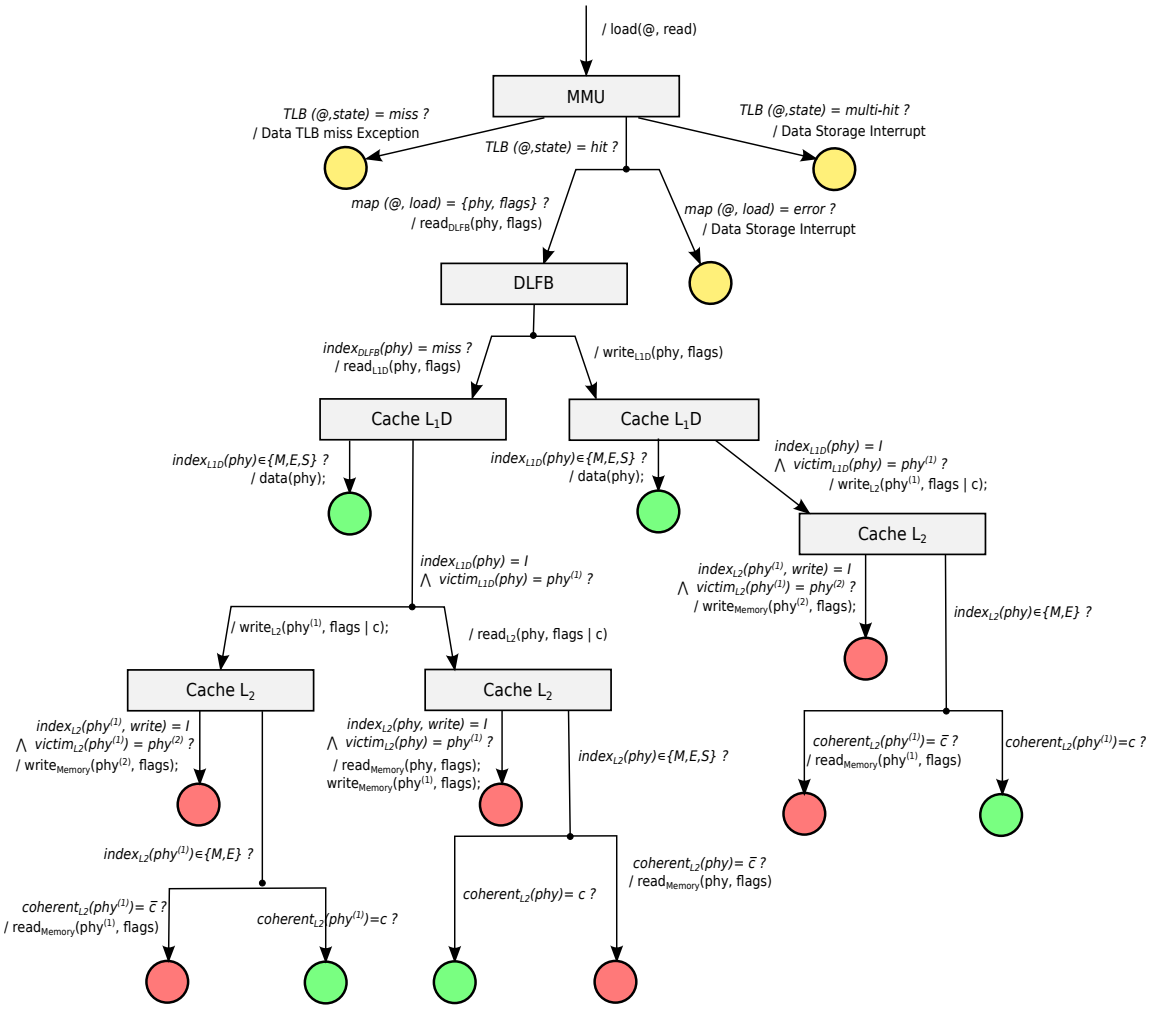


FIGURE 7.9 – Arbre d'activité du cœur e5500 sur un évènement de type *load*

Du point de vue du logiciel de contrôle, nous posons la règle suivante :

- Les seules données partagées et cacheables sont accessibles en lecture seule
- Toute donnée partagée accessible en écriture est non cacheable.

Les règles de conception que nous avons proposé permettent de réduire la complexité des arbres d'activité à un niveau qui permet leur analyse. Ainsi, l'arbre d'activités du cœur pour un évènement *load(@, read)*, émis lorsqu'une instruction de lecture est exécutée, est représenté sur la figure 7.9. L'arbre d'activités pour un évènement de type *fetch(@)* est représenté de la même manière sur la figure 7.10. Pour la majorité des autres opérations, on obtient des arbres similaires, en particulier pour les évènements de type *store(@, write)*, qui sont traités de la même manière dans le DLFB, la seule différence consistant en une fusion des écritures entre les données descendantes du pipeline et les données montantes des caches ou de la mémoire principale.

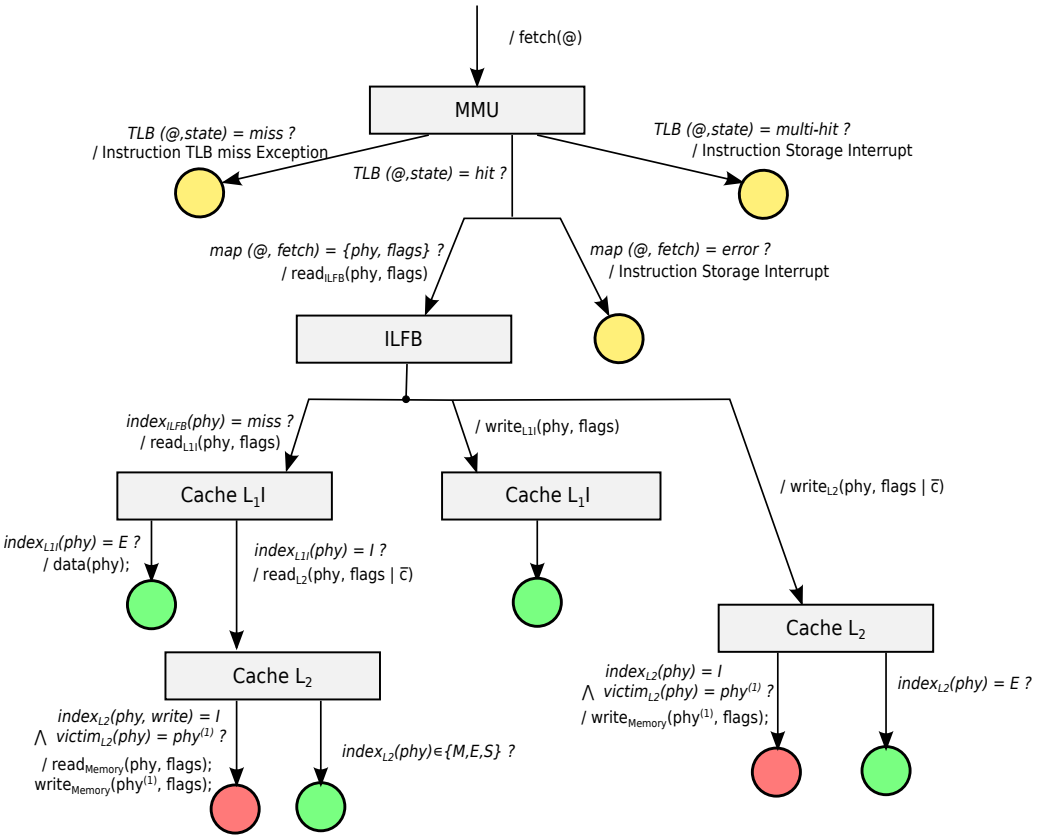


FIGURE 7.10 – Arbre d'activité du cœur e5500 sur un évènement de type *fetch*

Nous relevons toutefois une exception pour les opérations de type *load(@, flush)*, pour lesquelles l'arbre d'activités est représenté sur la figure 7.11.

La connaissance des arbres d'activités pour le cœur e5500 nous permet de conclure sur la première phase de notre méthode. Nous sommes partis d'une analyse de l'architecture du cœur e5500, et du jeu d'instructions PowerPC tel qu'il est implémenté. Cette analyse nous a permis d'identifier un ensemble d'évènements sensibles, dont nous avons formalisé les relations à travers des arbres d'activité, d'abord sur chaque composant, puis sur tout le cœur.

Pour limiter la complexité de l'identification des évènements et la construction des arbres d'activité, nous avons introduit plusieurs règles de conception qui s'appliquent au logiciel de contrôle, et que subit le logiciel utile. Ces règles portent sur :

- La gestion des données partagées entre les cœurs par le logiciel de contrôle. Ainsi, toute donnée accessible en écriture par plusieurs cœurs doit être non cacheable.
- La désactivation de mécanismes de cohérence de caches. Nous pouvons configurer des mécanismes au niveau du cœur et du processeur qui inhibent le traffic de cohérence de caches.
- La gestion de la mémoire non cacheable et de la mémoire "write through". Ce type de mémoire est interdit en accès direct pour le logiciel utile, mais peut être

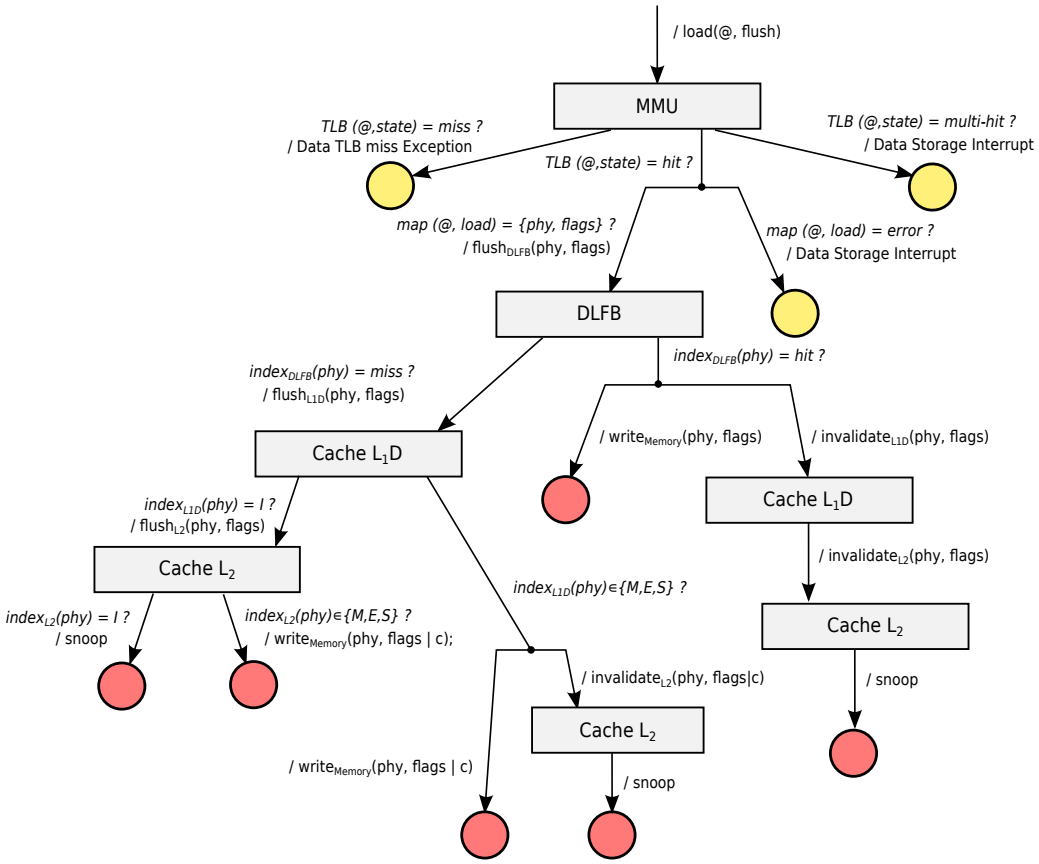


FIGURE 7.11 – Arbre d'activité du cœur e5500 sur un évènement de type *flush*

simulé par le logiciel de contrôle.

Nous développons dans la section suivante la deuxième phase de notre méthode, qui porte sur la contrôlabilité des évènements, ainsi que l'élimination des évènements non contrôlables.

7.4 Classification des évènements sensibles

Nous disposons d'un ensemble d'évènements sensibles définis sur le cœur e5500, ainsi qu'une formalisation de son comportement à travers un ensemble d'arbres d'activités. Dans cette section, nous développons la deuxième étape de notre démarche, qui porte sur l'étude de la contrôlabilité des évènements sensibles.

Comme défini dans le chapitre précédent, un évènement est contrôlable lorsque le cœur lève systématiquement une exception dirigée vers le logiciel de contrôle suite à l'occurrence de cet évènement. Lorsque cette exception est levée seulement dans certaines configurations du cœur, on dit que l'évènement est partiellement contrôlable.

Nous cherchons à établir la première hypothèse d'application du théorème de contrôlabilité, à savoir que tout évènement sensible est contrôlable ou éliminé par la configuration du cœur. Nous abordons cette analyse en deux temps. Dans la section 7.4.1 nous effectuons une classification des évènements précédemment identifiés sur le cœur. Nous obtenons une liste d'évènement contrôlables, ainsi qu'une liste d'évènements partiellement ou non contrôlables. Pour ces derniers nous définissons dans la section 7.4.2 un invariant qui les neutralise, i.e. élimine les cas où ces évènements sont sensibles.

7.4.1 Classification des évènements sensibles

Nous disposons des arbres d'activités du cœur e5500 pour les différents évènements que peut émettre le pipeline en exécutant le logiciel. À partir de ces arbres, nous pouvons appliquer les règles de classification d'évènements que nous avons introduits dans le chapitre précédent.

Les règles de classification pour la sensibilité sont les suivantes :

- Un évènement est sensible à une instruction lorsqu'il ne descend que d'évènements émis par le pipeline lors de l'exécution d'une instruction. Dans notre cas, on dira qu'un évènement est sensible à une instruction s'il ne descend que d'évènements de type *load*, *store*, *snoop* et *sync*.
- Un évènement est sensible à la configuration s'il possède parmi ses descendants dans l'arbre d'activités des évènements sensibles (rouges) et non sensibles (verts).
- Un évènement est non sensible si tous ses descendants sont non sensibles (verts)
- À défaut, l'évènement est sensible en permanence.

Les règles de classification pour la contrôlabilité sont les suivantes :

- Un évènement est contrôlable lorsque tous ses descendants dans l'arbre d'activité correspondent à la levée d'exceptions (oranges).
- Un évènement est partiellement contrôlable lorsqu'il possède des descendants correspondant à des exceptions (oranges) et d'autres correspondant à des évènements non sensibles (verts) ou sensibles (rouges).
- À défaut, un évènement est non contrôlable.

En appliquant ces règles, nous obtenons une classification des évènements que nous avons identifié sur le cœur e5500. Nous donnons une version détaillée de cette classification sur les évènements issus du pipeline sur la table 7.3. Pour certains évènements, nous avons distingué l'occurrence dans le logiciel utile de celle dans le logiciel de contrôle. Cela permet d'identifier les évènements contrôlables parce qu'ils résultent d'instructions réservées. Le logiciel de contrôle pourra donc les traiter lorsqu'ils surviendront dans le logiciel utile. Pour sa propre exécution, il faudra faire en sorte d'éviter de les générer, ou alors il devra les contrôler explicitement.

Lorsqu'un évènement n'est pas contrôlable, on peut envisager de l'éliminer en posant une règle de conception. C'est par exemple le cas des évènements de type *sync* générés

par les instructions assembleur *sync* et *mbar*. Nous traitons le cas de ces évènements par les règles de conception suivantes :

Règle de conception 1. *Le logiciel utile ne doit pas utiliser les instructions **sync** et **mbar**. Le logiciel de contrôle peut les utiliser dans des fenêtres temporelles maîtrisées. Il est recommandé d'utiliser d'autres instructions lorsque c'est possible.*

Nous donnons également une classification plus succinte pour les autres évènements identifiés sur le cœur sur la table 7.4.

On peut constater qu'en appliquant les règles de classification ci-dessus, les seuls évènements contrôlables ou partiellement contrôlables sont émis par le pipeline. Toute la stratégie de contrôle portera donc sur ces derniers. Les autres évènements n'étant pas contrôlables, nous cherchons à les neutraliser par un invariant. Leur neutralisation permet également de traiter les cas d'évènements partiellement contrôlables émis par le pipeline. Nous développons ces aspects dans la section suivante.

Discussion sur les évènements de type *load(@, flush)*

Nous avons relevé un cas particulier, portant sur l'évènement de type *load(@, flush*. Cet évènement est déclenché par les instructions *dcbf* et *dcbfep* (data cache block flush [external PID]). La première est autorisée en mode utilisateur, et est donc utilisable par le logiciel utile. Nous avons classé cet évènement comme non contrôlable. Pourtant, la transaction est traitée par la MMU, et est donc susceptible de lever des exceptions de type "Data TLB Miss" et "Data Storage Interrupt". On pourrait donc exhiber des cas où cet évènement est contrôlable. Nous avons choisi de ne pas le faire car cela limiterait la stratégie de contrôle des autres évènements de type *load* et *store*. Ces derniers sont pour la plupart sensibles à la configuration, ce qui ouvre la possibilité d'une stratégie de contrôle par neutralisation, qui est une piste intéressante. Chercher à contrôler les évènements de type *load(@, flush)* élimine de fait cette stratégie de contrôle pour les autres évènements.

Nous posons donc les règles de conception suivantes, applicable au logiciel utile :

Règle de conception 2. *Le logiciel utile ne doit pas utiliser l'instruction **dcbf**. Le logiciel de contrôle ne peut l'utiliser que dans des fenêtres temporelles maîtrisées.*

Nous développons dans la section suivante la construction de l'invariant.

7.4.2 Construction de l'invariant

Nous avons effectué dans la section précédente la classification des évènements sensibles que nous avions identifié sur le cœur e5500. Cette classification, couplée avec l'étude de la contrôlabilité, a montré que la plupart des évènements sensibles, sont partiellement contrôlables, voire non contrôlables. Pour satisfaire les conditions d'application du théorème de contrôlabilité, il est nécessaire d'inhiber les cas où ces évènements sont sensibles.

TABLE 7.3 – Classification des évènements émis par le pipeline

Évènement	Sensible à une instruction	Sensible à la configuration	Non sensible	Contrôlable	Partiellement contrôlable	Non contrôlable	Exceptions associées
fetch(@)		✓			✓		Inst. TLB Miss, Inst. Storage Interrupt
load(@, read)	✓	✓			✓		Data TLB Miss, Data Storage Interrupt
load(@, touch)	✓	✓			✓		Data TLB Miss, Data Storage Interrupt
load(@, flush)	✓					✓*	
load(@, invalidate) (log. utile)	✓			✓			Privilege Exception
load(@, invalidate) (log. contrôle)	✓					✓	
store(@, write)	✓	✓			✓		Data TLB Miss, Data Storage Interrupt
store(@, allocate)	✓				✓		Data TLB Miss, Data Storage Interrupt
snoop (log. utile) → *tlbivax, msgsnd*	✓			✓			Privilege Exception
snoop (log. ctrl) → *tlbivax, msgsnd*	✓					✓	
sync → *mbar, sync*	✓					✓*	
sync (log. utile) → *tlbsync*	✓			✓			Privilege Exception
sync (log. ctrl) → *tlbsync*	✓					✓	

* La contrôlabilité de cet évènement fait l'objet d'une discussion dans cette section.

TABLE 7.4 – Classification des évènements du cœur e5500 (hors pipeline)

Évènement	**Sensible à une instruction**	**Sensible à la configuration**	**Sensible en permanence**	**Non sensible**	**Contrôlable**	**Partiellement contrôlable**	**Non contrôlable**
Évènements associés au DLFB et aux caches L_1D et L_2							
read$_{DLFB/L1D/L2}$ *(phy, flags)*	✓	✓					✓
write$_{DLFB/L1D/L2}$ *(phy, flags)*	✓	✓					✓
allocate$_{DLFB/L1D/L2}$ *(phy, flags)*	✓	✓					✓
flush$_{DLFB/L1D/L2}$ *(phy, flags)*	✓						✓
touch$_{DLFB/L1D/L2}$ *(phy, flags)*	✓	✓					✓
invalidate$_{DLFB/L1D/L2}$ *(phy, flags)*	✓						✓
Évènements associés au cache L_1I et au ILFB							
read$_{ILFB/L1I}$*(phy, flags)*		✓					✓
touch$_{ILFB/L1I}$ *(phy, flags)*	✓	✓					✓
invalidate$_{ILFB/L1I}$ *(phy, flags)*	✓						✓
write$_{ILFB/L1I}$ *(phy)*				✓	N/A		
Évènements émis vers les ressources communes							
read$_{Memory}$*(phy, flags)*			✓				✓
write$_{Memory}$ *(phy, flags)*			✓				✓

La plupart de ces évènements sont sensibles à la configuration, et certains sont indispensables au fonctionnement du logiciel. Nous cherchons donc à définir un invariant qui restreigne l'occurrence de ces évènements aux cas où ils sont neutres. À cet effet, nous appliquons la démarche d'analyse des arbres d'activités du cœur, que nous avons définie dans le chapitre précédent.

Cette démarche consiste à suivre les chemins de l'arbre d'activité menant à des évènements sensibles en permanence, et à transcrire ces chemins sous forme de clauses du type $\alpha = \alpha_1 \bigwedge ... \bigwedge \alpha_n$, chaque terme α_i correspondant à la garde d'un nœud sur l'arbre. On obtient ainsi un ensemble de clauses $\bigwedge_{i \in I} \{\alpha^i\}$. Une clause β invalidant chaque clause α^i constitue un invariant. La construction de l'invariant doit prendre en compte que ce dernier sera mis en place par le logiciel de contrôle.

Construction de l'invariant à partir des arbres d'activités

Pour construire l'invariant, nous avons proposé dans le chapitre précédent des clauses du type $\alpha_1 \bigwedge ... \bigwedge \alpha_n \Rightarrow \beta$ comme point de départ.

Dans notre cas, nous nous basons sur les arbres d'activités du cœur sur la base des évènements *load(@,read)* (aussi valable pour les évènements de type *store*), et *fetch(@)*. Ces arbres sont représentatifs de l'activité du cœur.

Évènements de type *load/store*

Les clauses obtenues sont les suivantes, en parcourant l'arbre de la figure 7.9 de gauche à droite. Nous avons regroupé les clauses par paires de cas d'émission, ce qui explique que nous ayons trois clauses pour six cas d'émission.

$$\begin{aligned}
&\forall @ \in A,\ state \in \Sigma,\ op \in \{load,\ store\},\\
&\qquad TLB(@, state) = hit \ \wedge\ map(@, op) = \{phy,\ flags\}\\
&\quad \wedge\ index_{DLFB}(phy) = miss \ \wedge\ index_{L1D}(phy) = I \ \wedge\ victim_{L1D}(phy) = phy^{(1)}\\
&\qquad \Rightarrow index_{L2}(phy^{(1)}) \in \{M,\ E\} \ \wedge\ coherent_{L2}(phy^{(1)}) = c
\end{aligned} \tag{1.1}$$

$$\begin{aligned}
&\forall @ \in A,\ state \in \Sigma,\ op \in \{load,\ store\},\\
&\qquad TLB(@, state) = hit \ \wedge\ map(@, op) = \{phy,\ flags\}\\
&\quad \wedge\ index_{DLFB}(phy) = miss \ \wedge\ index_{L1D}(phy) = I \ \wedge\ victim_{L1D}(phy) = phy^{(1)}\\
&\qquad \Rightarrow index_{L2}(phy) \in \{M,\ E\} \ \wedge\ coherent_{L2}(phy) = c
\end{aligned} \tag{1.2}$$

$$\begin{aligned}
&\forall @ \in A,\ state \in \Sigma,\ op \in \{load,\ store\},\\
&\qquad TLB(@, state) = hit \ \wedge\ map(@, op) = \{phy,\ flags\}\\
&\quad \wedge\ index_{L1D}(phy) = I \ \wedge\ victim_{L1D}(phy) = phy^{(1)}\\
&\qquad \Rightarrow index_{L2}(phy^{(1)}) \in \{M,\ E\} \ \wedge\ coherent_{L2}(phy^{(1)}) = c
\end{aligned} \tag{1.3}$$

Évènement de type *fetch* :

Les clauses obtenues sont les suivantes, en parcourant l'arbre de la figure 7.10 de gauche à droite :

$$\forall @ \in A,\ state \in \Sigma,\ TLB(@, state) = hit \wedge map(@, fetch) = \{phy,\ flags\} \\ \wedge index_{ILFB}(phy) = miss \wedge index_{L1I}(phy) = I \\ \Rightarrow index_{L2}(phy) \in \{M,\ E,\ S\} \tag{1.4}$$

$$\forall @ \in A,\ state \in \Sigma,\ TLB(@, state) = hit \wedge map(@, fetch) = \{phy,\ flags\} \\ \wedge index_{ILFB}(phy) = miss \\ \Rightarrow index_{L2}(phy) \in \{M,\ E,\ S\} \tag{1.5}$$

Informellement, ces clauses signifient que tout accès initié par le pipeline, servi par la MMU, mais pas par les DLFB/ILFB ni par le cache L_1 sera servi par le cache L2 (clauses (1.2), (1.4) et (1.5)). En outre, les opérations de remplacement mises en œuvre par les caches ne génèrent pas d'écriture vers la mémoire principale (clauses (1.1) et (1.3)).

Les clauses que nous obtenons constituent un invariant, mais sont difficilement implémentables par le logiciel de contrôle, dans la mesure où elles contiennent des termes qui ne sont pas connus du logiciel, comme l'état des DLFB/ILFB. Nous entrons donc dans une seconde phase, consistant à réduire l'invariant de telle sorte à le rendre implémentable.

Réduction de l'invariant

Nous avons construit un premier invariant en analysant les arbres d'activités du cœur e5500. Cependant, on ne peut pas écrire de logiciel qui implémente directement cet invariant. Nous cherchons donc à le simplifier.

Notre invariant est une conjonction de clauses du type $\alpha_1 \wedge ... \wedge \alpha_n \Rightarrow \neg\beta$. Nous avons proposé dans le chapitre précédent les règles de réduction suivantes :

- Toute clause de l'invariant peut être remplacée par une clause du type $\bigwedge_{i \in I}\{\alpha_i\} \Rightarrow \beta$ pour $I \subset [1 \cdots n]$
- Si pour un terme α_i donné on dispose d'un terme γ tel que la propriété $\alpha_i \Rightarrow \gamma$ soit vraie, alors γ peut être substitué à α_i dans toute clause où ce dernier apparaît. De même, si la propriété $\gamma \Rightarrow \beta$ est vraie, alors γ peut être substitué à β.

On pourra remarquer que la clause (1.5) implique la clause (1.4), et que cette dernière peut donc être retirée. De même, certaines clauses contiennent des variables libres, par exemple la variable $phy^{(1)}$ dans la clause (1.2), que nous pouvons également retirer.

Les clauses font intervenir des propriétés sur l'état des composants comme la MMU, les caches L_1 et L_2 ou encore les ILFB et DLFB. Pour ces derniers, le logiciel n'a aucun moyen de connaître avec certitude leur contenu. On peut donc retirer les termes impliquant leur état dans les clauses, ce qui a pour effet de fusionner les clauses (1.1) et (1.3).

Nous obtenons ainsi les clauses suivantes :

$$\begin{aligned}\forall @ \in A,\ state \in \Sigma,\ op \in \{load,\ store\},\\ TLB(@, state) = hit \ \wedge\ map(@, op) = \{phy,\ flags\}\\ \wedge\ index_{L1D}(phy) = I \ \wedge\ victim_{L1D}(phy) = phy^{(1)}\\ \Rightarrow index_{L2}(phy^{(1)}) \in \{M,\ E\} \ \wedge\ coherent_{L2}(phy^{(1)}) = c\end{aligned} \tag{2.1}$$

(anc. 1.1 et 1.3)

$$\begin{aligned}\forall @ \in A,\ state \in \Sigma,\ op \in \{load,\ store\},\\ TLB(@, state) = hit \ \wedge\ map(@, op) = \{phy,\ flags\}\\ \wedge\ index_{L1D}(phy) = I\\ \Rightarrow index_{L2}(phy) \in \{M,\ E\} \ \wedge\ coherent_{L2}(phy) = c\end{aligned} \tag{2.2}$$

(anc. 1.2)

$$\begin{aligned}\forall @ \in A,\ state \in \Sigma,\ TLB(@, state) = hit \ \wedge\ map(@, fetch) = \{phy,\ flags\}\\ \Rightarrow index_{L2}(phy) \in \{M,\ E,\ S\}\end{aligned} \tag{2.3}$$

(anc. 1.5 et 1.4)

Au terme de cette étape intermédiaire, nous obtenons un invariant qui est plus simple à manipuler, et dont on peut voir les orientations.

Ainsi, la clause (2.1) porte sur la présence des données sélectionnées par la logique de remplacement du cache L_1D pour être déchargées dans le cache L_2. Ici, le fait que l'accès initial, qui a déclenché le déchargement, est référencé par la MMU est de peu d'intérêt. Nous réduisons donc cette clause pour ne conserver que la mention de $phy^{(1)}$, ce qui donne la clause (3.1).

La clause (2.2) signifie que tout accès en données référencé par la MMU et non servi par le cache L_1D est servi par le cache L_2 dans un état cohérent. L'état d'une donnée est cohérent si et seulement si elle n'a pas été accédée auparavant comme une instruction, suite à un évènement de type *fetch*. Nous avons discuté dans le chapitre précédent la question de la pertinence de maîtriser le contenu du cache L_1D , en y verrouillant des données, par rapport à une situation où il serait laissé libre. Les principaux arguments pour et contre sont les suivants :

+ Avec un cache L_1D contrôlé, on a la possibilité de choisir dans quel cache les données sont placées, et d'avoir des contenus de caches L_1D et L_2 disjoints, ce qui augmente l'espace disponible.
+ Il est plus facile d'implémenter la clause (3.1) de l'invariant avec un cache L_1D dont on maîtrise le contenu, et donc la logique de remplacement.
- Un cache L_1D non contrôlé laisse la possibilité au matériel d'utiliser le cache comme un accélérateur matériel du cache L_2 .
- Contrôler le cache L_1D et le cache L_2 avec du logiciel est plus complexe que le cache L_2 seul.

Dans notre cas, nous choisissons de laisser le contrôle du cache L_1D au matériel, et de concentrer le contrôle uniquement sur le cache L_2 . Concrètement, cela signifie que nous effaçons le terme $index_{L1D}$ dans la clause (2.2). Le résultat est noté dans la clause (3.2).

Enfin, nous reprenons la clause (2.3) sans modification en la notant (3.3). L'invariant final que nous obtenons est le suivant :

$$\forall phy \in Phy,\ victim_{L1D}(phy) = phy^{(1)} \Rightarrow index_{L2}(phy^{(1)}) \in \{M,\ E\} \wedge coherent_{L2}(phy^{(1)}) = c \quad (3.1)$$
(anc. 2.1)

$$\forall @ \in A,\ state \in \Sigma,\ op \in \{load,\ store\},\ TLB(@, state) = hit \wedge map(@, op) = \{phy,\ flags\} \Rightarrow index_{L2}(phy) \in \{M,\ E\} \wedge coherent_{L2}(phy) = c \quad (3.2)$$
(anc. 2.2)

$$\forall @ \in A,\ state \in \Sigma,\ TLB(@, state) = hit \wedge map(@, fetch) = \{phy,\ flags\} \Rightarrow index_{L2}(phy) \in \{M,\ E,\ S\} \quad (3.3)$$
(anc. 2.3)

Nous disposons d'un invariant, combinant de trois clauses, qui permet d'éliminer les occurrences des évènements sensibles qui ne sont pas contrôlables. Si cet invariant est mis en place et n'est pas altéré par le logiciel utile, nous remplissons la première hypothèse d'application du théorème de contrôlabilité.

Contrôle de l'état de cohérence des données dans le cache L_2

Dans l'invariant que nous avons défini, la clause (3.2) précise que tout accès en lecture ou en écriture doit être servi par une donnée cohérente. Si on peut contrôler la présence ou non d'une donnée dans le cache L_2 par des mécanismes de verrouillage, contrôler son mode de cohérence est plus difficile. Il dépend en effet de l'historique d'accès à cette donnée, qui n'a pas vocation à être contrôlé lorsque cette dernière est présente en cache.

Une ligne de cache dans un état cohérent a vocation à être accédée par des requêtes de type *load* et *store*. Une ligne de cache dans un état incohérent a vocation à être accédée par des requêtes de type *fetch*. Le passage de l'état cohérent à l'état incohérent n'entraîne pas d'émission d'accès, contrairement au passage de l'état incohérent à l'état cohérent. On peut donc contrôler l'état de cohérence d'une ligne de cache en garantissant qu'une fois que cette dernière a été accédée comme une instruction, elle ne peut plus être accédée comme une donnée. On peut apporter cette garantie de deux manières :

- Il est possible de configurer la MMU pour restreindre les droits d'accès à la page contenant une ligne de cache. Ainsi, une page contenant une ligne de cache incohérente peut être rendue inaccessible en lecture et en écriture, et inversement. En cas d'accès en données sur une ligne de cache incohérente, la MMU lèvera une exception et renverra vers une séquence de contrôle pour rechargement de la ligne dans un état cohérent. Cette méthode a pour avantage de mettre correctement en œuvre l'invariant. Cependant, elle impose au logiciel utile de placer données et instructions dans des pages mémoire séparées sous peine d'être exécuté de manière inefficace, ce qui est une contrainte de développement pénalisante.

- Si le logiciel utile respecte la règle de conception 3, les lignes de caches en état incohérent ne sont accédées que par des instructions. Par conséquent l'invariant est respecté. Cependant, il est invalidé si le logiciel utile ne respecte pas cette règle. Le logiciel de contrôle a la possibilité de détecter de type de violation en configurant un moniteur de performances pour incrémenter sur l'évènement "Cache miss du fait d'une donnée présente mais incohérente" [40, table 9-57, évènement 133].

Règle de conception 3. *Le logiciel utile assure que ses données et instructions sont situées sur des lignes de caches disjointes, ce qui pour un cœur e5500 revient à dire qu'elles ne sont pas alignées sur les mêmes 64 octets.*

Étant donné l'impact de la première méthode sur le logiciel utile, nous adoptons la deuxième méthode, qui dépend de la règle de conception 3. Nous disposons donc d'une garantie sur le fait que les données accédées par des requêtes de type *load* et *store* ne toucheront que des données cohérentes dans le cache L_2.

Nous disposons ainsi d'un invariant que nous avons construit en appliquant la méthode décrite au chapitre précédent. Nous terminons cette étape de définition de l'invariant par l'étude de sa robustesse vis-à-vis de l'exécution du logiciel utile. Il s'agit de montrer que le logiciel utile ne peut pas, au cours de son exécution, invalider l'invariant.

7.4.3 Robustesse de l'invariant vis-à-vis du logiciel utile

L'invariant va être mis en place par le logiciel de contrôle. Le logiciel utile va donc être activé dans une configuration où l'invariant sera vérifié. Le logiciel de contrôle peut être conçu de telle sorte à ne pas violer lui-même l'invariant. Il est cependant nécessaire de s'assurer que le logiciel utile, au cours de son exécution, ne l'invalidera pas non plus.

Pour cela, nous raisonnons par récurrence. Le logiciel utile démarre son exécution dans une configuration où l'invariant est validé. C'est notre condition initiale. L'hypothèse de récurrence se ramène à montrer qu'en supposant l'invariant vérifié, toute instruction exécutée par le logiciel utile n'invalide pas l'invariant ou bien déclenche une exception dirigée vers le logiciel de contrôle. Nous supposons dans l'hypothèse de récurrence que le logiciel utile est exécuté dans les modes de privilège "utilisateur" ou "superviseur invité". Le logiciel de contrôle est exécuté dans le mode "superviseur". Nous supposons que le cœur e5500 n'offre aucun moyen au logiciel utile d'exécuter son code dans le mode de privilège superviseur, ce qui en soit constituerait une faille de sécurité sur le cœur.

Notre problème est donc ramené à une étude du jeu d'instructions, qui est la seconde de ce type, comme évoqué en début de chapitre. En nous référant à la table 7.1 page 100, nous avons écarté certaines classes d'instructions qui, de manière non ambigüe, ne sont pas susceptibles d'altérer le contenu de la MMU, des caches ou encore des registres de configuration (SPR), et n'invalideront donc pas l'invariant.

Cette seconde analyse porte donc sur les classes d'instructions suivantes :

Opérations dans les registres de configuration (SPR). Les instructions *mfspr* et *mtspr* effectuent les opérations dans les registres de configuration. Leur niveau de privilège dépend du registre choisi, ce qui demande d'évaluer la situation au cas par cas pour les différents registres. Nous développons ce point dans la propriété 7.2, ci-après.

Gestion de la MMU. Les instructions de gestion de la MMU effectuent des échanges entre des registres SPR appelés MAS [38]. Les instructions disponibles visent l'écriture dans une entrée de la TLB (*tlbwe*), la lecture d'une entrée (*tlbre*), la recherche d'une entrée (*tlbsx*), l'invalidation locale (*tlbilx*) ou globale d'une entrée TLB (*tlbivax*), et enfin la synchronisation d'un signal d'invalidation propagé entre tous les cœurs (*tlbsync*). Toutes ces instructions sont privilégiées.

Gestion des caches. Les instructions de gestion des caches ont été étudiées précédemment pour caractériser les émissions qu'elles peuvent générer.

- *dcba, dcbal, dcbz, dcbzl, dcbt, dcbtst, icbt.* Ces instructions ont été prises en compte dans la construction de l'invariant. Leur effet est couvert par ce dernier. Elles ne peuvent donc pas l'invalider.

- *dcbtls, dcbtstls, dcblc, icbtls, icblc.* Ces instructions gèrent le verrouillage dans le cache. Elles peuvent être rendues privilégiées en configurant les registres suivants : MSR[UCLE] $= 0$, MSRP[UCLE] [39] $= 1$. L'écriture dans ces registres est privilégiée.

- *dcbi, icbi.* Ces instructions sont privilégiées.

- *dcbf, dcbst.* L'usage de cet instruction est prohibé par la règle de conception 2, justifiée page 114.

Levée d'interruptions inter-cœurs. Les instructions correspondantes (*msgsnd* et *msgclr*) sont privilégiées. Leur exécution lève une exception de privilège dirigée vers le logiciel de contrôle.

Lectures et écritures dans l'espace adressable. L'altération des caches, en particulier les niveaux L_1D et L_1I ont été prises en compte dans la définition de l'invariant. Elles ne peuvent donc pas l'invalider.

Propriété 7.2. *Les registres de configuration (SPR) accessibles au logiciel utile dans le mode utilisateur et/ou "superviseur invité" ne peuvent pas invalider l'invariant.*

Démonstration : Nous nous intéressons aux registres SPR pour lesquels une opération d'écriture est susceptible d'altérer l'état du registre ou d'une ressource associée. Les registres en lecture seule ne sont donc pas considérés. Nous montrons que les registres en question sont soit privilégiés, soit neutres vis-à-vis de l'état des ressources impliquées dans l'invariant.

Nous évaluons les registres SPR impliqués dans les opérations suivantes :

38. MMU ASsist registers
39. Machine State Register [Protect], User Cache Lock Enable

Programmation de la MMU. Le seul registre accessible en écriture est MMUCSR0, qui initie les invalidations des TLB. Il est privilégié.
Les registres MAS0 à MAS8, qui programment les TLB, n'altèrent pas directement l'état de la MMU, dans la mesure où les instructions qui entraînent la mise à jour des TLB à partir de ces registres sont privilégiées.

Gestion des caches. Les registres SPR pouvant altérer l'état d'un cache sont les registres L1CSR, L2CSR, et les registres de gestion et d'injection d'erreurs dans le cache. Tous ces registres sont privilégiés.

Ces registres sont les seuls concernés par la gestion du cache et de la MMU. Nous pouvons donc conclure sur le fait que les opérations de lecture et d'écriture dans les SPR ne peuvent pas invalider l'invariant. □

Nous vérifions ainsi notre hypothèse de récurrence portant sur la validité de l'invariant suite à l'exécution par le logiciel utile d'une instruction PowerPC. Nous avons donc montré par récurrence que l'invariant est robuste.

Au final, l'invariant que nous avons défini porte sur des éléments que l'on peut manipuler avec du logiciel, à savoir le contenu des caches et de la MMU. Par construction, il élimine les évènements non contrôlables sur la base de la classification que nous avons effectuée précédemment. La première hypothèse d'application du théorème de contrôlabilité, qui dit que tout évènement sensible est contrôlable ou éliminé par la configuration du cœur, est donc vérifiée.

Nous abordons dans la section suivante la troisième étape de notre méthode, qui consiste à construire les séquences de contrôle.

7.5 Construction des séquences de contrôle

La deuxième hypothèse d'application du théorème de contrôlabilité dit que pour tout évènement sensible qui est contrôlable, il existe une ou plusieurs séquences de contrôle qui reproduisent cet évènement. Nous avons défini dans le chapitre précédent les types de séquences de contrôle suivantes :

Contrôle par émulation. Une séquence de contrôle par émulation peut être définie pour un évènement sensible à une instruction. Elle cherchera à reproduire les effets de l'instruction en question, sans que cette dernière ne soit exécutée. Le logiciel invité a l'illusion que l'instruction sensible a été exécutée.

Contrôle par neutralisation. Une séquence de contrôle par neutralisation s'applique à un évènement sensible à la configuration. Elle cherchera à placer le cœur dans un état où l'évènement n'est plus sensible pour ensuite relancer l'exécution du logiciel utile qui génèrera à nouveau l'évènement.

Comme expliqué dans le chapitre précédent, une séquence de contrôle doit satisfaire la propriété de contrôle des ressources. Pour cela, elle doit maîtriser les dates des accès

qu'elle initie vers les ressources communes. À cet effet, nous avons introduit la propriété de synchronisabilité dans le chapitre précédent (cf. page 74). Cette dernière précise que chaque cœur doit disposer d'un moyen d'accéder par une séquence d'opérations non sensibles à une source d'horloge globale. Nous montrons ci-après cette propriété pour le P5040.

Dans la section 6.3.6 page 87, nous avons proposé une structure de séquence de contrôle en quatre phases, qui étaient illustrées sur la figure 6.12 page 86. Ces phases sont les suivantes :

Préparation. Cette phase plannifie les accès à effectuer et prépare les ressources en conséquence. Lors de cette phase il faut prendre en compte le respect de l'invariant.

Synchronisation. Cette phase aura pour objectif de placer le cœur en attente active d'une fenêtre temporelle où il pourra réaliser ses opérations sur les ressources communes. Comme expliqué dans le chapitre précédent, cette phase repose principalement sur la propriété de *synchronisabilité* (cf définition 6.6 page 74), que nous montrons ci-après pour le P5040. Elle doit également prendre en compte un certain nombre de paramètres, comme la précision de la mesure de la date de référence, la durée entre la fin de son exécution et le début de l'émission effective par la phase d'émission, et enfin le temps d'exécution de la phase d'émission, qui doit être terminée avant la fin de la fenêtre temporelle.

Émission. Cette phase réalise les opérations sur les ressources distantes. il est préférable qu'elle soit synchrone, i.e. que les opérations soient terminées à la fin de son exécution.

Terminaison. Cette phase prépare la reprise de l'exécution du logiciel utile. Elle rétablit l'invariant s'il a été invalidé.

Après avoir montré la propriété de synchronisabilité sur le P5040, nous développons un exemple de séquence de contrôle par neutralisation, sur le même modèle que dans le chapitre précédent. Nous nous concentrons sur les phases de pré-traitement et d'émission, qui sont les plus complexes. Nous abordons par la suite un exemple de séquence de contrôle par émulation.

Synchronisabilité du P5040

Propriété 7.3. *Le P5040 est synchronisable au sens de la définition 6.6.*

Démonstration : Chaque cœur dispose d'une paire de registres appelée "base de temps" (time base, ou TB). Ces paires de registres sont incrémentées sur un signal d'horloge qui vient soit de l'extérieur, soit de l'horloge de l'interconnexion après passage par un diviseur d'horloge, d'un facteur 32 sur le P5040. La fréquence du signal d'horloge résultant est comprise entre 10MHz et 100MHz.

Le circuit d'horloge n'est pas documenté, mais Freescale assure que le déphasage entre chaque cœur est négligeable devant une période du signal, dans la mesure où le circuit est faiblement asymétrique.

Enfin, il est possible de faire démarrer de manière synchrone l'intégralité des bases de temps sur chaque cœur par une écriture dans un registre du processeur [40], qui est rattaché à l'espace des registres de configuration [41]. On a ainsi la certitude qu'à tout moment toutes les bases de temps contiennent la même valeur à une unité près. □

7.5.1 Exemple de séquence de contrôle par neutralisation

Nous reprenons dans cette section l'exemple de séquence de contrôle que nous avons développé dans le chapitre précédent (voir section 6.3.6). Dans le cas présent, cette séquence de contrôle portera sur des évènements de type *load(@, read)*, *store(@,write)* ou encore *fetch(@)*.

Les principales étapes de la séquence de contrôle suivent le même schéma que dans le chapitre précédent :

- Au cours de la phase de préparation, le logiciel de contrôle va reproduire de l'action de la MMU pour effectuer une traduction d'adresse. Dans un second temps, il va allouer un emplacement dans le cache pour une page contenant la donnée à charger. Si nécessaire, il sélectionnera une page victime, qui sera déchargée du cache L_2 puis déréférencée de la MMU.
  Pour effectuer ces opérations, le logiciel de contrôle maintient dans des tables de données une description du contenu de la MMU tel que le voit le logiciel utile, ainsi que du cache L_2. Ces tables de données sont appellées respectivement ***table des pages virtuelles*** et ***table des descripteurs de pages***.
  Contrairement à la séquence de contrôle proposée dans le chapitre précédent, il n'est pas nécessaire de s'assurer que toute donnée relative à la page victime a été retirée du cache L_1D. Nous avons en effet décidé de configurer le cache L_1D dans le mode "Write Shadow", dans lequel le cache L_1D propage toute écriture vers le cache L_2. Dans ce mode, les données sélectionnées comme victimes sont invalidées et non pas réécrites dans le cache L_2.
- Au cours de la phase de synchronisation, le logiciel de contrôle se réfèrera à deux tables de configuration appelées ***table des transactions*** et ***table des fenêtres temporelles*** qui lui indiquent selon l'adresse de la page chargée la –ou les– fenêtres temporelles dans lesquelles la phase d'émission peut avoir lieu.
- Au cours de la phase d'émission, le logiciel de contrôle effectuera une séquence d'émission, la plus courte possible, qui chargera l'intégralité des données de la page dans le cache. La page victime sera écrite en mémoire à ce moment. Le logiciel de contrôle pourra également collecter des statistiques d'utilisation.
- Au cours de la phase de terminaison, le logiciel de contrôle s'assurera que les données nouvellement chargées dans le cache ont bien été verrouillées.

Nous nous focalisons dans cet exemple sur des questions d'implémentation de la phase de préparation sur le cœur e5500. Cette dernière est la plus complexe. Elle gère les mé-

40. Core Timebase Enable Register (CTBENRL)
41. Configuration, Control and Status Registers (CCSR)

canismes de traduction d'adresse, de gestion du cache et de la MMU. Nous développons des points d'implémentation de ces mécanismes à travers les points suivants :

- La manière de programmer la MMU pour référencer correctement les pages présentes dans le cache.
- La gestion des pages en cache référencées par plusieurs règles de traduction.s
- Les politiques de remplacement que l'on peut mettre en place, entre autres dans le cache.

Référencement des pages dans la MMU

Sur le cœur e5500, la taille minimale d'une page est de quatre kilo-octets. La granularité d'un échange entre le cache et la mémoire principale est donc de 4K. Sur un cœur e5500, ces pages peuvent être référencées dans les TLB. Le cache L_2 a une capacité de 512K organisée en huit voies de 64K. Certaines voies seront utilisées par le logiciel de contrôle pour son propre stockage, ce qui diminuera la capacité restante pour le logiciel utile. Il donc faut prévoir une centaine d'entrées dans les TLB pour référencer tout le contenu du cache. Le cœur e5500 dispose de deux TLB. la TLB0 contient 512 entrées réservées aux pages de 4K, organisées en quatre voies, et indexées sur l'adresse effective. La TLB1 contient 64 entrées et est entièrement associative. Cependant, certaines entrées sont utilisées par le logiciel de contrôle.

Nous choisissons d'utiliser la TLB0 pour référencer les pages chargées en cache, la TLB1 n'offrant pas assez d'entrées. Nous disposons de beaucoup plus d'entrées dans la TLB0 que d'emplacements nécessaires pour référencer tout le cache. Cependant, on peut aboutir à la situation paradoxale où le logiciel de contrôle charge des données dans le cache, mais ne peut pas toutes les référencer. Ce dernier n'a en effet que peu de maîtrise sur l'utilisation de l'espace d'adressage effectif telle qu'effectuée par le logiciel utile. On peut se retrouver dans la situation extrême où toutes les pages sont référencées à la même adresse effective, mais à des adresses physiques différentes. Le cache peut toutes les contenir, mais la TLB0 ne peut en référencer qu'un petit nombre, en l'occurrence quatre.

Le référencement des pages fait donc l'objet d'une logique de remplacement qui est implémentée par le logiciel de contrôle. Ce dernier embarque à cet effet une table de données, appelée ***table des descripteurs de TLB***.

Référencement multiple par la MMU des pages en cache

Dans une MMU, plusieurs entrées dans les TLB peuvent référencer la même page dans l'espace physique. Le logiciel de contrôle a par exemple recours à ce mécanisme lorsqu'il charge une page : il dispose d'une plage d'adresse fixe pour le chargement des pages, et il redéfinit à la volée une règle de traduction pour accéder à cette page depuis sa propre plage d'adresse.

Le même phénomène peut se produire dans la table des pages virtuelles. Ici, le référencement multiple est à l'initiative du logiciel utile. Ce dernier peut donc accéder à une même page dans le cache en utilisant plusieurs règles de traduction différentes.

Lorsque c'est le cas le logiciel de contrôle va allouer plusieurs entrées dans la TLB0, qui pointent vers cette page. Il devra s'assurer qu'elles sont toutes invalidées lorsque la page en question sera sélectionnée comme victime. Pour cela, nous ajoutons aux descripteurs de page en cache une structure de liste dans laquelle nous conservons la trace de toutes les descripteurs de TLB menant à cette page.

Politique de remplacement dans le cache L_2

Les logiques de remplacement dans le cache ainsi que dans la TLB0 sont implémentées par le logiciel de contrôle. Le choix de la politique de remplacement est contraint par le niveau d'informations dont dispose le logiciel de contrôle sur l'historique des accès du logiciel utile dans le cache. Sur le cœur e5500, ce niveau est faible, nous ne disposons pas de cet historique. Cela nous empêche d'implémenter des politiques de remplacement telles que LRU [42] ou PLRU [43], qui sont les plus efficaces, y compris dans les analyses de WCET [77].

Le logiciel de contrôle peut maintenir un historique des chargements des pages, ce qui lui permet d'implémenter une politique FIFO [44]. Il en est de même pour le remplacement dans la TLB0.

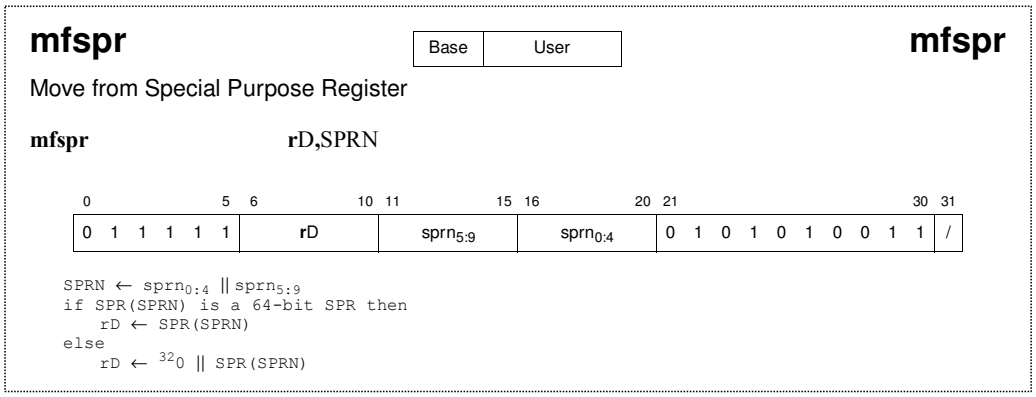
Nous disposons à travers cet exemple d'une séquence de contrôle applicable à des évènements de type *load(@,read)*, *store(@,write)* et *fetch(@)*, qui sont les plus nombreux. On peut proposer une séquence similaire pour traiter les autres évènement de type *load* et *store*. Nous abordons ci-après un exemple de séquence de contrôle par émulation.

7.5.2 Exemple de séquence de contrôle par émulation

Les séquences de contrôle par émulation concernent les instructions sensibles, ainsi que celles qui peuvent invalider l'invariant. Ces séquences de contrôle visent à reproduire le comportement de ces instructions, sans que ces dernières ne soient exécutées.

L'émulation d'instructions est une technique connue notamment dans les solutions de virtualisation, que nous pouvons reprendre dans notre cas. Dans les cas où une instruction n'altère que des registres, ce qui correspond à la majorité des situations, la construction d'une séquence de contrôle ne pose pas de difficultés. On reprend la sémantique de cette instruction, telle que décrite dans le standard [72] ou dans le manuel de référence [41]. Un exemple de sémantique pour une instruction de lecture dans un registre spécial est donné sur la figure 7.12. Pour ce type d'intruction, la séquence de contrôle effectuera les éventuels accès, puis altèrera les registres concernés, et retournera l'exécution au logiciel utile. Une manière de procéder consiste à donner au logiciel de contrôle le moyen d'altérer les valeurs des registres récupérées lors de la sauvegarde de contexte, implémentée dans le vecteur d'exception.

42. Least Recently Used
43. Pseudo Least Recently Used
44. First In First Out



Note : L'instruction *mfspr* ("Move from Special Purpose Register") est appelée pour lire dans les registres de configuration et d'état du cœur. Ces registres permettent entre autre de configurer la MMU, les caches, de retrouver des informations sur l'état du logiciel lors d'une exception, et d'accéder à des moniteurs de performances présents sur le cœur.

FIGURE 7.12 – Sémantique de l'instruction *mfspr*, décrite dans le manuel de référence des cœurs e500 [41]

Pour les instructions de gestion des caches et de la MMU, qui sont en minorité, la séquence de contrôle est amenée à altérer la table des pages virtuelles et la table des pages en cache. Assurer la cohérence entre le contenu de ces tables, l'état du cœur et la bonne exécution du logiciel utile pose certaines difficultés d'implémentation.

Nous proposons dans cette section d'illustrer quelques unes de ces difficultés à travers un exemple portant sur l'émulation de l'instruction *tlbwe*, qui effectue les écritures dans la MMU. Cette instruction a été identifiée lors de la seconde analyse du jeu d'instructions, portant sur les instructions pouvant invalider l'invariant. Comme dans la section précédente, l'essentiel de la complexité se situe dans la phase de préparation.

L'écriture dans la MMU est interceptée par le logiciel de contrôle, qui évaluera la règle de traduction associée et la propagera dans la table des pages virtuelles. Lors de cette opération, toute modification doit être propagée dans la TLB0, dont les entrées ont été programmées à partir des informations contenues dans la table des pages virtuelles.

A titre d'exemple, nous considérons la séquence d'opérations suivante, effectuées par le logiciel utile, sur la base d'une configuration donnée de la table des pages virtuelles :

- Le logiciel utile effectue un certain nombre d'accès, ce qui a entraîné des chargements de données dans le cache L_2. Considérons en particulier une donnée chargée à partir d'une règle de traduction d'adresse présente dans la table des pages virtuelles.
- Le logiciel utile invalide cette règle de traduction par une instruction *tlbwe*. Pour éviter d'invalider l'invariant, le logiciel de contrôle propage cette invalidation dans la table des pages virtuelles et non dans la MMU.

- Si le logiciel de contrôle laissait le logiciel utile reprendre son exécution à ce stade, ce dernier pourrait effectuer de nouveaux accès utilisant la règle de traduction qu'il vient d'invalider. En temps normal, cela déclencherait une exception, mais pas ici : les accès seraient reconnus par les entrées générées par le logiciel de contrôle dans la TLB0.
  Le logiciel de contrôle doit donc au préalable invalider toutes les entrées dans la TLB0 issues de cette entrée. Pour celà, il doit disposer des informations lui permettant de retrouver à partir d'une entrée dans la table des pages virtuelles l'ensemble des entrées dans la TLB0 issues de cette entrée. Nous associons ainsi à chaque entrée dans la table des pages virtuelles une liste des descripteurs de TLB générés par cette entrée. En cas de modification de l'entrée en question, chaque descripteur présent dans cette liste est invalidé.

Cet exemple montre que la construction d'une séquence de contrôle par émulation pour certaines instructions doit être pensée de manière cohérente avec le reste de la logique implémentée dans le logiciel de contrôle. Cependant, les difficultés que nous avons soulevées ne remettent pas en cause l'existence de séquences de contrôle. Nous pouvons conclure positivement quand à l'existence de séquences de contrôle pour chaque évènement sensible aux instructions.

Cette dernière étape nous permet de valider la seconde hypothèse d'application du théorème de contrôlabilité, et nous permet donc de conclure sur son application. Au cours de cette étape, nous avons été amenés à définir les principaux mécanismes que notre logiciel de contrôle est amené à implémenter. La construction des séquences de contrôle aboutit donc à une spécification à un niveau de détails relativement élevé de la plupart des mécanismes du logiciel de contrôle.

7.6 Synthèse

Nous avons développé dans ce chapitre un cas d'étude basé sur le processeur P5040. L'objectif était de montrer que, sous certaines conditions, ce processeur satisfait les conditions d'application du théorème de contrôlabilité. C'est effectivement le cas, sous réserve que les règles suivantes soient respectées dans le logiciel utile :

- Le logiciel utile n'utilise pas les instructions *mbar*, *sync* (cf règle 1), ainsi que *dcbf* (cf règle 2)
- Le logiciel utile est organisé pour que données et instructions ne partagent pas les mêmes lignes de caches, i.e. ne soient jamais contenues dans le même bloc de 64 octets. (cf règle 3)

Nous avons démontré la contrôlabilité du P5040 en appliquant la méthodologie que nous avions proposée dans le chapitre 6. Cette méthode visait dans un premier temps à caractériser le cœur e5500 pour en extraire un ensemble d'évènements sensibles, dont les relations ont été formalisées par des arbres d'activités. Dans un deuxième temps nous avons classifiés ces évènements selon leur type de sensibilité, et leur contrôlabilité, pour

satisfaire la première hypothèse d'application du théorème de contrôlabilité. Cela nous a amené à définir une propriété invariante sur l'état du cœur, dont la mise en œuvre est à la charge du logiciel de contrôle. Dans un troisième temps, nous avons construit des séquences de contrôle, dont nous avons détaillé certains aspects dans la section 7.5.

Limites du cas d'étude

Les analyses que nous avons effectuées pour caractériser le cœur e5500 ont été menées de la manière la plus exhaustive possible. Toutefois, cette exhaustivité a été limitée par le fait que le cœur e5500 est un composant COTS, que Freescale a documenté à un certain niveau de détails. Même si ce niveau de détails est élevé, le niveau de confiance dans notre analyse sera toujours limité par le caractère COTS du cœur.

Se pose également la question de l'applicabilité de notre méthode sur d'autres processeurs. Une partie importante de notre méthode repose sur le fait que l'on est capables de contrôler le contenu des caches de manière logicielle. Cela est possible sur un cœur PowerPC car il dispose d'instructions de gestion de caches. En passant sur d'autres cœurs, on risque de ne pas retrouver ce type d'instructions. Il faudrait alors investiguer d'autres pistes pour maîtriser le contenu des caches.

Spécification du logiciel de contrôle

La combinaison des mécanismes qui implémentent l'invariant ainsi que les séquences de contrôle que nous avons construit aboutit à une spécification presque complète du logiciel de contrôle.

Dans notre cas, le logiciel de contrôle devra implémenter les éléments suivants :

- Une réplique logicielle de la MMU afin de décoreller les informations relatives à la traduction d'adresse, qui proviennent du logiciel utile, de l'état réel de la MMU qui devra respecter l'invariant.
- Une table de données contenant les informations nécessaires pour maintenir et décider du remplacement de pages de quatre kilo-octets dans le cache L_2.
- Une table de données contenant les informations nécessaires pour référencer le contenu du cache L_2 dans la MMU, plus précisément dans la TLB0.

Les séquences de contrôle, tant par émulation que par neutralisation, ont de nombreuses intéractions avec ces structures de données. Elles participent notamment à une logique de remplacement dans le cache L_2 avec une granularité d'une page mémoire, soit quatre kilo-octets. Ce point fait l'objet d'un développement dans le chapitre suivant.

Nous avons montré dans ce chapitre qu'il était possible de mettre en place un logiciel de contrôle sur le P5040, en assurant les propriétés de contrôle des ressources et d'équivalence vis-à-vis du logiciel utile. Cela a abouti à la réalisation d'un prototype, dont nous étudions les aspects d'efficacité dans le chapitre suivant.

Chapitre 8

Efficacité du logiciel de contrôle

Sommaire

8.1 Vue d'ensemble de Marthy 132
8.1.1 Principaux modules logiciels 132
8.1.2 Mécanisme de démarrage des cœurs secondaires 133
8.1.3 Organisation du cache L2 134
8.2 Démarche d'évaluation 134
8.3 Évaluation de benchmarks industriels 137
8.3.1 Efficacité du logiciel de contrôle sur Stream 137
8.3.2 Efficacité du logiciel de contrôle sur AES 138
8.3.3 Efficacité du logiciel de contrôle sur FFT 140
8.4 Evaluation d'une application avionique 141
8.4.1 Efficacité du logiciel de contrôle sur l'application avionique 142
8.4.2 Pistes pour l'optimisation du logiciel de contrôle 143
8.5 Synthèse 144

Nous avons développé dans le chapitre précédent un cas d'étude dans lequel nous avons établi la contrôlabilité du processeur P5040, appartenant à la gamme QorIQ développée par Freescale. Pour appliquer le théorème de contrôlabilité, nous avons été amenés à définir une logique de contrôle qui met en œuvre une gestion logicielle du cache L_2, ainsi qu'une réplique logicielle de la MMU.

Nous avons implémenté cette logique de contrôle dans un prototype, appelé Marthy [45], dont nous donnons une vue générale dans ce chapitre. Nous nous intéressons, à travers la notion d'efficacité du logiciel de contrôle, à la manière dont du logiciel utile se comporte au niveau macroscopique, lorsqu'il subit cette logique de contrôle. L'objectif est de faire ressortir des caractéristiques générales du logiciel utile qui font que le logiciel de contrôle sera efficace, ou à l'inverse inefficace.

Ce chapitre est découpé en quatre sections. Dans un premier temps nous donnons une vue d'ensemble des principales caractéristiques de notre prototype. Cette section vise

45. Multicore Avionic Real-Time Hypervisor

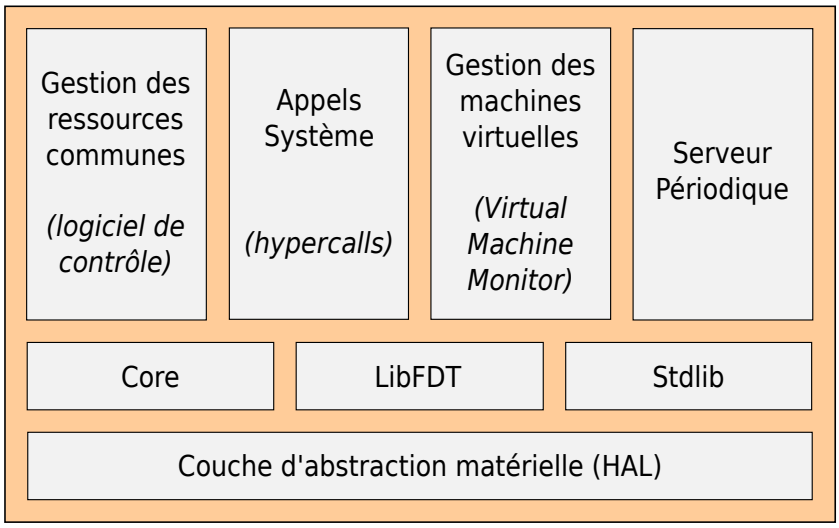


FIGURE 8.1 – Vue d'ensemble de l'architecture logicielle de Marthy

à illustrer l'environnement logiciel dans lequel le logiciel de contrôle va s'insérer. Dans la section 8.2, nous développons la notion d'efficacité du logiciel de contrôle, ainsi que notre démarche d'évaluation avec les différentes configurations de test. Dans la section 8.3, nous appliquons cette démarche sur trois benchmarks utilisés dans l'industrie. Enfin, dans la section 8.4, nous appliquons cette démarche à un exemple d'application avionique.

8.1 Vue d'ensemble de Marthy

Le logiciel de contrôle introduit dans notre approche est implémenté dans un hyperviseur appelé Marthy, qui a été développé dans le cadre de cette thèse. Nous en présentons les principales fonctionnalités dans cette section.

Marthy intègre ponctuellement du code de Topaz [42], qui est un hyperviseur développé par Freescale sous licence autorisant la redistribution et la réutilisation du code source.

8.1.1 Principaux modules logiciels

Comme illustré sur la figure 8.1, Marthy embarque plusieurs modules logiciels indépendants. Ils remplissent les fonctions suivantes :

Le Gestionnaire de Ressources Communes. Ce module implémente le logiciel de contrôle. Ce dernier est appelé pour exécuter les séquences de contrôle. Il met en place l'invariant que nous avons défini dans le chapitre précédent.

Le Gestionnaire de Machines Virtuelles. Ce module implémente les mécanismes de virtualisation qui font de Marthy un hyperviseur. Ce module a été en partie repris

du code de Topaz. Il implémente notamment le concept *d'espace d'adressage physique invité*, qui est utilisé en virtualisation. Ce concept a été intégré dans la gestion de la table des pages virtuelles, manipulée par le logiciel de contrôle.

Le serveur de maintenance. Ce module assure des tâches qui doivent être répétées périodiquement. Ces tâches assurent entre autres la gestion des communications inter-cœurs, qui permettent le multiplexage des consoles.

Les appels systèmes (hypercalls). Il s'agit de l'implémentation de l'interface de para-virtualisation proposée par Marthy. Cette interface offre des services qui fournissent au logiciel utile une console de débogage, qui est multiplexée, des méthodes facilitant l'instrumentation du logiciel utile, ainsi qu'un appel pour déclencher le serveur de maintenance.

Les couches logicielles de bas niveau. Ces couches logicielles embarquent les mécanismes de démarrage des cœurs secondaires, les canaux de communication inter-cœurs, la gestion des fichiers de configuration, appelés *device tree*, qui sont définis dans le standard ePAPR. Ces couches embarquent enfin le logiciel en support des autres modules, par exemple la console de débogage et l'adhérence matérielle.

Nous venons de donner une vue d'ensemble de l'organisation de Marthy. Le fait que Marthy soit un hyperviseur est ici opportun, cela nous permet de mettre en œuvre des techniques de virtualisation pour embarquer des systèmes d'exploitation au sein du logiciel utile. Le logiciel du contrôle est dans ce cas un module intégré au noyau de Marthy, mais il pourrait également être intégré dans le noyau d'un système d'exploitation.

Nous présentons dans la suite de cette section certains points intéressants dans l'implémentation de Marthy. Ces mécanismes portent sur le démarrage des cœurs secondaires, qui doit obéir à une règle de séparation introduite dans le chapitre précédent, ainsi que l'organisation du cache L_2 mise en œuvre par le logiciel de contrôle.

8.1.2 Mécanisme de démarrage des cœurs secondaires

Nous avons introduit dans le chapitre précédent une règle de conception disant qu'aucune donnée accessible en écriture ne doit être partagée entre les répliques déployées sur chaque cœur. À cet effet, nous concevons le logiciel de contrôle avec des sections de données et d'instructions séparées. La section d'instructions, qui contient également des données en lecture seule, est partagée entre les cœurs. Comme illustré sur la figure 8.2, nous appliquons sur la section de données un décalage propre à chaque cœur, qui est passé en argument dans le code de démarrage.

Ce mécanisme de démarrage permet de garantir une séparation des données entre les cœurs, pour un nombre arbitraire de cœurs. Cela complique le développement du logiciel de contrôle, dans la mesure où il faut éliminer toute variable initialisée, dans laquelle on a besoin d'écrire, mais qui ne peut être recopiée dans les sections de données de chaque cœur. Nous nous restreignons à l'utilisation de constantes, et de variables non initialisées.

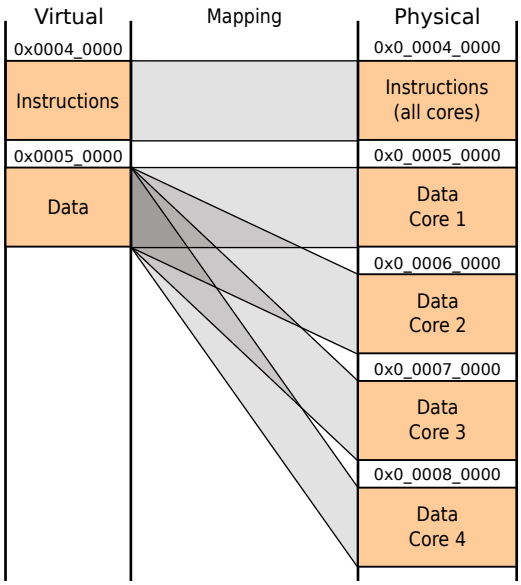


FIGURE 8.2 – Mapping de la mémoire garantissant la séparation des données entre cœurs

8.1.3 Organisation du cache L2

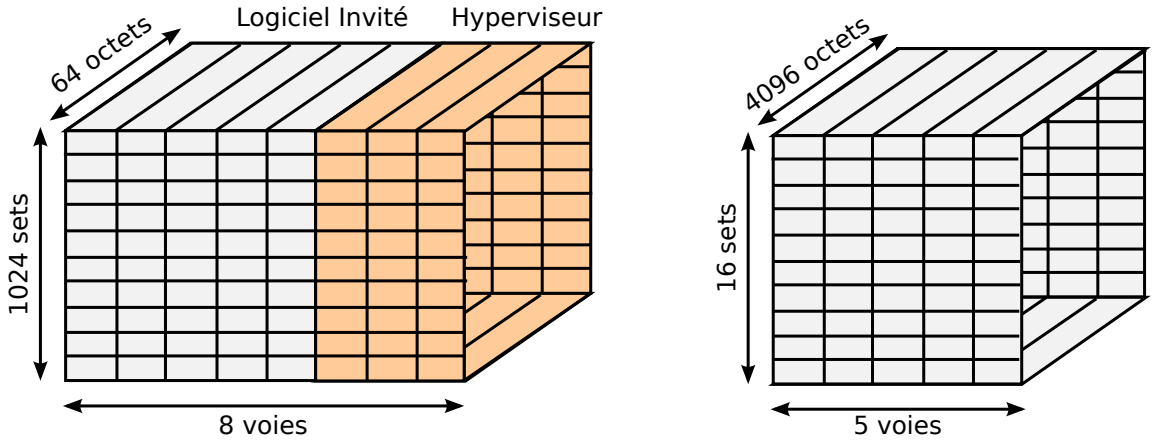
Marthy est conçu comme un hyperviseur monolithique de petite taille. Il est répliqué sur chaque cœur, en étant stocké de manière persistente dans le cache L_2. A cet effet, il a une empreinte mémoire d'environ cent vingt kilo-octets, et occupe trois voies du cache L_2, deux pour les instructions et une pour les données.

Comme illustré sur la figure 8.3, cela libère cinq voies de 64K chacune, soit un cache L_2 équivalent de 320K organisé en cinq voies avec une politique de remplacement FIFO. On peut décrire ce cache par un modèle équivalent tel qu'il est "vu" par le logiciel utile. En effet, la logique de contrôle implémentée dans Marthy gère les échanges entre le cache L_2 et la mémoire principale avec une granularité d'une page mémoire, soit quatre kilo-octets. On peut donc considérer un cache équivalent disposant de lignes de quatre kilo-octets, à cinq voies gérées par une politique FIFO, qui est implémenté par la logique de contrôle.

Cette description donne une vue globale sur l'architecture logicielle de Marthy, et ses principales fonctionnalités. L'accent est mis sur le fait que le logiciel de contrôle est un module de Marthy, mais n'est pas central dans son architecture. Son utilisation dans un autre contexte est possible, par exemple intégré à un système d'exploitation.

8.2 Démarche d'évaluation

Nous avons défini dans notre approche la notion d'efficacité du logiciel de contrôle comme étant le ratio entre un temps d'exécution dans une configuration de référence et une configuration de test incluant le logiciel de contrôle.



(a) Vue du cache L_2 du cœur e5500 occupé par Marthy

(b) Cache équivalent vu par le logiciel utile

FIGURE 8.3 – Organisation du cache L_2 effectuée par Marthy

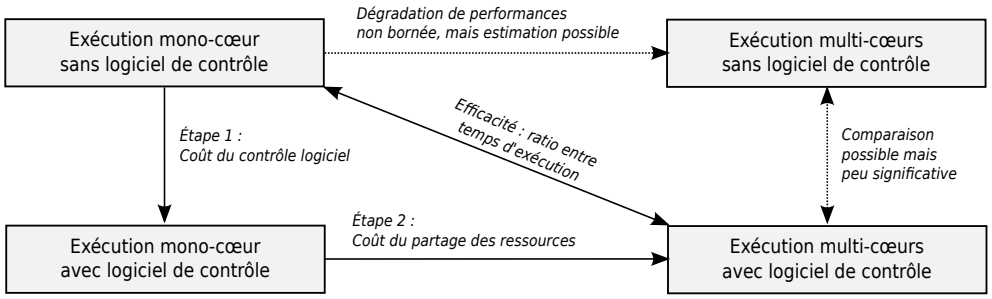


FIGURE 8.4 – Vue d'ensemble de la démarche d'évaluation de l'efficacité du prototype

Comme illustré sur la figure 8.4, qui reprend la figure 5.2 dans le chapitre d'approche, la configuration de référence est définie comme la configuration mono-cœur sans logiciel de contrôle. Nous proposons une évaluation de l'efficacité en deux étapes :

- Dans un premier temps, nous évaluons le ratio des temps d'exécution entre la configuration de référence et une configuration où le logiciel de contrôle est actif, mais autorise immédiatement l'accès aux ressources communes. Les séquences de contrôle mettent en œuvre la logique de remplacement par pages de 4K dans le cache L_2. Cette première étape permet d'évaluer la manière dont une application se comporte vis-à-vis des mécanismes de contrôle que nous avons implémentés, en particulier la logique de remplacement dans le cache L_2, que nous avons abstrait par la notion de cache équivalent.
- Dans un second temps, nous configurons le logiciel de contrôle pour respecter une stratégie d'utilisation TDMA, en faisant varier la proportion de temps alloué à chaque cœur. Cette seconde étape permet d'évaluer à quel point la stratégie TDMA impacte le temps d'exécution du logiciel utile.

Dans la suite de ce chapitre, nous utilisons la terminologie suivante pour désigner les configurations dans lesquelles se sont déroulés les tests :

Exécution de référence. L'exécution de référence correspond à une configuration où l'application est exécutée sans logiciel de contrôle. Elle accède donc directement aux ressources communes.

Configuration à un cœur. La configuration à un cœur correspond au cas où du logiciel utile est exécuté avec les mécanismes de contrôle, en particulier la logique de remplacement du cache L_2. Le mécanisme de contrôle autorise immédiatement le cœur à accéder aux ressources partagées.

Configuration à 2 (resp. 3, 4, 8, 12) cœurs. Dans cette configuration, une période de 2 (resp. 3, 4, 8, 12) fenêtres temporelles est instanciée. Le cœur sur lequel est exécuté l'application se voit allouer une seule fenêtre. Cette configuration simule une répartition équitable de la bande passante entre 2 (resp. 3, 4, 8, 12) cœurs.

Conditions expérimentales

Le processeur est configuré avec les caractéristiques suivantes :

- Les cœurs sont alimentés par une horloge à 2,26GHz. L'horloge du bus d'interconnexion est à 800MHz et les contrôleurs DDR ont une horloge à 600MHz.
- Le cache L_3, appelé aussi "Corenet Platform Cache" (CPC) est désactivé.
- Les contrôleurs DDR sont configurés pour concentrer un maximum de trafic sur un minimum de pages ouvertes, dans le but de maximiser la contention. Pour cela, nous désactivons les mécanismes d'entrelacement proposés par le processeur au niveau de l'interconnexion et des contrôleurs DDR. Ainsi, l'intégralité du trafic se concentre sur un seul des deux contrôleurs, à la moitié de sa capacité en terme de nombre de pages ouvertes.
- La cohérence de cache est désactivée.

Lors de la phase expérimentale, nous avons utilisé l'environnement logiciel suivant :

Compilateur : gcc, version 4.9.1, avec le niveau d'optimisation "-O2", avec une gestion matérielle des nombres à virgule flottante.

libc : Newlib, version 2.1.0

bootloader : U-boot, pré-installé sur la carte d'évaluation.

Les temps d'exécution sont mesurés au niveau utilisateur ou au niveau du logiciel de contrôle, selon les applications. La mesure utilise les registres de base temporelle (Time Base), dont la précision est de l'ordre d'une dizaine de nano-secondes. L'échantillonage varie de quelques dizaines de mesures à quelques milliers. Il faut également prendre en compte le fait que certains benchmarks comme Stream reposent sur un échantillonnage interne, qu'il n'est pas nécessaire de redonder.

Nous nous sommes concentrés sur l'évaluation de l'efficacité de benchmarks utilisant des données en mémoire principale. Ce type de benchmark nous permet en effet d'évaluer l'efficacité de notre logique de gestion du cache L_2, qui est le mécanisme central de notre logiciel de contrôle.

8.3 Évaluation de benchmarks industriels

Nous nous sommes intéressés aux trois benchmarks suivants, qui sont utilisés dans l'industrie :

Stream. Cette application effectue des opérations simples sur des nombres à virgule flottante. Chaque opération demande au cœur de charger quelques données, entre 8 et 24 octets selon les opérations. Le benchmark déduit de son temps d'exécution une bande passante. Il effectue un certain nombre d'itérations et affiche la meilleure bande passante mesurée.

AES. Nous avons utilisé un algorithme de chiffrement selon le protocole AES [46]. Cet algorithme lit un tableau de valeurs en entrée, chiffre chaque octet selon une clé qui est définie dans le code source, et écrit chaque octet chiffré dans un tableau de sortie.

L'intérêt de cet algorithme est qu'il parcourt linéairement l'intégralité de ses données d'entrées et de sortie. Quand il traite des données d'une taille supérieure à 4K, l'intégralité des données d'une page de 4K est utilisée, sans entraîner de déchargement et de rechargement de cette page. On peut s'attendre à ce que le logiciel de contrôle soit efficace sur cette application.

FFT. Nous avons pris un algorithme de calcul de transformée de Fourier rapide (FFT) issu de la suite Mi-Bench [47]. Cet algorithme génère dans un tableau d'entrée un signal avec un nombre configurable d'harmoniques, puis en effectue la FFT, qu'il stoque dans un tableau de sortie. L'intérêt de cet algorithme est qu'il manipule tout le temps l'intégralité de ses données. Si la taille des données manipulées dépasse celle du cache, on peut s'attendre à ce que le mécanisme de remplacement dans le cache soit beaucoup sollicité, et donc que le logiciel de contrôle ne soit pas efficace sur cette application.

Nous développons dans la suite de cette section l'évaluation de l'efficacité du logiciel de contrôle sur chaque benchmark.

8.3.1 Efficacité du logiciel de contrôle sur Stream

La comparaison entre l'exécution de référence et la configuration à un cœur montre que l'on perd une part significative de la bande passante, entre 40% et 55% selon les opérations.

Cette perte de bande passante ne peut pas être imputée à l'inefficacité de la séquence d'émission. Nous avons mesuré des chargements de pages d'une durée d'environ 4,5 microsecondes, ce qui correspond à un débit supérieur à 900 MB/s. On peut cependant imputer cette perte de bande passante au temps d'exécution du reste de la séquence de contrôle, qui est de l'ordre de 3 micro-secondes, et que l'on peut chercher à améliorer. Il

46. Téléchargement : http ://web.cs.ucdavis.edu/ rogaway/ocb/ocb-ref/rijndael-alg-fst.c
47. Téléchargement : https ://github.com/embecosm/mibench/tree/master/telecomm/FFT

TABLE 8.1 – Bandes passantes mesurées par Stream dans les différentes configurations

	Bande passante mesurée (MB/s)			
Configuration	Copy	Scale	Add	TriAdd
Exécution de référence	994,3	787,4	1225,9	1199,9
Configuration à un cœur	518,6	472,9	507,8	508,1
Configuration à deux cœurs	336,9	336,9	336,9	337,0
Configuration à trois cœurs	224,6	224,6	224,6	224,6
Configuration à quatre cœurs	168,4	168,4	168,4	168,4
Configuration à huit cœurs	84,2	84,2	84,2	84,2
Configuration à douze cœurs	56,1	56,1	56,1	56,1

est également possible de faire en sorte que le code de Stream ne soit jamais déchargé du cache, alors que c'est le cas actuellement. Une piste intéressante consisterait à utiliser un mécanisme de verrouillage logique dans le cache.

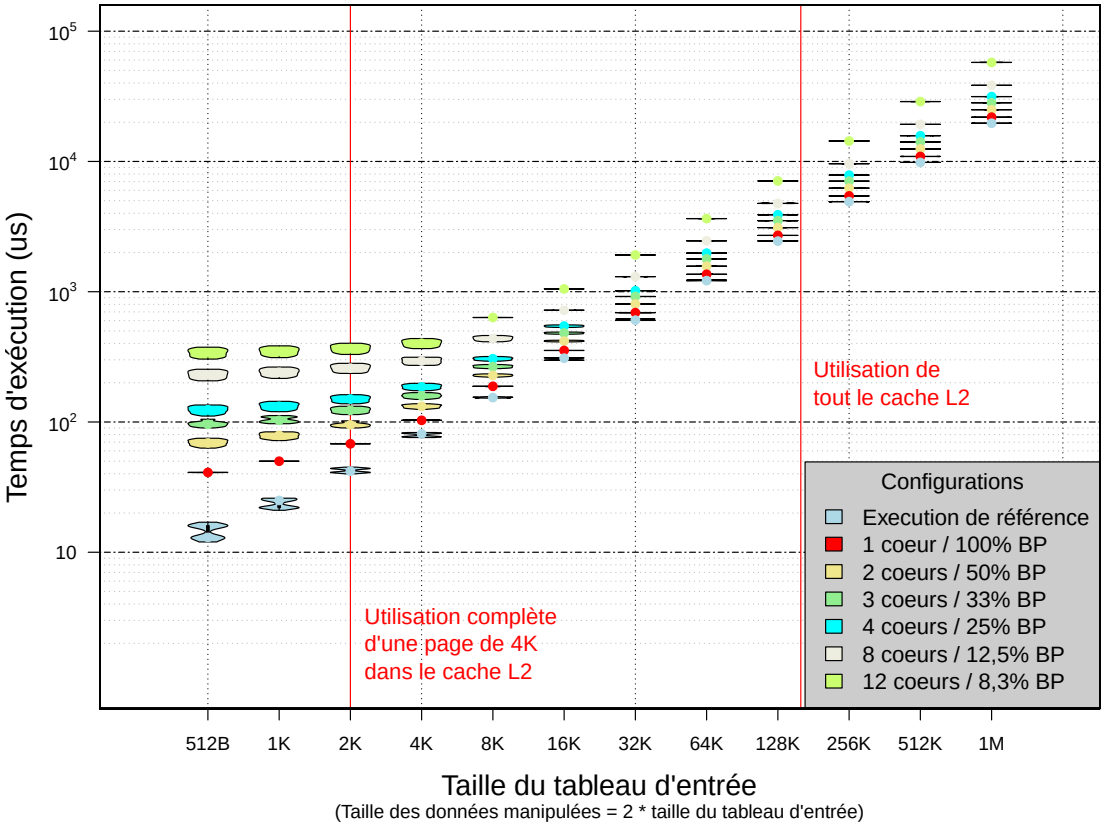
En outre, on peut constater que dans l'exécution de référence, toutes les opérations n'ont pas la même durée. Cet écart se réduit dans la configuration à un cœur, pour ensuite s'annuler dans les configurations suivantes. On arrive donc à stabiliser la bande passante associée au cœur, même si les valeur restent basses.

In fine, les mesures de performance sur Stream sont moyennes. La bande passante est dégradée de manière significative, y compris pour la configuration à un cœur. Cependant, le niveau de bande passante atteint nous laisse penser qu'il est possible d'arriver à une bande passante correcte en proposant des optimisations de Marthy.

8.3.2 Efficacité du logiciel de contrôle sur AES

L'application de chiffrement selon l'algrithme AES est l'exemple de référence d'une application qui sera a priori efficace sur notre logiciel de contrôle. Cette dernière traite des données de manière linéaire, i.e. lit des données dans un tableau de manière séquentielle, les chiffre à la volée et les écrit immédiatement dans un tableau de sortie. Si ces tableaux sont d'une taille suffisamment élevée, ils occupent une page complète dans le cache, ce qui correspond à une utilisation efficace des 4K chargés en un seul bloc.

Nous pouvons effectuer plusieurs constats à partir des séries de mesures que nous avons réalisée. D'une part, AES se comporte comme nous l'avions prévu. En dessous d'une certaine taille de données, seule une petite partie du bloc de 4K chargé en cache est effectivement utilisée par le logiciel invité. La majorité du bloc est chargée pour rien, mais à un coût quasi fixe. L'exécution utilisant le logiciel de contrôle n'est pas efficace.



Note : Les courbes représentées ici montrent sur l'axe vertical les plages de temps d'exécution mesurés lors des différentes exécutions de l'application dans une même configuration. Le point de couleur correspond à la valeur médiane. Les courbes qui sont dessinées de part et d'autre de l'axe vertical correspondent à la distribution de la série de valeurs.

FIGURE 8.5 – Mesures de temps d'exécution de l'application de chiffrement AES dans les différentes configurations

Dès que l'on arrive à une taille de 4096 octets à chiffrer, soit l'équivalent de deux pages chargées en cache, la différence entre les temps d'exécution entre l'exécution de référence et la configuration à un cœur se stabilise à 10%. L'efficacité se dégrade progressivement selon l'augmentation du nombre de cœurs. Dans cet exemple, le logiciel de contrôle est efficace –au sens de notre définition initiale– jusque dans une configuration à huit cœurs.

On pourra enfin constater que les exécutions sur des petites valeurs montrent que le temps d'exécution est naturellement variable sur le logiciel de contrôle, dans une amplitude de quelques dizaines de micro-secondes. Cette amplitude qui augmente en même temps que le nombre de cœurs. Le phénomène correspond à la durée d'attente

associée à la première phase de synchronisation avec les fenêtres d'accès aux ressources. Cette durée est variable mais bornée.

Curieusement, pour des plus grandes valeurs d'entrées, la même variabilité n'est plus observée. Nous expliquons ce phénomène par un effet de bord de la méthode de collecte des temps d'exécution. Celle-ci a lieu au sein du logiciel utile, éventuellement par un appel système, mais le problème est le même : il faut charger l'instruction correspondante. Or, l'instruction en question a probablement été déchargée du cache, et doit être rechargée, ce qui implique une nouvelle phase d'accès, avec une nouvelle synchronisation. Cette dernière synchronisation élimine toute la variabilité dans le temps d'exécution de l'application.

8.3.3 Efficacité du logiciel de contrôle sur FFT

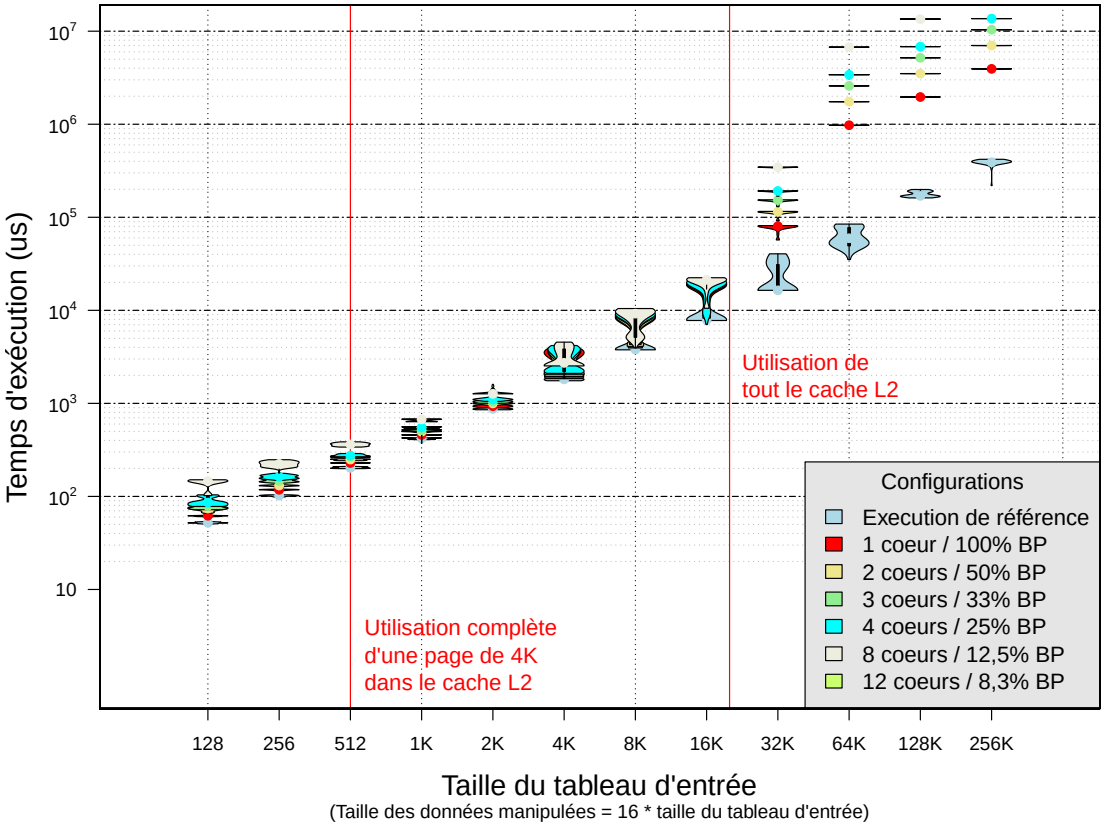
Les algorithmes de FFT [48] se caractérisent par le fait qu'ils résonnent toujours sur l'intégralité de leurs données. Dans notre cas d'étude, nous avons choisi cet algorithme car nous pensons que c'est un mauvais candidat pour l'efficacité du logiciel de contrôle. En effet, ce type d'algorithme stimule beaucoup les mécanismes de remplacement du cache, et la manière d'implémenter cet algorithme en minimisant le taux de cache miss est un champ de recherche encore actif aujourd'hui.

Nous avons pu constater, d'après les mesures illustrées sur la figure 8.6, que tant que les données manipulées ne débordent pas du cache de 320K alloué au logiciel utile, l'impact sur le temps d'exécution est faible, en tout cas compatible avec notre définition de logiciel de contrôle efficace. L'écart entre les différentes configurations est plus réduit que pour AES, ce qui peut s'expliquer par le fait qu'une FFT effectue beaucoup de calcul. Le temps passé à calculer depuis le cache est donc plus grand devant le temps passé à attendre l'autorisation d'accéder aux ressources communes.

Dès que la taille des données manipulées dépasse la taille du cache, on peut constater que le temps d'exécution de la FFT et nettement plus élevé lorsqu'elle est exécutée avec le logiciel de contrôle. On peut observer jusqu'à un facteur 20 dans le ratio de temps d'exécution. Un algorithme comme la FFT, dès qu'il sort de son domaine d'utilisation dans lequel il est efficace, devient inexploitable sur le logiciel de contrôle. Vu les ratio de performances observés, nous considérons qu'il n'est pas pertinent de chercher à optimiser le logiciel de contrôle pour ce type d'application.

Nous abordons dans la section suivante le cas d'une application avionique de taille moyenne.

48. Fast Fourier Transform

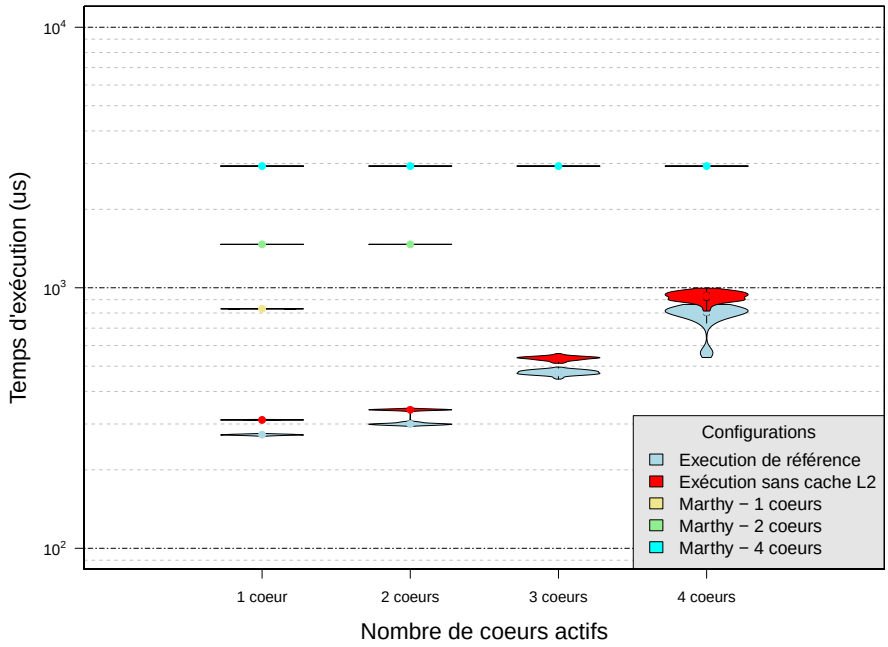


Note : Les courbes représentées ici montrent sur l'axe vertical les plages de temps d'exécution mesurés lors des différentes exécutions de l'application dans une même configuration. Le point de couleur correspond à la valeur médiane. Les courbes qui sont dessinées de part et d'autre de l'axe vertical correspondent à la distribution de la série de valeurs.

FIGURE 8.6 – Mesures de temps d'exécution de l'application de FFT dans les différentes configurations

8.4 Evaluation d'une application avionique

Nous avons présenté dans la section précédente le cas de trois benchmarks industriels, sur lesquels nous avons évalué l'efficacité du logiciel de contrôle. Nous avons constaté que ce dernier est efficace pour l'application de chiffrement AES, efficace pour l'application de calcul de FFT tant que la taille des données manipulées tient dans le cache L_2, et très inefficace au delà. Sur l'application Stream, le logiciel de contrôle n'est pas efficace, mais se situe à un niveau où on peut espérer le devenir si on dispose de biais pour optimiser la logique de contrôle.



Note : Contrairement aux figures précédentes, cette figure montre la distribution de temps d'exécution de l'application avionique tandis que les autres cœurs exécutent une application de stress sur la mémoire principale. Les paramètres en abscisse représentent le nombre de cœurs actifs. À partir de deux cœurs, on a du stress.
Une configuration sans Marthy, mais où le cache L_2 a été désactivé a été insérée. Cette configuration se rapproche de configurations avioniques.

FIGURE 8.7 – Mesures de temps d'exécution de l'application avionique

Nous nous intéressons dans cette section au cas d'une application avionique de taille moyenne.

8.4.1 Efficacité du logiciel de contrôle sur l'application avionique

L'application que nous évaluons est utilisée dans le contrôle des écrans du cockpit d'un ATR-42. Elle sélectionne les informations à afficher à partir d'entrées issues de différents capteurs et calculateurs. Il s'agit d'une application de taille moyenne, contenant environ 200.000 lignes de code, une partie étant générée. Une fois compilée, cette application a une empreinte mémoire d'environ 600K, répartie à deux tiers pour les instructions et un tiers pour les données. Dans un environnement opérationnel, cette application est exécutée sur des données qui sont lues à travers l'API d'ARINC 653. Dans notre contexte, nous étudions cette application sans système d'exploitation, et en utilisant des données

fixes. Cette application s'exécute en boucle courte, avec un temps d'exécution de l'ordre de la milli-seconde.

Cette application est un bon exemple de logiciel sur lequel évaluer notre logiciel de contrôle. D'une part, il s'agit d'une boîte noire, dont nous ne connaissons pas le comportement ni la structure. D'autre part, son empreinte mémoire dépasse la taille du cache L_2 dans une proportion raisonnable. On peut donc s'attendre à ce que la logique de remplacement soit sollicitée.

Comme illustré sur la figure 8.7, on peut constater que l'impact du logiciel de contrôle sur l'application est élevé. En outre, le temps d'exécution de l'application augmente significativement lorsque l'on met en œuvre des configurations à plusieurs cœurs. On constate également que le logiciel de contrôle élimine la variabilité du temps d'exécution dûe aux interférences, ce qui était son objectif initial.

Nous considérons ici que le logiciel de contrôle a correctement fonctionné, mais qu'il n'a pas été efficace sur cette application. Cependant, étant donné les ratio avec lesquels le temps d'exécution a augmenté, on peut espérer atteindre un niveau d'efficacité correct en introduisant des optimisations sur le logiciel de contrôle vis-à-vis de cette application. Nous présentons ci-après quelques pistes pour effectuer ces optimisations.

8.4.2 Pistes pour l'optimisation du logiciel de contrôle

Nous avons constaté que le logiciel de contrôle n'est pas efficace sur l'application avionique. Toutefois, il reste dans un domaine où l'on peut proposer des optimisations visant à retrouver cette efficacité. Nous proposons à cet effet des optimisations portant sur le logiciel de contrôle, et non sur l'application.

Lors de l'exécution de l'application avionique, nous avons remarqué que la séquence de contrôle a été sollicitée à 220 reprises, ce qui représente un trafic total de 880K, un peu moins de trois fois la taille du cache. Les pistes intéressantes pour l'optimisation de la logique de contrôle portent sur les points suivants :

- L'identification des pages qui sont rechargées dans le cache à de nombreuses reprises. Cette identification est envisageable avec un module de trace associé à la logique de contrôle. Une fois ces pages identifiées, il est possible de les pré-charger et/ou de les verrouiller dans le cache.
- L'identification du nombre d'accès à chaque page contenant des données. Celles qui sont très peu sollicitées par le logiciel peuvent ne pas être chargées, auquel cas le logiciel de contrôle leur affectera une séquence de contrôle par émulation. Pour obtenir ce nombre d'accès, il est possible de rejouer l'exécution du logiciel utile en utilisant systématiquement des séquences de contrôle par émulation, et en comptant leurs occurrences.
- La séparation entre données et instructions dans le cache. Nous avons constaté que lorsqu'un logiciel manipule de grosses quantités de données, ces dernières finissent par entraîner des déchargements d'instructions, ces dernières devant être immédiatement rechargées. On peut aborder ce problème en séparant les voies du cache en deux partitions, le choix de la partition étant effectué en fonction du

type d'accès intercepté, en donnée ou en instruction.

Ces pistes visent à améliorer l'efficacité du logiciel de contrôle sans pour autant chercher à retoucher le logiciel utile. On peut également envisager de rendre ce genre de mécanisme directement utilisable par le logiciel utile. Cela permettrait à un développeur d'application de transmettre au logiciel de contrôle des informations utiles pour que ce dernier puisse adapter sa logique de contrôle.

8.5 Synthèse

Nous avons donné dans ce chapitre une vue d'ensemble du logiciel de contrôle tel qu'il avait été spécifié dans le chapitre précédent. Ce logiciel de contrôle est intégré au sein du noyau d'un hyperviseur de petite taille. Cela nous permet de tirer parti de techniques de virtualisation permettant le déploiement de systèmes d'exploitation dans le logiciel utile. Ce prototype nous permet de déployer du logiciel utile, mais également d'observer la durée des accès effectués vers les ressources communes. Nous avons constaté à cette occasion que lorsque le logiciel de contrôle met en œuvre une stratégie TDMA, la distribution des temps d'accès aux ressource ne montre plus de signes d'interférences.

Nous avons ensuite proposé un ensemble de configurations du logiciel de contrôle sur le processeur. Ces configurations visent à évaluer d'une part la sensibilité du logiciel utile aux mécanismes implémentés dans la logique de contrôle, et d'autre part la sensibilité du logiciel utile à une stratégie d'utilisation TDMA instanciée pour un nombre de cœurs fictifs qui varie. Nous avons constaté les faits suivants :

- Le logiciel de contrôle est efficace lorsqu'une application traite un minimum de quelques kilo-octets de données regroupées dans des pages mémoire proches.
- Le logiciel de contrôle est inefficace sur une application qui manipule des données de manière désordonnée.
- Une application est peu sensible à la politique TDMA lorsqu'elle effectue beaucoup de calculs sur les données qu'elle manipule. En ce sens, nous avons constaté que le temps d'exécution d'une FFT varie très peu en fonction du nombre de cœurs selon le nombre de cœurs associés à la configuration. C'était moins le cas pour l'application AES.

Nous avons enfin testé notre logiciel de contrôle sur une application avionique de taille moyenne. Les résultats que nous avons obtenus sont mitigés. Le logiciel de contrôle n'est pas efficace vis-à-vis de cette application, mais il n'est pas loin de l'être, au moins sur les configurations à peu de cœurs. Pour que notre concept de logiciel de contrôle soit viable, il sera nécessaire d'explorer un certain nombre de pistes visant à améliorer son efficacité.

Cinquième partie

Conclusion générale

Chapitre 9

Conclusion

Sommaire

9.1 Résumé des travaux effectués 147
9.1.1 Existence du logiciel de contrôle 148
9.1.2 Efficacité du logiciel de contrôle 149
9.1.3 Limites de l'approche 150
9.2 Perspectives 150
9.2.1 Perspectives à court terme 151
9.2.2 Perspectives à long terme 152

9.1 Résumé des travaux effectués

Nous avons abordé dans cette thèse le problème de l'évaluation du pire temps d'exécution (WCET) d'un ensemble de tâches déployées sur un processeur multi-cœurs COTS. Ce problème est connu comme difficile du fait de la présence d'interférences entre les activités électroniques générées par les différents cœurs. La présence d'interférences entraîne une augmentation du temps d'exécution du logiciel embarqué, que l'on ne sait pas correctement borner.

Nous avons montré que les approches applicables aux processeurs COTS doivent combiner une stratégie d'utilisation qui rassemble les restrictions d'usage du processeur, et une stratégie de contrôle qui assure la mise en œuvre de la stratégie d'utilisation. La stratégie d'utilisation communément employée est centrée sur une politique d'arbitrage TDMA, selon laquelle chaque cœur dispose d'une fenêtre temporelle dans laquelle il peut accéder à l'intégralité des ressources communes.

Nous nous sommes intéressés à la mise en œuvre de la stratégie de contrôle dans un logiciel dédié, que nous avons appelé logiciel de contrôle. Nous avons abordé dans notre problématique de thèse la question de son existence et celle de son efficacité.

In fine, l'approche que nous avons développée dans cette thèse permet de rendre un processeur multi-cœurs COTS prédictible, et donc d'appliquer correctement des méthodes

d'évaluation du WCET en respectant les contraintes que nous nous étions fixées à partir de notre contexte industriel, à savoir :

- Rendre prédictible des processeurs COTS, tout en autorisant des configurations comportant un nombre arbitraire de cœurs. Nous avons mis en place une stratégie d'utilisation TDMA, qui assure une isolation temporelle entre l'activité des différents cœurs.
- Pouvoir nous intégrer dans une architecture de systèmes IMA. Nous avons choisi la configuration AMP (Asymetrical Multi-Processing) dans laquelle chaque cœur se comporte d'un point de vue fonctionnel comme un système autonome. Chaque cœur embarque une instance du logiciel de contrôle, un système d'exploitation mono-cœur et un ensemble de partitions ARINC 653.
- Pouvoir réutiliser du logiciel existant. Cela concernait des applications avioniques et dans une moindre mesure le système d'exploitation. Les mécanismes de contrôle sont invisibles du logiciel utile, ce qui assure la rétro-compatibilité vis-à-vis du logiciel existant.

Nous abordons dans la suite de cette section un résumé des contributions sur l'étude de l'existence du logiciel de contrôle, puis celle de son efficacité.

9.1.1 Existence du logiciel de contrôle

Pour répondre à la problématique d'existence du logiciel de contrôle, nous avons effectué une analogie avec le cadre de virtualisation de Popek et Goldberg [69]. À partir de cette analogie, nous avons défini notre logiciel de contrôle sur la base des propriétés de contrôle des ressources et d'équivalence définies par Popek et Goldberg. Nous avons ensuite proposé le théorème de contrôlabilité, qui généralise le théorème de virtualisation de Popel et Goldberg [69]. Le théorème de contrôlabilité donne une condition d'existence du logiciel de contrôle sur un processeur donné.

L'application du théorème de contrôlabilité se traduit par un certain nombre d'analyses à effectuer sur le cœur. Nous avons proposé une méthodologie sur un modèle de cœur simplifié. Cette méthodologie comporte trois étapes, centrées respectivement sur :

- l'identification d'évènements matériels sensibles, i.e. susceptibles d'entraîner l'émission d'accès vers les ressources communes.
- L'évaluation de la contrôlabilité des évènements sensibles, i.e. la possibilité de prévenir leur occurrence en levant une exception dirigée vers le logiciel de contrôle. Les cas d'évènements non contrôlables sont éliminés en introduisant une propriété invariante sur l'état du cœur, que le logiciel de contrôle doit vérifier.
- La construction de séquences de contrôle applicables à chaque évènement contrôlable.

L'application de cette méthode permet de vérifier les hypothèses d'application du théorème de contrôlabilité. En pratique, elle permet également de construire une partie du logiciel de contrôle en définissant d'éventuelles règles de conception ainsi que le détail des séquences de contrôle.

Nous avons appliqué cette méthodologie dans un cas d'étude basé sur le processeur P5040 de la gamme QorIQ commercialisée par Freescale. Ces processeurs embarquent des cœurs e5500 de la famille PowerPC. Nous avons montré que, en tenant compte de quelques restrictions sur le comportement du logiciel utile et sur le développement du logiciel de contrôle, le cœur e5500 satisfait les hypothèses du théorème de contrôlabilité. Il est donc possible d'y mettre en œuvre un logiciel de contrôle, que nous avons spécifié en appliquant notre méthodologie.

À partir de cette spécification, nous avons réalisé un prototype de logiciel de contrôle, appelé Marthy, que nous avons intégré dans un hyperviseur. Cela nous a permis au cours de cette étude d'étendre la stratégie de contrôle à des classes de logiciels diverses, y compris à des systèmes d'exploitation, avec un effort de portage réduit. Nous avons dans un second temps étudié l'efficacité de ce prototype.

9.1.2 Efficacité du logiciel de contrôle

Nous avons étudié dans un second temps l'efficacité de notre prototype de logiciel de contrôle sur un ensemble d'applications données. Nous avons introduit la notion d'efficacité comme le ratio entre les temps d'exécutions d'une application dans une configuration multi-cœurs, par rapport à une exécution de référence de la même application dans un contexte mono-cœur, sans logiciel de contrôle. Nous considérons qu'un ratio égal à deux constitue un bon compromis entre des temps d'exécution dégradés individuellement pour chaque application, mais permettant l'utilisation de processeurs multi-cœurs sans entraîner de dégradation supplémentaire. Cette valeur reste subjective, mais constitue un objectif permettant de tirer parti du parallélisme au delà de deux cœurs.

Nous avons observé une dégradation du temps d'exécution des applications que nous avons évalué. Cela n'est pas surprenant, dans la mesure où le logiciel de contrôle est instrusif, et altère le comportement du cœur à un point où il est possible de le décrire par des modèles équivalents, dans lesquels les lignes de caches font quatre kilo-octets. Ces caches ont une pénalité de rechargement élevée, mais un taux de cache miss plus faible. Un phénomène compense donc l'autre.

L'impact sur le temps d'exécution du logiciel utile est très variable. Parmis les cas que nous avons évalués, nous avons distingué trois classes d'applications :

- Les applications pour lesquelles la baisse du taux de cache miss compense la pénalité plus élevée de remontée de données dans le cache. Pour ces applications, nous rentrons dans le critère d'efficacité. Les applications de petite taille et travaillant de manière linéaire sur des quantités de données supérieures à quelques dizaines de kilo-octets, à l'instar de l'application de chiffrement AES que nous avons étudiée, semblent appartenir à cette catégorie d'applications.
- Les applications qui doivent disposer de l'intégralité de leur données d'entrée pour fonctionner correctement. Le calcul de FFT en est un exemple. Ces applications ne sont pas efficaces lorsque la taille des données dépasse celle du cache, dans la mesure où elles passent la majorité de leur temps d'exécution à attendre des

autorisations d'accéder aux ressources communes. Nous considérons que ces applications sont hors du domaine d'efficacité du logiciel de contrôle.
- Les applications intermédiaires, qui ne rentrent pas dans le domaine d'efficacité, mais n'en sont pas loin. Sur ces applications, on peut chercher des biais dans le logiciel de contrôle permettant d'améliorer leur temps d'exécution. Nous développons certains de ces biais dans la section 9.2.1. L'application avionique que nous avons testée entre dans cette catégorie d'applications.

9.1.3 Limites de l'approche

L'approche que nous avons proposée répond correctement à la problématique que nous avons étudié dans cette thèse. Toutefois, elle souffre de plusieurs limites, que nous développons ci-après.

Nous avons proposé une méthode de construction du logiciel de contrôle se basant sur les caractéristiques d'un cœur. Comme nous l'avons constaté dans le cas d'étude sur le cœur e5500, le niveau de détails requis est élevé. Nous avons pu l'atteindre grâce à une documentation des cœurs fournie par Freescale à un niveau de détails élevé. Cependant, nous ne sommes pas convaincus que ce même niveau puisse être atteint à l'avenir sur des cœurs d'autres fabricants, ou sur de futurs cœurs, qui seront plus complexes. Or, la validité de la méthode repose sur le fait que la caractérisation du cœur est correcte, ce que nous ne pouvons pas garantir car le processeur est un COTS.

D'autre part, la méthode que nous avons proposé est générique, mais sa portabilité se limite aux principes généraux. Par exemple, sur un cœur avec une MMU et deux niveaux de caches, on retrouvera presque sûrement l'exigence de cache intégralement référencé par la MMU, et de cache L_1 inclus dans le cache L_2. Cependant, le principal effort se situe dans les détails, comme la caractérisation des instructions exotiques, dont l'implémentation est propre à chaque cœur. L'analyse de ces détails n'est pas portable, et doit être revue pour chaque cœur.

Enfin, notre approche diminue les performances du logiciel embarqué dans des proportions qui fragilisent son utilisation dans un contexte industriel. Un travail d'optimisation sur notre prototype est nécessaire pour aller dans ce sens. Nous développons ce point dans les sections suivantes, qui abordent les perspectives de ces travaux.

9.2 Perspectives

Nous distinguons certaines perspectives à court termes, qui peuvent être explorées dans la continuité de ces travaux. Ces perspectives visent principalement à rendre l'approche plus robuste pour une éventuelle poursuite de ces travaux dans un cadre industriel. À l'inverse, d'autres perspectives appellent à des travaux plus ouverts, qui ont vocation à être effectués à plus long terme. Nous donnons une vue d'ensemble de ces perspectives dans la suite de cette section.

9.2.1 Perspectives à court terme

Nous voyons à ces travaux des perspectives à court termes sur les sujets suivants :

- L'amélioration de l'efficacité de Marthy. Il s'agit de la perspective à explorer en priorité.
- L'extension du support de Marthy en terme de processeurs –dans la gamme QorIQ– et de périphériques.
- L'intégration d'un système d'exploitation ARINC 653 dans le logiciel utile.

Nous développons ces pistes dans la suite de cette section.

Optimisation du prototype

À court terme, il est nécessaire de chercher à améliorer les performances de Marthy. La recherche d'efficacité doit cependant venir de Marthy, et non de la manière de développer le logiciel utile. Des recommandations en ce sens ne seraient en effet pas compatibles avec notre objectif de réutilisation de logiciel existant.

Nous souhaitons nous intéresser en priorité aux pistes suivantes :

Verrouillage de pages dans le cache. Certaines pages du logiciel utile sont déchargées et rechargées un certain nombre de fois au cours de l'exécution du logiciel utile. Nous pensons qu'il est pertinent de les identifier sur des traces d'exécution du logiciel utile collectées par Marthy, puis de mettre en place un mécanisme de verrouillage logique dans le cache de telle sorte à les rendre persistentes.

Pré-chargement de sections du logiciel utile. Un piste intéressante consiste à proposer au logiciel utile un format de table de données permettant à Marthy de pré-charger en une fois certaines sections de code et/ou de données du logiciel utile.

Extension du support de Marthy

À l'heure actuelle Marthy est orienté pour effectuer du contrôle d'accès à la mémoire principale. Nous envisageons à court terme d'étendre son contrôle à quelques périphériques qui sont utilisés dans l'avionique. Un bus PCIe, des moteurs DMA et des bus à bas débit comme CAN, SPI ou I2C sont de bons candidats. Ce support est nécessaire pour arriver à des démonstrateurs plus complets.

Il est également intéressant d'étendre le support de Marthy à d'autres processeurs. Nous considérons par exemple les processeurs de la série T de la gamme QorIQ. Ainsi, le T4240 dispose de trois clusters de quatre cœurs partageant deux méga-octets de cache L_2. Il est possible de partitionner ces caches pour nous ramener à une configuration proche du P5040, mais avec 12 cœurs. L'application d'une stratégie d'utilisation TDMA doit rendre le processeur prédictible de la même manière que le P5040.

Intégration d'un système d'exploitation au sein du logiciel utile

L'intégration d'un système d'exploitation dans le logiciel utile était un objectif que nous avions de longue date. Au début de ces travaux, nous nous intéressions plus aux techniques de virtualisation que de contrôle dans la perspective d'avoir un hyperviseur pour l'avionique. Même si ce n'est pas le point central de ces travaux, notre réalisation est bien un hyperviseur, sur lequel nous avons fait démarrer POK, qui est un système d'exploitation ARINC 653 open-source [32].

Nous abordons en tant que perspective les points suivants :

Intégration d'une tâche de maintenance dans un ordonnancement ARINC 653. Le logiciel de contrôle peut embarquer un serveur de maintenance, qui effectuera des opérations de maintenance, et de communication par des mécanismes de polling.

Nous pensons qu'il est pertinent de définir le serveur de maintenance comme une tâche apériodique, à laquelle on associe une durée maximale entre deux exécutions. cela permet à un système d'exploitation de prévoir son ordonnancement, en sachant qu'une partie du temps de calcul sera prise par le logiciel de contrôle. Le non respect de la durée inter-activations peut entraîner la levée d'une exception de watchdog, afin de prévenir d'éventuelles défaillances du système d'exploitation.

Partitionnement du cache L_2. Le système d'exploitation va partager l'espace disponible dans le cache avec le logiciel applicatif. L'un et l'autre vont interférer en chargeant et déchargeant des données qui ne leur appartiennent pas. Il est possible de remédier à ce défaut en implémentant un mécanisme de partitionnement de cache. Ainsi, l'application peut avoir une section du cache dont l'état ne dépend que de son exécution, ce qui la protège du système d'exploitation, et réciproquement.

Nous avons décrit les perspectives que nous envisageons à court termes. Au cours de ces travaux, des problématiques plus ouvertes ont émergé. Nous les développons dans la section suivante.

9.2.2 Perspectives à long terme

Les perspectives à long terme portent sur des sujets assez variés. Ce sujets constituent à la fois une ouverture scientifique et industrielle pour ces travaux. Ils sont décrits à travers les points suivants :

- La robustesse du contrôle logiciel mis en œuvre sur un processeur COTS
- L'amélioration de la stratégie d'utilisation
- La prise en compte du logiciel de contrôle dans l'évaluation du WCET, amenant à la notion de machine virtuelle à performances garanties

Nous développons ces points dans la suite de cette section.

Robustesse d'un logiciel de contrôle déployé sur un processeur COTS

Une des limites que nous avons relevé dans notre approche est le niveau de confiance que nous plaçons dans la contrôlabilité d'un cœur. Sur le cas d'étude que nous avons développé, nous avons été aussi loin qu'il nous semblait possible dans la caractérisation du comportement du cœur. Il nous semble probable que certains évènements sensibles n'aient pas été correctement identifiés, faute d'informations pertinentes à disposition.

Une piste intéressante consiste donc à définir et étudier la notion de *robustesse* de la stratégie de contrôle. Cette piste viserait à évaluer le comportement d'un processeur lorsque le logiciel de contrôle a une bonne couverture des évènements sensibles, à l'exception de quelques cas qui n'ont pas été traités, par manque d'information ou par souci d'efficacité.

Amélioration de la stratégie d'utilisation d'un processeur COTS

Nous avons adopté dans notre approche une stratégie d'utilisation d'un processeur centrée sur la politique TDMA pour le partage des ressources communes. Cette politique est basée sur l'hypothèse, conservative, que les ressources du processeur constituent une ressource atomique, et que tout accès concurrent peut mener à des interférences. Dès lors, chaque cœur est cloisonné à une fenêtre temporelle dans laquelle il a un accès exclusif aux ressources communes.

Or, on peut constater que les processeurs multi-cœurs COTS sont capables de traiter certaines activités concurrentes sans que l'on puisse observer des interférences. Il semble donc intéressant de chercher à améliorer la stratégie d'utilisation pour tenir compte de ces capacités de traitement parallèle du processeur, tout en restant conservatif lorsque le niveau d'information sur le processeur fait défaut.

Une piste qui semble prometteuse consiste à proposer une résolution en deux temps.

- Dans une première étape on cherchera à construire des ensembles d'accès pouvant être effectués en parallèle sans générer d'interférences.
- Dans un second temps, on posera un problème d'ordonnancement dont l'objectif est de satisfaire un ensemble d'exigences en termes de latence et de débit sur chaque couple constitué d'un cœur et d'une ressource. La résolution de ce problème d'ordonnancement consiste à sélectionner des ensembles de chemins déterminés précédemment, et à leur associer une fenêtre ou plusieurs fenêtres temporelles.

Prise en compte du logiciel de contrôle dans l'évaluation du WCET

Nous avons cherché dans ces travaux à mettre en place un domaine d'usage dans lequel le processeur est prédictible, ce qui rend possible l'évaluation du WCET du logiciel utile. Cependant, l'utilisateur final est conscient d'être exécuté par dessus un logiciel de

contrôle, pour lequel il doit disposer d'un niveau d'information élevé pour mener son analyse de WCET.

Il semble intéressant de proposer une méthode d'évaluation de WCET dans laquelle le logiciel de contrôle est enfoui, i.e. l'utilisateur n'a pas conscience que son application est exécutée sous le contrôle de notre logiciel. Une piste intéressante consiste à proposer une notion de *modèle de cœur équivalent*, qui reproduise l'action du logiciel de contrôle sur les ressources internes du cœur.

Ainsi, si on se place dans notre cas d'étude, un modèle de cœur équivalent contiendrait un cache L_2 de 320K dont les lignes font quatre kilo-octets, associatif à cinq voies, avec une politique de remplacement FIFO. Les caches L_1D et L_1I font 32K, associatifs à huit voies, et ont des lignes de 64 octets. Leur contenu est inconnu lorsque le cache L_2 est mis à jour. Ce type de modèle peut être implémenté dans une bibliothèque de calcul de WCET comme Otawa [17], qui est open-source.

Dans un second temps, on peut proposer la notion de *machine virtuelle à performances garanties* comme étant un cadre dans lequel on peut effectuer une évaluation du WCET du logiciel utile. Ce cadre aurait vocation à masquer le détail des mécanismes implémentés dans le logiciel de contrôle, mais également la manière dont la politique TDMA est configurée. Cela permettrait d'une part de faire évoluer les politiques de partage des ressources tout en conservant des performances garanties à chaque machine virtuelle, et donc un WCET stable, et d'autre part de conserver le WCET en passant d'un processeur à un autre, dans la mesure où les processeurs partagent les mêmes cœurs.

Annexes

Annexe A

Complément à l'étude de cas sur le P5040

A.1 Description du P5040

l'architecture du P5040, représentée sur la figure A.1, est adaptée pour traiter de manière efficace et sécurisée différents flux concurrents sur un réseau ethernet à très haut débit.

Outre quatre cœurs e5500, le P5040 intègre les composants suivants :

CoreNet. Il s'agit de l'interconnexion principale du processeur. Elle est présentée comme un réseau switché avec un arbitrage centralisé. Outre le transfert des transactions des maîtres (cœurs, bus PCIe et SRIO, DMA) vers les esclaves, elle est en charge du maintien de la cohérence de caches et du support pour les opérations atomiques. Elle s'appuie pour cela sur la notion de domaine de cohérence, qui associe à une plage d'adresses un ensemble de périphériques qui doivent être notifiés lorsque des messages de cohérence de cache circulent.

A quelques détails près, CoreNet est une boîte noire. Sa topologie et sa politique d'arbitrage ne sont pas décrites. Son protocole de communication a fait l'objet d'un dépot de brevet par Freescale [33] et vise à éviter les pertes de transactions et les interblocages. Pour cela, il garantit qu'une transaction complexe nécessitant plusieurs sous-transactions ne sera autorisée que si l'intégralité des ressources nécessaires pour propager toutes les sous-transactions sont disponibles. Un exemple de transaction complexe est décrit sur la figure A.2.

Contrôleurs DDR. Les contrôleurs DDR implémentent le protocole de communication associé à la mémoire principale. Pour cela, ils gèrent les ouvertures de banques mémoire, ainsi que la sélection des lignes et des colonnes dans les banques ouvertes. De plus, ils assurent un ordonnancement des requêtes entrantes, afin de minimiser le nombre d'ouvertures et de fermetures de banques, qui sont des opérations coûteuses pour le temps de traitement des requêtes. L'algorithme d'ordonnancement

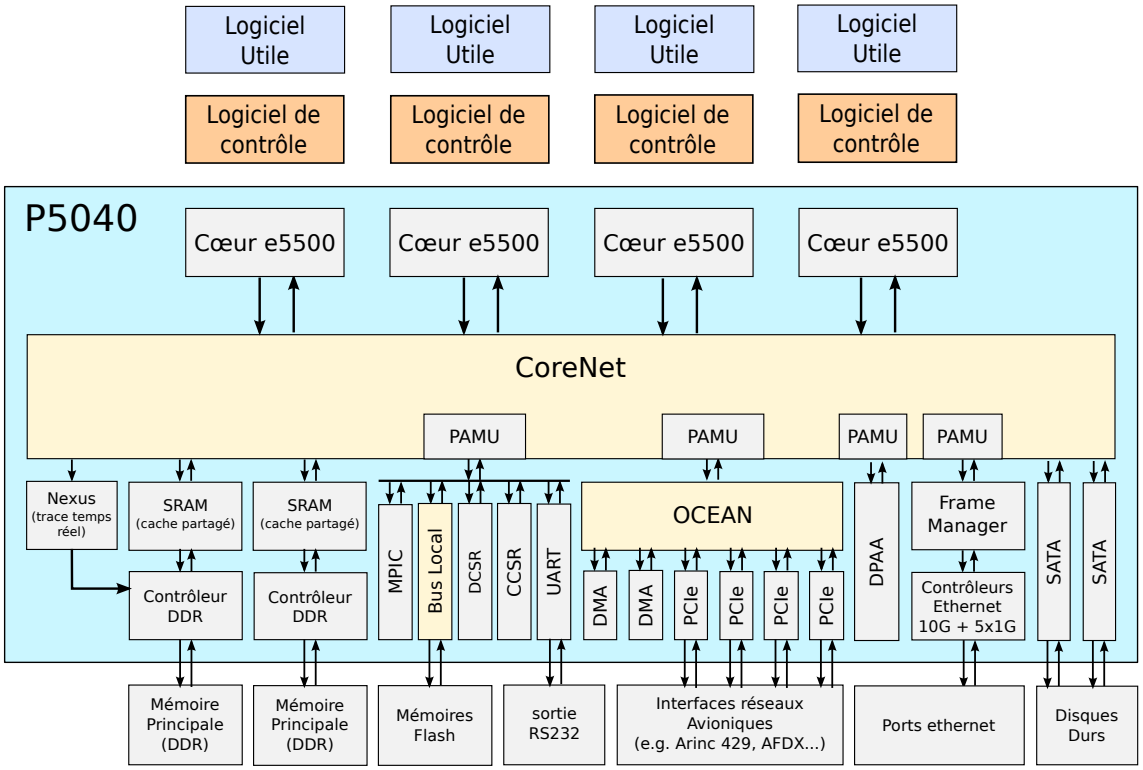


FIGURE A.1 – Vue d'ensemble du processeur P5040 et des périphériques associés

n'est pas documenté et peut être complexe. Les contrôleurs DDR se chargent également d'initier les opérations périodique de rechargement de la mémoire.

Chaque contrôleur DDR contient quatre ***chip select***, qui maintiennent chacun un contexte de banque mémoire ouverte ainsi qu'une ligne pré-chargée dans cette banque. L'affectation d'une requête à un contrôleur mémoire, puis à un chip select dépend de son adresse, et l'entrelacement choisi. Ce dernier point vise à affecter des plages d'adresses proches à des contrôleurs et/ou des chip select différents pour maximiser le nombre de banques ouvertes dans ces plages, ce qui offre de meilleurs temps d'accès au logiciel qui les utilise. Dans le cas contraire, c'est-à-dire sans entrelacement, les plages mémoire associées à un chip select sont contigües. Dans le cas d'un système partitionné, cette seconde option est la plus courante car elle favorise la ségrégation des contextes mémoire entre partitions, i.e. des banques et lignes ouvertes.

Dans notre cas d'étude, nous adoptons une configuration où l'entrelacement est désactivé au niveau des contrôleurs DDR et au niveau des chip select.

Cache Partagé. On trouve en amont de chaque contrôleur DDR un cache partagé de niveau L3 et d'une capacité de 1M. La capacité totale est donc de 2M. Chaque cache est organisé en 32 voies, qui peuvent être affectées à différentes plages mémoires. On parle de cache partitionné. Chaque cache dispose de quatre ports

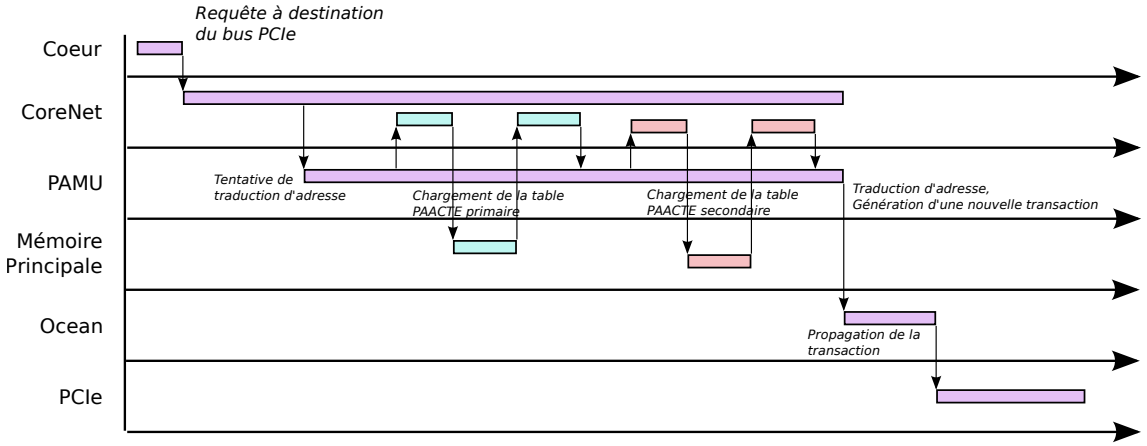


FIGURE A.2 – Exemple d'activité nécessitant l'émission de plusieurs requêtes sur CoreNet

en lecture et un port en écriture, ce qui leur permet de traiter plusieurs transactions concurrentes sans interférences.

Nous considérons dans notre étude que les caches partagés sont désactivés, même si un usage possible consisterait à verrouiller certaines données dans ce cache. Cela permettrait de prévoir des temps d'accès plus courts pour ces données en particulier.

Trace Nexus. Les processeurs de la gamme QorIQ disposent d'une IP permettant de collecter de manière non intrusive une trace de certains évènements survenant sur le processeur. Ces évènements portent sur l'exécution du logiciel sur les CPU, mais également sur l'activité des ressources partagées. On peut par exemple disposer de la liste des accès à un contrôleur DDR, ainsi que des informations sur le traitement de ces accès, comme une ouverture de banque. Il existe également une trace sur CoreNet, sur Ocean et le DPAA. La trace peut être dirigée vers une sortie dédiée, appelée Aurora, puis sur une sonde de trace. Elle peut également être envoyée sur un contrôleur DDR et stockée dans une plage mémoire prévue à cet effet.

Dans notre contexte, la trace Nexus est une manière pertinente de constater d'une part que l'activité électronique du processeur est correctement spécifiée, i.e. tous les évènements que l'on observe sont reliés à une activité bien définie et connue. D'autre part, elle permet de vérifier que sur une exécution, la stratégie d'utilisation telle qu'elle a été définie a été respectée.

CCSR (Configuration, Control and Status Registers). Il s'agit d'un espace de configuration des ressources communes du processeur, stocké sur une SRAM. La capacité est de 16M organisée en registres. Même si les 16M ne sont pas intégralement utilisés, l'espace de configuration est très important, et certains registres ne sont pas documentés.

DCSR (Debug, Control and Status Registers). Il s'agit d'un espace de registres dédiés à la manipulation de moniteurs de performances qui comptent les occurrences de certains évènements. Il sert également à programmer la trace Nexus. On relèvera

que certains composants disposent de fonctionnalités de trace, mais ne sont pas documentés.

PAMU (Peripheral Access Management Unit). Il s'agit d'une IP ayant un rôle similaire à celui d'une MMU. On rencontre d'ailleurs souvent ce type d'IP sous le nom d'IOMMU. Elle effectue la traduction entre l'espace d'adressage des périphériques et l'espace d'adressage physique, vu du processeur. En outre elle peut effectuer de la vérification de droits d'accès ainsi que de l'adaptation de protocole pour la compatibilité avec le protocole de CoreNet. Les tables de traduction des PAMU, analogues aux TLB dans une MMU, se rechargent automatiquement à partir de table des pages stockées dans la mémoire principale.

Bus Local. Le bus local, ou ELBC, relie à CoreNet un certain nombre de mémoires flash, sur lesquelles sont stockées entre autres les images des exécutables.

UART, SPI, I2C, USB, GPIO. Ces sorties à bas débit sont reliées à CoreNet via un chemin commun avec le CCSR, le DCSR et le bus local. Dans notre étude, nous utilisons une UART pour la console de debug.

Ocean. Il s'agit d'une interconnexion qui est un crossbar. Il relie divers contrôleurs de périphériques (PCIe, SRIO) ainsi que les DMA à CoreNet.

MPIC (Multicore Programmable Interrupt Controller). Il s'agit du contrôleur d'interruptions partagé. Selon sa configuration, il redirige les lignes d'interruptions vers un ou plusieurs cœurs. Il est également capable d'initier certaines opérations comme le redémarrage d'un cœur.

DMA (Direct Memory Access). Les contrôleurs DMA sont des accélérateurs programmables pour la copie de données dans l'espace adressable. Ils sont typiquement utilisés pour libérer un cœur pour recopier de grosses quantités de données.

PCIe (Peripheral Communication Interface). Ce bus à haut débit a vocation à relier le processeur à divers périphériques, dont des cartes d'interface avec les réseaux avioniques (ARINC 429, AFDX...), une éventuelle carte graphique, etc... Selon sa configuration, un bus PCIe peut initier des accès en tant que maître ou en tant qu'esclave sur l'interconnexion.

DPAA (Data Path Acceleration Architecture). Ce composant regroupe un ensemble d'IP dont la vocation est de faciliter le traitement de paquets reçus sur le réseau. Ces IP portent entre autre sur le chiffrement/déchiffrement de paquets (Sec. Engine), la gestion de files d'attentes (Queue Manager), de buffers en mémoire principale (Buffer Manager) ou encore de pattern matching. Dans notre étude, son utilisation n'est pas envisagée.

Frame Manager. Ce composant gère les contrôleurs réseau intégrés dans le processeur. Il récupère les trames ethernet reçues, les classe selon leur protocole et leur fait subir un ensemble de traitements en se basant sur le DPAA. Il peut être amené à diriger un flux de paquets vers un cœur, à lever des interruptions sur l'arrivée de certains paquets, mais également recopier des paquets entrants dans la mémoire principale, et inversement. Ce composant est connu pour être complexe, et son

utilisation n'est pas envisagée dans un cadre avionique. Nous ne le considérons pas dons notre étude.

SATA (Serial Advanced Technology Attachment). Cette interface a vocation à relier un ou plusieurs disques durs au processeur. Dans le cadre de notre étude, nous n'envisageons pas l'utilisation de cette interface, même si elle constitue une perspective intéressante.

SRIO (Serial RapidIO). Ce contrôleur de bus est proposé par le processeur. Dans notre cas d'étude, son utilisation n'est pas envisagée.

A.2 Complément à la première analyse du jeu d'instruction PowerPC

TABLE A.1 – Evènements associés aux instructions PowerPC supportées par le cœur [85, section 6.4]

Mnémoniques	**Noms**	**Évènements associés**
Lectures et écritures de nombres entiers dans l'espace adressable [43, Section 3.4.3.2]		
lbz, lbzu, lbzux, lbzx **lbepx** **lhz, lhzu, lhzux, lhzx** **lha, lhax, lhau, lhaux** **lharx, lhbrx, lhdx, lhepx** **lwz, lwzu, lwzux, lwzx** **lwa, lwax, lwaux** **lwarx, lwbrx, lwdx,** **lwepx** **ld, ldu, ldux, ldx** **ldarx, ldbrx, lddx, ldepx** **lmw**	**l** : load **a** : algebraic **ar** : and reserve **b** : byte **br** : byte reversed **d** : double word **dd** : double word with decoration **ep** : external PID **h** : half word **hd** : half word with decoration **m** : multiple **u** : update **w** : word **wd** : word with decoration **x** : indexed **z** : zero	*load(@, read)*
stb, stbu, stbux, stbx **stbcx, stbdx, stbepx** **sth, sthu, sthux, sthx** **sthbrx, sthcx, sthdx,** **sthepx** **stw, stwux, stwx** **stwbrx, stwcx, stwdx,** **stwepx** **std, stdu, stdux, stdx** **stdbrx, stdcx, stddx,** **stdepx** **stmw**	**st** : store **b** : byte **bd** : byte with decoration **br** : byte reversed **c** : conditional **d** : double word **ep** : external PID **h** : half word **hd** : half word with decoration **m** : multiple **u** : update **w** : word **wd** : word with decoration **x** : indexed	*store(@, write)*

TABLE A.2 – Evènements associés aux instructions PowerPC supportées par le cœur (suite) [85, section 6.4]

Mnémoniques	Noms	Évènements associés
Lectures et écritures flottantes dans l'espace adressable, à destination de la FPU		
lfs, lfsx, lfsu, lfsux **lfd, lfdx, lfdu, lfdux** **lfddx, lfdepx stfs, stfsx,** **stfsu, stfsux**	**l** : load **st** : store **d** : double precision **dd** : double precision with decoration **ep** : external PID **f** : floating-point **s** : single precision **x** : indexed	*load(@, read)*
stfd, stfdx, stfdu, stfdux **stfddx, stfdepx, stfiwx**		*store(@, write)*
Barrières mémoire [43, Section 3.4.8]		
isync	instruction synchronize	Pas d'évènement associé
mbar	Memory barrier	*sync*
sync	Memory Synchronize	*sync*
lwsync (sync 1)	Lightweight synchronize	Pas d'évènement associé
Gestion de la MMU [43, Section 3.4.11.3]		
tlbilx, tlbre, tlbwe, tlbsx	**tlb** : TLB **il** : invalidate local **re** : read entry **we** : write entry **s** : search **x** : indexed	Pas d'évènement associé
tlbsync	TLB synchronize	*sync*
tlbivax	TLB invalidate virtual address indexed	*snoop*

TABLE A.3 – Evènements associés aux instructions PowerPC supportées par le cœur [85, section 6.4] (suite et fin)

Mnémoniques	**Noms**	**Évènements associés**
Instructions de gestion du cache [43, Section 3.4.10.1]		
dcba, dcbal, dcbz, dcbzl	**dcb** : data cache block **a** : allocate **l** : line **z** : zero	*store (@, allocate)*
dcbt, dcbtst, dcbtls, dcbtstls	**dcb** : data cache block **t** : touch **st** : store **ls** : lock set	*load(@, touch)*
dcblc	data cache block lock clear	Pas d'évènement associé
dcbf, dcbst	data cache block flush, store	*load(@, flush)*
dcbi	data cache block invalidate	*load(@, invalidate)*
icbt, icbtls	instruction cache block touch, touch and lock set	*load(@, touch)**
icblc	instruction cache block lock clear	Pas d'évènement associé
icbi	instruction cache block invalidate	*load(@, invalidate)**
Émission et acquittement d'interruptions inter-cœurs		
msgsnd	Message Send	*snoop*
msgclr	Message Clear	Pas d'évènement associé

* Les instructions *icbi*, *icbt* et *icbtls* sont implémentées comme des *load* vis-à-vis de la traduction d'adresse et des droits d'accès [43, Table 4-9, Note].

A.3 Arbres d'activités des différents composants du cœur e5500

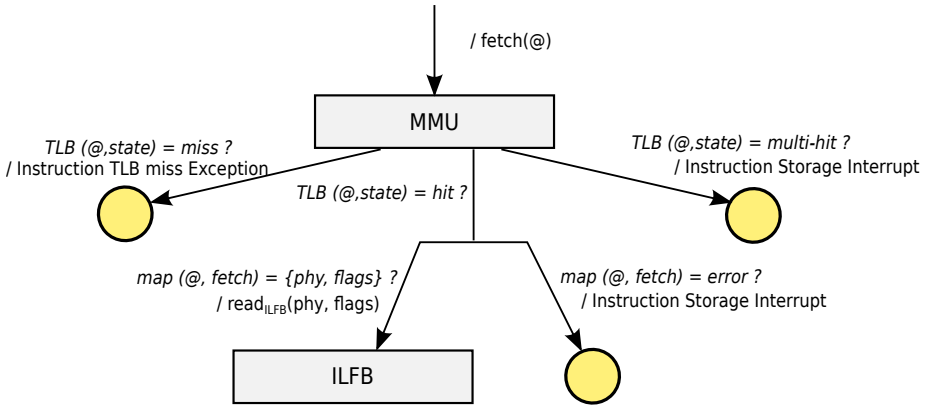
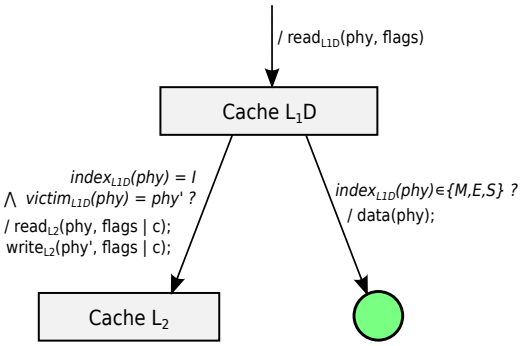
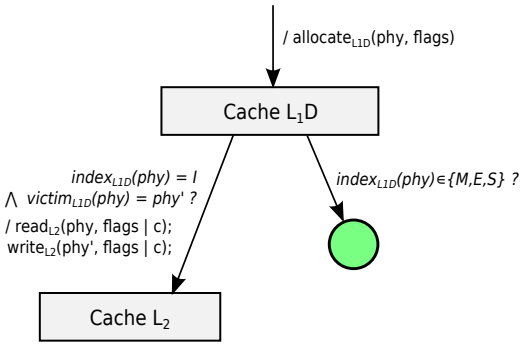


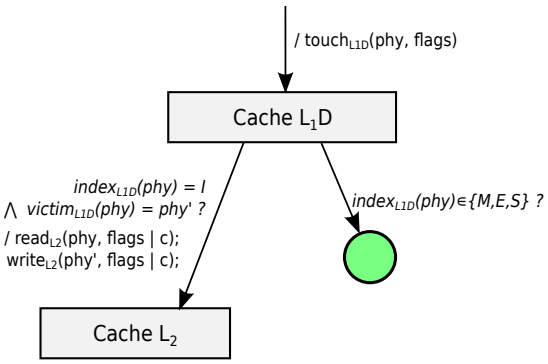
FIGURE A.3 – Arbre d'activité d'une MMU traitant un évènement de type *fetch*

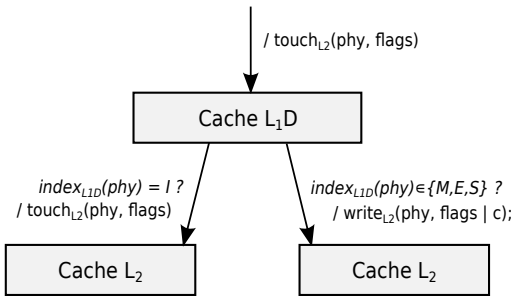


(a) Traitement d'un évènement de type *read*

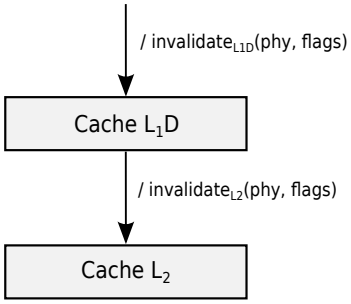


(b) Traitement d'un évènement de type *allocate*

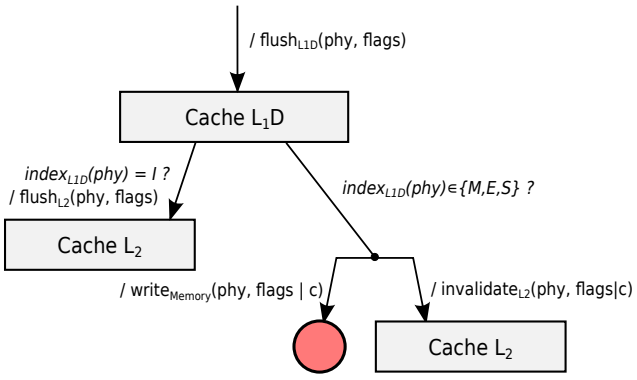




(c) Traitement d'un évènement de type *touch*

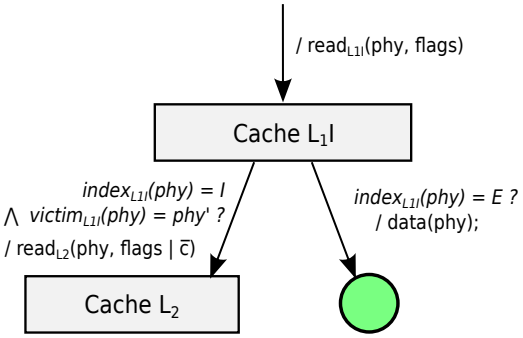


(d) Traitement d'un évènement de type *invalidate*

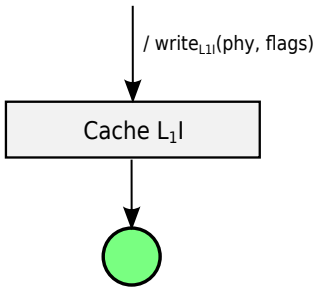


(e) Traitement d'un évènement de type *flush*

FIGURE A.4 – Arbres d'activités associés au cache L1D

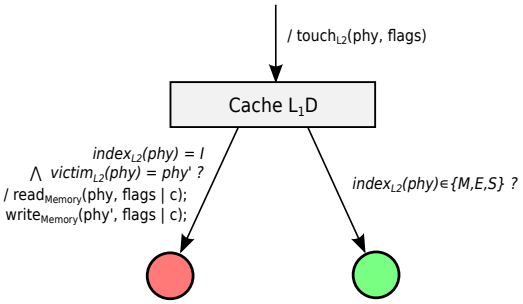


(a) Traitement d'un évènement de type *read*

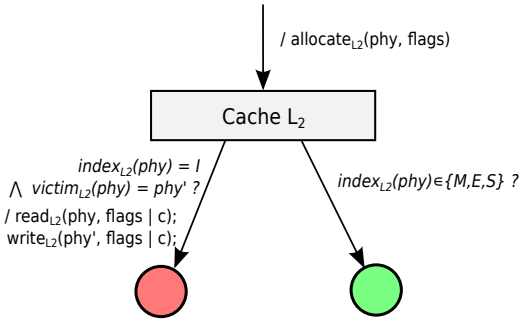


(b) Traitement d'un évènement de type *write*

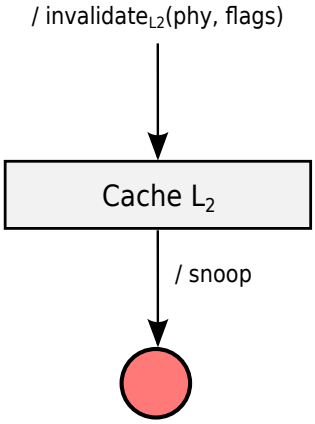
FIGURE A.5 – Arbres d'activités associés au cache L1I



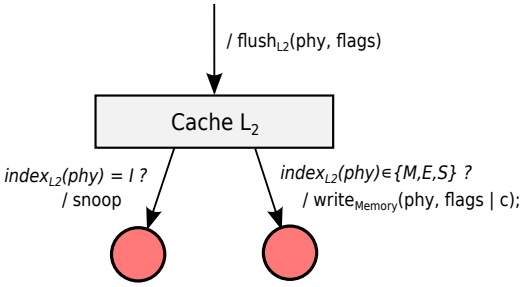
(a) Traitement d'un évènement de type *touch*



(b) Traitement d'un évènement de type *allocate*



(c) Traitement d'un évènement de type *invalidate*



(d) Traitement d'un évènement de type *flush*

FIGURE A.6 – Arbres d'activités associés au cache L2

Annexe B

Techniques de virtualisation

Sommaire

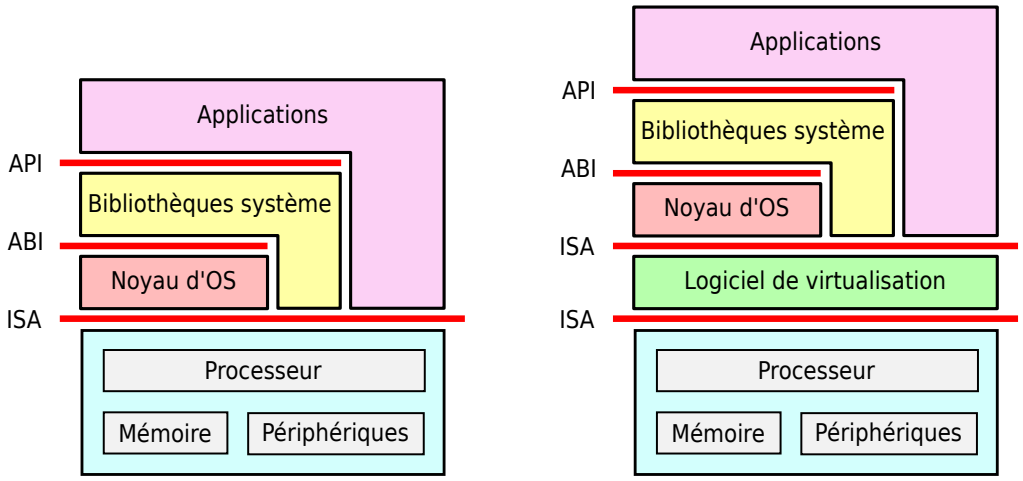
B.1 Vue générale du concept de virtualisation 169
B.2 Classification des hyperviseurs existants 172
B.2.1 Classification par environnement d'exécution 172
B.2.2 Classification par technique de virtualisation 173
B.3 Concepts implémentés dans un hyperviseur 176
B.3.1 Virtualisation de l'espace adressable 178
B.3.2 Virtualisation des périphériques 179

La possibilité de réutiliser des composants logiciels existants est une problématique centrale dans les systèmes IMA. Elle s'applique dans un contexte multi-cœurs à la fois aux applications avioniques et aux systèmes d'exploitation ARINC 653. Ce n'est pas une contrainte dure, un effort de portage léger du logiciel est acceptable, par exemple l'altération d'une adhérence matérielle. Les composants logiciels que nous considérons sont conçus pour une exécution mono-cœur, et ont vocation à le rester. La virtualisation est une technique qui est apparue dans les années 2000 dans le domaine embarqué. Elle vise à apporter une solution pérenne à la problématique de réutilisation de composants logiciels.

Les solutions de virtualisation, les propriétés qu'elles offrent et le cadre dans lequel elles s'appliquent sont très diverses. Nous donnons dans cette section une vue d'ensemble des techniques existantes.

B.1 Vue générale du concept de virtualisation

Le terme *virtualisation* fait référence à de nombreux concepts qui touchent à la fois le domaine grand public, le domaine de l'informatique dématérialisée et le domaine embarqué. Dans le langage courant, on parle de virtualisation ou d'émulation de périphérique, de réseau, ou encore d'une machine complète. Une définition générale du concept de



(a) Décomposition d'une système logiciel en niveaux d'abstraction séparés par des interfaces

(b) Simulation d'une interface de bas niveau par un logiciel de virtualisation

FIGURE B.1 – Intégration d'un logiciel de virtualisation selon Smith et Nair [89]

virtualisation a été donnée par Smith et Nair [89], que nous illustrons sur la figure B.1. Cette définition se base sur la décomposition du logiciel embarqué en niveaux d'abstractions séparés par des interfaces. Celles ci vont masquer la complexité du fonctionnement du matériel et du logiciel de plus bas niveau. Ces interfaces sont les suivantes :

Le jeu d'instructions (ISA). Il s'agit de l'ensemble des instructions exécutables par le processeur. Le jeu d'instructions masque la complexité de l'architecture du processeur, le logiciel étant représenté comme une séquence d'instructions qui intéragit avec le processeur par l'espace adressable.

Les appels systèmes (ABI). Il s'agit d'une interface qui décrit les services offerts par le noyau du système d'exploitation, habituellement exécuté au niveau de privilège superviseur. Ces services sont utilisables par les couches logicielles de plus haut niveau, qui sont exécutées à un niveau de privilège moindre. On rencontrera les applications, les drivers et les bibliothèques systèmes. L'appel à un service via l'ABI passe par un appel système, qui est une exception. Le passage d'arguments est possible par une convention d'utilisation des registres.

Les interfaces de programmation (API). Il s'agit de la description des méthodes implémentées dans des bibliothèques tierces, utilisables par le logiciel lors de la phase de compilation. L'API permet d'utiliser une bibliothèque comme une boîte noire.

Il y a donc une relation de compatibilité entre les niveaux d'abstraction et les interfaces qui leur sont associées. Un composant à un certain niveau d'abstraction peut être utilisé par n'importe quelle interface déployée à un niveau d'abstraction plus élevé. Les interfaces sont habituellement hiérarchisées. Smith et Nair définissent le concept de virtualisation de la manière suivante, en inversant cette hiérarchie :

Définition B.1 (Virtualisation)**.** *La virtualisation correspond à la mise en place à un niveau d'abstraction d'une interface présente à un niveau d'abstraction moins élevé.*

Cette définition couvre l'intégralité des approches de virtualisation que l'on peut rencontrer dans la littérature et dans le monde industriel. Elle permet de représenter la notion de virtualisation de périphérique, de mémoire ou de réseau sur le même plan que la virtualisation d'un processeur complet. Ainsi, sur la figure B.1b, le logiciel de virtualisation, qui est exécuté par dessus l'interface correspondant au jeu d'instructions, déploie une interface correspondant à un jeu d'instruction. Cela signifie que le logiciel exécuté par dessus cette interace, appelé *logiciel invité* –par opposition au *système hôte*–, ne pourra intéragir avec le logiciel de virtualisation qu'à travers le jeu d'instructions.

L'intérêt de déployer une couche de virtualisation est d'altérer les propriétés de certains composants matériels et logiciels pour répondre à des besoins spécifiques. Selon ces besoins, la nature du composant et les propriétés voulues, on pourra par exemple effectuer les opérations suivantes :

Partitionnement. On segmentera le composant en sous-composants disjoints, que l'on allouera de manière exclusive à différentes entités. Le cas typique de partitionnement est la mémoire principale, que l'on alloue par plages distinctes à des partitions logiques.

Multiplexage. On assemblera les activités de plusieurs entités logicielles dans une seule activité que l'on propagera sur le composant visé.

Ordonnancement. On segmentera temporellement l'accès au composant en question.

Contrôle d'accès. On effectuera systématiquement des opérations spécifiées à l'avance en intéragissant avec le composant.

Distribution. On répartira sur plusieurs composants du système hôte l'activité que le logiciel invité verra comme sur un seul composant. Cela peut être le cas lorsque l'on cherche à répartir entre plusieurs serveurs le traffic sur un réseau virtuel.

Réplication. L'activité initiée par le logiciel invité sur ce qu'il voit comme un composant sera répliquée sur plusieurs composants du système hôte. Cela peut être le cas lorsque l'on souhaite répliquer les écritures sur un disque dur.

Nous disposons donc d'un cadre dans lequel il est possible de situer l'intégralité des pratiques courantes dans le domaine de la virtualisation. En ce qui nous concerne, nous cherchons à réutiliser des systèmes avioniques composés d'un système d'exploitation et d'un ensemble de partitions ARINC 653. Les logiciels de virtualisation qui permettent la réutilisation de tels composants sont appelés *hyperviseurs*, ou *gestionnaires de machines virtuelles*[49]. Ils déploient un environnement d'exécution simulant un processeur complet, aussi appelé *machine virtuelle*. D'après le cadre exposé ci-dessus, les techniques présentes dans l'état de l'art introduisent des niveaux d'abstractions décrits par un jeu d'instructions ou une ABI. Elles correspondent respectivement à des concepts dits de *virtualisation totale*, où le logiciel invité peut être exécuté de manière inchangée, et de

49. Virtual Machine Monitors

para-virtualisation, où le logiciel invité doit être conforme à une ABI prédéfinie, ce qui requiert un effort de portage.

Nous développons dans les deux sous-sections suivantes une classification des hyperviseurs présents dans l'état de l'art, puis les différents concepts qu'il est nécessaire d'implémenter dans un hyperviseur pour déployer des machines virtuelles.

B.2 Classification des hyperviseurs existants

On classe habituellement les hyperviseurs selon deux critères :

- L'environnement dans lequel l'hyperviseur est déployé. Cela peut être directement à même le processeur, ou dans un processus géré par un système d'exploitation hôte. On parle respectivement d'hyperviseurs de type 1 et de type 2, ou encore de machine virtuelle systèmes ou processus [89].
- La technique de virtualisation employée. On rencontre trois grandes techniques, qui sont la virtualisation totale par traduction de binaires, la virtualisation totale assistée par le matériel, et la para-virtualisation.

B.2.1 Classification par environnement d'exécution

Les hyperviseurs de type 1, appelés aussi hyperviseurs bas niveau (bare metal hypervisors), sont déployés directement sur le processeur. Ils sont en général légers en termes d'empreinte mémoire et de taille de code, ce qui en fait de bons candidats pour les systèmes temps réels critiques, où l'accumulation de couches logicielles n'est pas souhaitable. La contrepartie est un nombre de fonctionnalités réduit. Ces hyperviseurs ont en général pour seule finalité d'assurer un partitionnement spatial entre les machines virtuelles qui sont déployées.

Les hyperviseurs de type 2 sont exécutés en tant que processus d'un système d'exploitation hôte. Il sont répandus dans le domaine de l'informatique dématérialisée et de la bureautique, car ils offrent de nombreuses fonctionnalités dans la gestion des machines virtuelles. Il leur suffit par exemple d'instancier un nouveau processus pour créer à la volée une machine virtuelle. Comme ce sont des processus intégrés dans un système d'exploitation, ils héritent des propriétés que ces derniers leurs confèrent. Ainsi, ils répondent bien au besoin d'ordonnancement de machines virtuelles, car ils utilisent l'ordonnanceur du système d'exploitation. De la même manière ils permettent d'effectuer des opérations de multiplexage, de distribution ou encore de réplication en utilisant les éventuels services fournis par le système d'exploitation hôte.

Dans la communauté des systèmes embarqués, les hyperviseurs de type 1 sont les plus répandus. Nous abordons maintenant un second critère de classification, qui porte sur la technique de virtualisation mise en place.

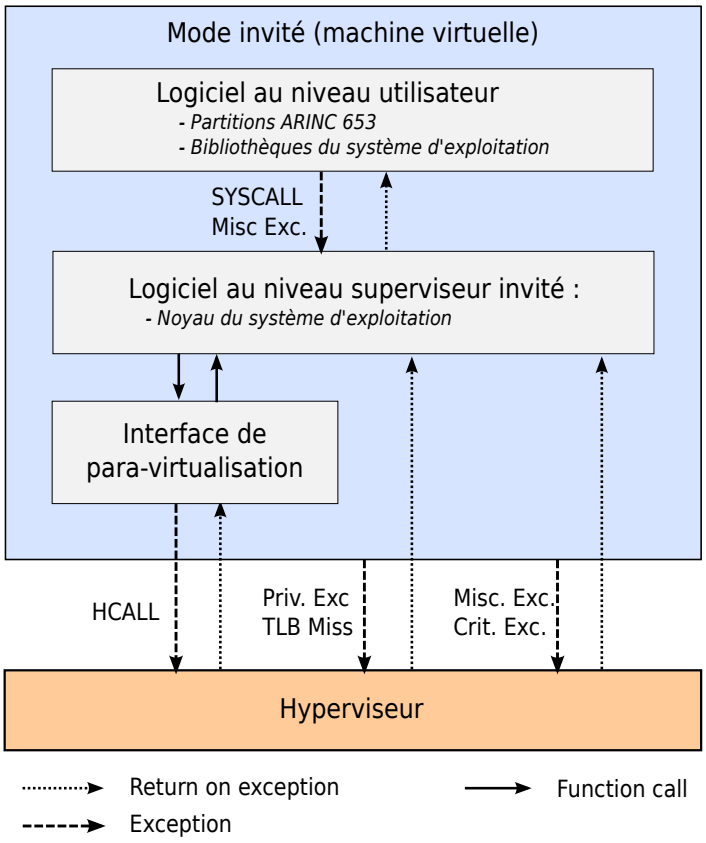


FIGURE B.2 – Illustration des techniques de virtualisation.

B.2.2 Classification par technique de virtualisation

On peut également classer les hyperviseurs selon trois familles de techniques de virtualisation, qui ont été développées pour répondre à des problématiques de portabilité et de performances [95] :

- la virtualisation totale par traduction de binaire
- la virtualisation totale asisstée par le matériel
- la para-virtualisation

Virtualisation totale par traduction de binaires

La virtualisation totale par traduction de binaires consiste à interpréter à la volée le binaire du logiciel invité pour générer des instructions équivalentes qui seront exécutées par le processeur. Cette technique permet d'émuler des jeux d'instructions et des ressources matérielles différentes de celles de la machine hôte. Elle est donc pertinente pour réutiliser des composants logiciels qui sont adhérents à des composants matériels qui ne sont plus disponibles. Les solutions les plus répandues actuellement sont QEMU [50],

50. www.qemu.org/

ou encore Simics [51]. Tous sont des hyperviseurs de type 2, et ont besoin d'un système d'exploitation hôte pour leur fonctionnement. Historiquement, cette technique souffrait de deux défauts : les performances du logiciel invité étaient fortement dégradées, et il était très difficile d'ajouter un nouveau jeu d'instructions. Ces deux points ont été améliorés, en particulier dans QEMU [19]. L'amélioration consiste à regrouper le code traduit par segments d'exécution qui sont activés à la volée, et stockés dans une table dédiée assure leur validité. La généricité est obtenue par l'introduction d'un front-end, appelé TCG [52], associé à un langage de description de l'activité interne d'un cœur. La traduction de binaire est alors obtenue par un traducteur (back-end) vers le jeu d'instructions de la machine hôte. In fine, on a pu observer des performances quasi-équivalentes à celles du système natif [98]. Toutefois, à notre connaissance, la technique de virtualisation par traduction de binaire n'a jamais été appliquée dans l'embarqué temps réel critique.

Virtualisation totale assistée par le matériel

Le principe général de la virtualisation totale assistée par le matériel (hardware assisted) consiste à exécuter le logiciel invité directement sur le processeur et à l'interrompre quand il tente d'effectuer une opération sensible, par exemple une reprogrammation de la MMU. Contrairement à la technique précédente de traduction de binaires, il s'agit ici d'un mode de virtualisation passif : le logiciel de virtualisation ne s'exécute que lorsque c'est nécessaire, pour simuler l'effet d'une opération sensible intentée par le logiciel invité. Il est significativement plus léger en terme d'instrusivité dans le logiciel invité. En contrpartie, seul du logiciel invité respectant le jeu d'instructions du système hôte peut être déployé.

Cette technique a été formalisée en 1974 par Popek et Goldberg [69] par les trois propriétés suivantes :

- L'efficacité : les instructions non sensibles sont exécutées sans intervention du logiciel de virtualisation.
- Le contrôle des ressources : Le logiciel invité n'a aucun moyen d'accéder à une ressource matérielle qui ne lui a pas été allouée.
- L'équivalence : La séquence d'instructions exécutée par le logiciel invité est invariante selon la présence ou non d'un logiciel de virtualisation, à l'exception du temps d'exécution.

Popek et Goldberg ont montré qu'il suffisait que l'ensemble des instructions qualifiées de "sensibles", c'est-à-dire influant sur la configuration ou le comportement du processeur, déclenchent une exception pour qu'il existe un logiciel de virtualisation qui vérifie les trois définitions ci-dessus. Ces travaux ont donné lieu à diverses extensions portant sur la nature des instructions sensiibles, par exemple sur des processeurs ARM récents [68]. Ils sont à la base de la virtualisation assistée par le matériel.

Historiquement, le logiciel de virtualisation se substituait au système d'exploitation et déployait les machines virtuelles dans le mode de privilège utilisateur. Le système

51. http://www.windriver.com/simics/
52. Tiny Code Generator

d'exploitation et les applications étaient exécutées au même niveau de privilège. Cela augmentait la complexité de l'hyperviseur, qui devait assurer l'intégrité du système d'exploitation invité par rapport aux applications, intégrité qui est habituellement garantie par une séparation de niveaux de privilèges. D'autre part cela impliquait une intervention systématique de l'hyperviseur pour des évènements qui ne concernaient que le système d'exploitation invité, par exemple les appels systèmes déclenchés par les applications.

A partir des années 2000, les processeurs ont disposé de mécanismes limitant au strict nécessaire l'intervention de l'hyperviseur dans l'exécution du logiciel invité. Ces mécanismes, illustrés sur la figure B.2, sont les suivants :

- L'introduction d'un niveau de privilège intermédiaire entre le niveau superviseur et le niveau utilisateur, dit *superviseur invité* (guest supervisor). Le système d'exploitation invité a vocation à être exécuté à ce niveau de privilège, qui est reconnu par la MMU. Cela permet au système d'exploitation invité de garantir son intégrité par rapport aux applications.
- Certains évènements déclenchent des exceptions qui sont directement dirigées vers le système d'exploitation invité, et basculent vers le privilège superviseur invité. Il n'y a donc pas d'intervention de l'hyperviseur.

La plupart des hyperviseurs existants –quel que soit leur type– relèvent, au moins partiellement, de cette technique de virtualisation, car elle offre de meilleures performances que la traduction de binaires [98] tout en permettant d'effectuer de la virtualisation totale, donc en ayant un effort de portage nul. Toutefois, malgré les améliorations du matériel, l'intervention de l'hyperviseur lors d'une opération sensible peut être lourde. Il est en effet nécessaire d'inférer les paramètres de l'opération telle qu'elle a été intentée par le logiciel invité. Cela demande parfois d'interpréter son code binaire, ce qui est coûteux en taille de code et en temps de calcul. L'usage de la technique de para-virtualisation, que nous présentons ci-après, vise à éviter cet effet.

Para-virtualisation

La para-virtualisation consiste à mettre en place une interface d'appels systèmes que le logiciel invité peut appeler pour solliciter certains services de l'hyperviseur. D'après la définition de Smith et Nait, cela signifie que l'hyperviseur déploie une machine virtuelle décrite, au moins partiellement, par une ABI. La para-virtualisation vise à se substituer à des mécanismes de virtualisation totale dont l'implémentation est lourde en termes de taille de code et de performances pour réaliser des opérations sensibles sur le processeur. Les mécanismes de virtualisation totale ont en effet assez peu d'informations à leur disposition quand à la nature de l'opération que le logiciel invité a intentée. Ils doivent la reconstituer à partir de l'état du processeur et du code binaire du logiciel invité.

L'interface de para-virtualisation définit un ensemble d'appels systèmes, habituellement appelés *hypercalls*, qui réalisent :

- Certaines opérations sensibles pour le compte du logiciel invité.

- Des services de plus haut niveau que l'on peut rencontrer dans un système d'exploitation, par exemple l'allocation dynamique de ressources (mémoire et périphériques).
- Des remontées d'informations et des opérations de maintenance sur la machine virtuelle.

Le passage d'arguments lors d'un appel système suit le même principe que pour un système d'exploitation, à savoir une convention d'utilisation des registres. La majorité des hyperviseurs qui ont une interface de para-virtualisation fournissent une bibliothèque de compatibilité qui effectue les appels systèmes et les rend disponibles par une API, ce qui simplifie leur intégration.

La mise en place d'une interface de para-virtualisation suppose que le logiciel invité ait connaissance de cette interface. Un effort de portage est donc nécessaire, ce qui va à l'encontre de la problématique de réutilisation de composants. Toutefois, l'usage de la para-virtualisation est répandu dans les systèmes embarqués car il simplifie la complexité du code embarqué d'un point de vue global, et améliore ses performances [61]. Cet usage a aussi des raisons historiques : la para-virtualisation est arrivée plus tôt que la virtualisation assistée par le matériel, et a longtemps été la seule alternative à la traduction de binaires.

Dans la communauté embarquée, la para-virtualisation et la virtualisation assistée par le matériel sont donc les deux techniques de virtualisation les plus utilisées. La plupart des hyperviseurs existants mettent en œuvre ces deux techniques. Nous présentons dans la prochaine sous-section les différents concepts que l'on doit mettre en œuvre lors de la conception d'un hyperviseur, et les techniques habituelles pour les implémenter.

B.3 Concepts implémentés dans un hyperviseur

Un hyperviseur est un composant logiciel capable de déployer sur un processeur une ou plusieurs machines virtuelles. Chacune d'entre elle permet à un système d'exploitation de s'exécuter et d'ordonnancer un ensemble d'application en ayant l'illusion des faits suivants :

- Il dispose de manière exclcusive de ressources matérielles et le temps d'exécution que l'hyperviseur a alloué à la machine virtuelle. Il ne voit pas les autres ressources matérielles.
- Il contrôle de manière autonome, parmi les ressources qui sont à sa disposition, l'allocation des ressources matérielles et de temps d'exécution aux applications qu'il héberge.

Une machine virtuelle doit donc offrir un environnement dans lequel du logiciel invité verra son exécution comme ininterrompue, et aura l'illusion d'être déployé sur un processeur réel. Elle doit également offrir des ressources qui permettront à un système d'exploitation invité de gérer ses applications de manière autonome. Pour cela, un hyperviseur doit virtualiser trois types de ressources :

- l'espace adressable afin de permettre au système d'exploitation de définir des règles de pagination qui lui sont propres.

TABLE B.1 – Classification des hyperviseurs existants

Hyperviseur	Technique de virtualisation			Type		Caractéristiques		
	Traduction de binaire	Virt. assistée par le matériel	Para-virtualisation	Type 1	Type 2	Open source	Hyperviseur embarqué	Hyperviseur grand public
XtratuM		X	X	X			X	
Topaz		X	X	X		X	X	X
VirtualBox		X			X			X
VMware		X			X			X
Xen		X	X	X			X	
KVM		X	X		X			X
Integrity		X	X	X			X	
PikeOS		X	X	X			X	
Enea Hypervisor		X	X	X			X	
X-Hyp			X			X	X	
QEMU	X				X	X		X
Simics	X			X				X
SeL4		X	X	X			X	
OKL4 Microvisor		X	X	X				
LynxSecure		X	X	X			X	

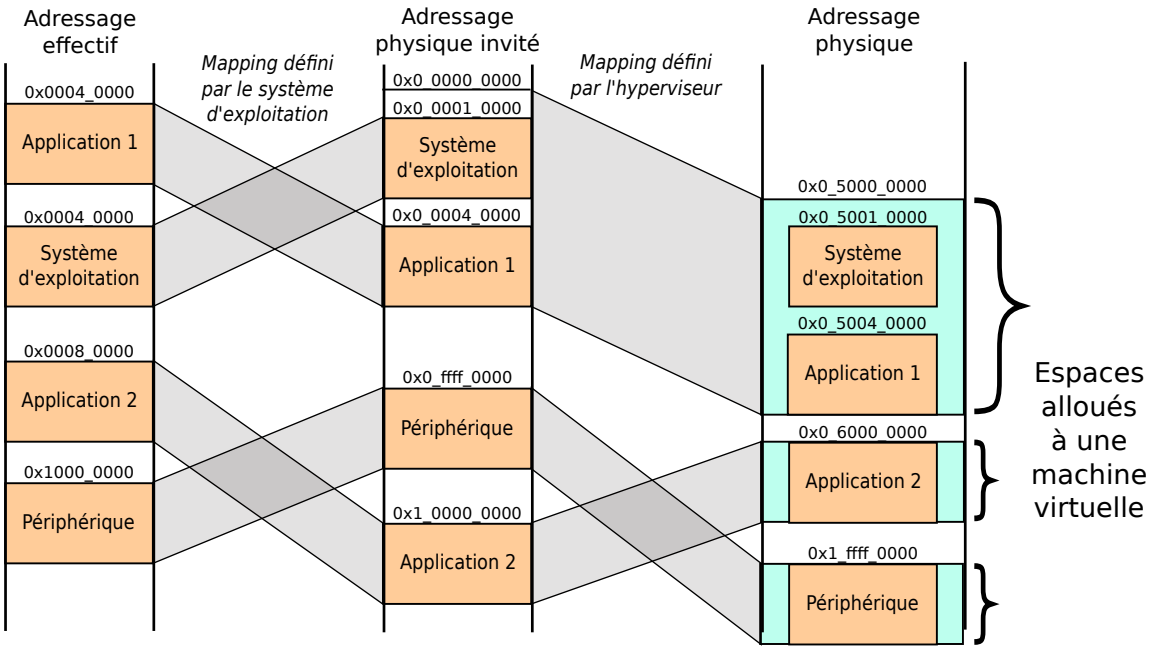


FIGURE B.3 – Virtualisation de l'espace adressable physique par un hyperviseur

- Les périphériques afin de gérer le partage desdits périphériques entre machines virtuelles.
- Les interruptions, dans le même but que les périphériques.

D'une manière générale, virtualiser une ressource comme l'espace adressable, un périphérique ou une interruption consiste à donner au logiciel invité l'illusion de contrôler cette ressource, et d'en avoir l'exclusivité. Or cette ressource doit être sous contrôle de l'hyperviseur, dans un but d'isolation et/ou de partage de cette ressource, ce qui échappe au logiciel invité. Nous détaillons ci-après les particularités liées à la virtualisation de ces trois ressources.

B.3.1 Virtualisation de l'espace adressable

La virtualisation de l'espace adressable a pour principal objectif de séparer les plages mémoires qui sont utilisées par les machines virtuelles, tout en laissant au logiciel invité l'illusion de contrôler l'utilisation de ces plages mémoire en définissant ses règles de pagination.

Sur un système non virtualisé, le système d'exploitation est en charge de définir la correspondance entre l'espace d'adressage effectif, que voit le logiciel, et l'espace d'adressage physique, que voit le processeur. A cet effet, il programme la MMU avec les règles de pagination correspondantes. Les applications ne sont pas conscientes de ce lien et ne voient que l'espace effectif.

L'hyperviseur doit laisser le système d'exploitation libre de définir ses propres règles de pagination, car lui seul dispose des informations pour le faire. Dans le même temps, il doit assurer la maîtrise de l'empreinte mémoire de chaque machine virtuelle, par exemple pour assurer qu'elles sont disjointes.

Pour concilier ces deux points de vue, la solution classiquement adoptée consiste à introduire un espace d'adressage intermédiaire appelé *physique invité* (guest physical), qui est illustré sur la figure B.3. Le système d'exploitation invité est en charge de définir des règles de pagination entre l'espace d'adressage effectif et l'espace physique invité, qu'il croit être l'espace physique. L'hyperviseur définit des règles de pagination entre l'espace physique invité et l'espace physique. Il peut ainsi allouer des plages de mémoire à chaque machine virtuelle sans connaître à l'avance les règles de pagination qui seront définies par le système d'exploitation.

Dans les solutions de virtualisation existantes, l'espace d'adressage physique invité est mis en place de manière logicielle. Ainsi, l'hyperviseur stocke dans son espace mémoire privé une réplique de la table des pages, appelée *table des pages virtuelles*[53] [96]. La table des pages virtuelle est maintenue par l'hyperviseur lorsque le logiciel invité cherche à modifier les règles de pagination sur la machine virtuelle. L'hyperviseur les compose ensuite avec des règles de pagination qui résultent de la définition des machines virtuelles.

Il est attendu que dans la prochaine génération de processeurs, la table des pages virtuelle pourra être maintenue par le matériel. Par exemple ARM-v8 [16] a introduit un modèle de MMU à plusieurs niveaux, le premier étant programmable par le système d'exploitation invité, le second étant restreint à l'hyperviseur.

B.3.2 Virtualisation des périphériques

La virtualisation des périphériques répond à différents objectifs. Il s'agit selon les cas :

- D'allouer un périphérique de manière exclusive à une machine virtuelle.
- D'interdire l'accès à un périphérique à une machine virtuelle.
- De multiplexer les accès concurrents de plusieurs machines virtuelles à un périphérique.

Sur un processeur, l'accès à un périphérique peut se faire par l'espace d'adressage (c'est le cas sur les processeurs ARM et PowerPC) et/ou par des ports (c'est le cas sur les processeurs Intel et AMD), pour lesquels l'écriture et la lecture sont des opérations réservées. Dans le cas des accès à travers l'espace d'adressage, l'hyperviseur peut autoriser ou interdire un accès en configurant la MMU. Dans la majorité des cas, la machine virtuelle et l'hyperviseur échangent les données à destination des périphériques par des buffers internes, l'hyperviseur se chargeant d'effectuer les opérations sur les périphériques. Ce concept a été notamment implémenté dans l'interface VirtIO [81], qui ets implantée dans les noyaux linux.

53. On y fait référence dans la littérature sous le terme de *shadow page table*

On rencontre sur les processeurs COTS des périphériques qui supportent le multiplexage de plusieurs flux de requêtes au niveau matériel. C'est par exemple le cas des bus PCIe, des contrôleurs DMA ou encore des contrôleurs de réseau, pour lesquels on dispose de plusieurs canaux d'entrées, puis associer le flux d'un cœur à un canal en particulier. Le périphérique effectue lui-même un ordonnancement sur les différents canaux.

Lorsque le logiciel invité cherche à programmer les périphériques, il est courant de mettre en place un moyen de protection. Les processeurs récents disposent de MMU dédiées aux accès aux périphériques, souvent appelées IOMMU [54] ou PAMU [55]. Ces unités matérielles, situées en amont des périphériques, filtrent les accès que ceux-ci reçoivent, et peuvent effectuer des traduction d'adresses, de la vérification de droits d'accès, et dans certains cas de l'adaptation de protocole.

Lorsque le matériel n'offre pas de support pour la virtualisation de périphériques, ce qui est souvent le cas pour les périphériques à bas débit comme les UART [56], il est possible d'effectuer le multiplexage de manière logicielle. Un cœur va alors héberger un serveur d'I/O, qui écoutera des ports de communication inter-tâches et génèrera un message assemblé à destination du périphérique concerné. Dans le sens inverse il découpera le message reçu et distribuera selon des séparateurs pré-définis les flux démultiplexés aux ports de communication inter-tâches.

D'une manière générale, plus l'abstraction qui implémente l'interface virtualisée du périphérique altère le comportement de l'abstraction native, plus la complexité du logiciel de virtualisation augmente et ses performances sont altérées. Ce phénomène a été résumé par Russell [80] comme suit :

> *"Le danger consiste à introduire une abstraction tellement éloignée de la réalité que les performances s'écroulent, qu'il y a plus de code de compatibilité que de code utile, et qu'il y a des critères de validité qui semblent arbitraires"*

54. Inputs/Outputs Memory Management Units
55. Peripheral Access Management Units
56. Universal Asynchronous Receiver/Transmitter

Bibliographie

[1] RTCA/DO-178B - EUROCAE/ED-12 : Software Considerations in Airborne Systems and Equipment Certification, Décembre 1992. (Cité page 12.)

[2] SAE/ARP-4754 : Certification Considerations for Highly-Integrated or Complex Aircraft Systems, Novembre 1996. (Cité pages 11 et 13.)

[3] SAE/ARP-4761 : Guidelines And Methods For Conducting The Safety Assessment Process On Civil Airborne Systems And Equipment, Décembre 1996. (Cité pages 11, 12 et 13.)

[4] RTCA/DO-254 - EUROCAE/ED-80 :Software Considerations in Airborne Systems and Equipment Certification, Avril 2000. (Cité pages 12 et 17.)

[5] RTCA/DO-297 : Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations, Novembre 2005. (Cité pages 10, 13, 22 et 38.)

[6] CS-25, Certification Specifications for Large Aeroplanes, Amendment 6, Juillet 2009. (Cité pages 11 et 12.)

[7] ARINC–653 : Avionics Application Software Standard Interface Part 1 – Required Services, Novembre 2010. (Cité pages 21, 22 et 23.)

[8] Development Assurance of Airborne Electronic Hardware - EASA Certification Memorandum, Août 2011. (Cité pages 12, 17 et 18.)

[9] RTCA/DO-178C - EUROCAE/ED-12 : Software Considerations in Airborne Systems and Equipment Certification, Mai 2012. (Cité page 12.)

[10] B. Akesson, K. Goossens, and M. Ringhofer. Predator : A predictable SDRAM memory controller. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2007 5th IEEE/ACM/IFIP International Conference on*, pages 251–256, Septembre 2007. (Cité page 47.)

[11] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens. Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration. In *14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA '08.*, pages 3–14, Août 2008. (Cité page 47.)

[12] Muhammad Ashraful Alam and S Mahapatra. A comprehensive model of PMOS NBTI degradation. *Microelectronics Reliability*, 45(1) :71–81, 2005. (Cité page 19.)

[13] Patrick Jocelyn Andrianiaina, Daniel Simon, Alexandre Seuret, Jean-Michel Crayssac, and Jean-Claude Laperche. Weakening Real-time Constraints for Embedded Control Systems. Research Report RR-7831, INRIA, Décembre 2011. (Cité page 15.)

[14] P.J. Andrianiaina, A. Seuret, and D. Simon. Robust control under weakened real-time constraints. In *Decision and Control and European Control Conference (CDC-ECC), 2011 50th IEEE Conference on*, pages 2016–2021, Décembre 2011. (Cité page 15.)

[15] ARM. *CoreLink CCI-400 Cache Coherent Interconnect Technical Reference Manual, Rev G*. (Cité page 41.)

[16] ARM. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*. ARM, a.c edition, Juillet 2014. (Cité pages 69 et 179.)

[17] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA : an open toolbox for adaptive WCET analysis. In *Proceedings of the 8th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems*, SEUS'10, pages 35–46, Berlin, Heidelberg, 2010. Springer-Verlag. (Cité pages 36 et 154.)

[18] Sanjoy Baruah and Alan Burns. Sustainable Scheduling Analysis. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, RTSS '06, pages 159–168, Washington, DC, USA, 2006. IEEE Computer Society. (Cité page 22.)

[19] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, Avril 2005. (Cité page 174.)

[20] Herbert S Bennett. Will future measurement needs of the semiconductor industry be met ? *Journal of Research of the National Institute of Standards and Technology*, 112(1) :25–38, Janvier 2007. (Cité page 17.)

[21] G. Bernat, A. Colin, and S.M. Petters. WCET analysis of probabilistic hard real-time systems. In *23rd IEEE Real-Time Systems Symposium (RTSS)*, pages 279–288, 2002. (Cité pages 15 et 37.)

[22] Benoit Berthe. A380 ATA 42 certification. In *Proceedings of the CISEC IMA day*, 2007. (Cité page 25.)

[23] Philippe Bieth and Vincent Brindejonc. COTS-AEH - Use of complex COTS (Components-Off-The-Shelf) in Airborne Electronic Hardware - Failure Mode and Mitigation. Technical Report EASA 2012/04, European Aviation Safety Agency, Novembre 2013. (Cité pages 14 et 18.)

[24] Jingyi Bin, Sylvain Girbal, Gracia Pérez Daniel, Arnaud Grasset, and Alain Merigot. Studying co-running avionic real-time applications on multi-core COTS architectures. In *7th European Congress On Embedded Real Time Software And Systems (ERTS)*, 2014. (Cité page 42.)

[25] M. Bohr. The new era of scaling in an SoC world. In *Solid-State Circuits Conference - Digest of Technical Papers, 2009. ISSCC 2009. IEEE International*, pages 23–28, Février 2009. (Cité page 19.)

[26] Frédéric Boniol, Hugues Cassé, Eric Noulard, and Claire Pagetti. Deterministic Execution Model on COTS Hardware. In *Proceedings of the 25th International Conference on Architecture of Computing Systems*, ARCS'12, pages 98–110, Berlin, Heidelberg, 2012. Springer-Verlag. (Cité page 48.)

[27] Shekhar Borkar, Tanay Karnik, and Vivek De. Design and Reliability Challenges in Nanometer Technologies. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, pages 75–75, New York, NY, USA, 2004. ACM. (Cité page 19.)

[28] Roman Bourgade. *Analyse du temps d'exécution pire-cas de tâches temps-réel exécutées sur une architecture multi-coeurs*. PhD thesis, Institut de Recherche en Informatique de Toulouse (IRIT), Octobre 2012. (Cité page 42.)

[29] Francisco J. Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery Berger, Jaume Abella, Franck Wartel, Michael Houston, Luca Santinelli, Leonidas Kosmidis, Code Lo, and Dorin Maxim. PROARTIS : Probabilistically Analyzable Real-Time Systems. *ACM Trans. Embed. Comput. Syst.*, 12(2s) :94 :1–94 :26, May 2013. (Cité page 37.)

[30] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quinones, and F.J. Cazorla. Measurement-Based Probabilistic Timing Analysis for Multi-path Programs. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 91–101, Juillet 2012. (Cité pages 15 et 37.)

[31] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4) :35 :1–35 :44, October 2011. (Cité page 22.)

[32] Julien Delange and Laurent Lec. POK, an ARINC653-compliant operating system released under the BSD license. In *Proceedings of the 13th Real-Time Linux Workshop*, 2011. (Cité page 152.)

[33] Sanjay Deshpande. Flow Control Mechanisms for Avoidance of Retries and/or Deadlocks in an Interconnect, Décembre 2010. (Cité pages 41 et 157.)

[34] Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and Wolfgang Puffitsch. Predictable Flight Management System Implementation on a Multicore Processor. In *Embedded Real Time Software and Systems*, ERTS '14, 2014. (Cité page 48.)

[35] Clifton A Ericson. *Hazard analysis techniques for system safety*. John Wiley & Sons, 2005. (Cité page 11.)

[36] Frédéric Fauberteau. *Sûreté temporelle pour les systèmes temps réel multiprocesseurs*. These, Université Paris-Est, December 2011. (Cité page 15.)

[37] Frédéric Faubladier and David Rambaud. Safety Implications of the use of system-on-chip (SoC) on commercial-of-the-shelf (COTS) devices in airborne critical applications. Technical Report Research Project EASA.2008/1, European Aviation Safety Agency (EASA), 2008. (Cité pages 14 et 18.)

[38] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Proceedings of the First International Workshop on Embedded Software*, EMSOFT '01, pages 469–485, London, UK, UK, 2001. Springer-Verlag. (Cité page 37.)

[39] H. Forsberg and K. Karlsson. COTS CPU Selection Guidelines for Safety-Critical Applications. In *25th Digital Avionics Systems Conference, 2006 IEEE/AIAA*, pages 1–12. IEEE, 2006. (Cité pages 17 et 92.)

[40] Freescale. *e5500 Core Reference Manual*. Freescale Semiconductor, Freescale Semiconductor Technical Information Center, CH370 1300 N. Alma School Road Chandler, Arizona 85224 +1-800-521-6274 or +1-480-768-2130, rev g edition, Octobre 2010. (Cité page 121.)

[41] Freescale. *EREF 2.0 : A Programmer's Reference Manual for Freescale Power Architecture Processors*. Freescale Semiconductor, Freescale Semiconductor Technical Information Center, CH370 1300 N. Alma School Road Chandler, Arizona 85224 +1-800-521-6274 or +1-480-768-2130, rev 1 edition, Juin 2014. (Cité pages xvi, 102, 127 et 128.)

[42] Freescale Semiconductor. *Freescale's Embedded Hypervisor for QORIQ P4 Series Communication Platforms*, Octobre 2008. White paper. (Cité page 132.)

[43] Freescale Semiconductor. *e5500 Core Reference Manual, Rev G*, Novembre 2010. (Cité pages xv, xvi, 27, 97, 101, 102, 104, 162, 163 et 164.)

[44] Freescale Semiconductor. *EREF 2.0 : A Programmer's Reference Manual for Freescale Power Architecture Processors, Rev 0*, Septembre 2011. (Cité page 95.)

[45] Rudolf Fuchsen. How To Address Certification For Multi-core Based Ima Platforms : Current Status And Potential Solutions. In *29th IEEE/AIAA Digital Avionics Systems Conference : Improving Our Environment through Green Avionics and ATM Solutions (DASC '10)*, pages 5. E.31–5. E.311, Octobre 2010. (Cité page 43.)

[46] Laurent George. État de l'art sur la robustesse temporelle des systèmes temps-réel monoprocesseur. *Journal Européen des Systèmes Automatisés*, 42/9 :1135–1160, 2008. (Cité page 15.)

[47] Bob Green, Joseph Marotta, Brian Petre, Kirk Lillestolen, Spencer Richard, Nikhil Gupta, Daniel O'Leary, Jason Dan Lee, John Strasburger, Arnold Nordsieck, Bob Manners, and Rabi Mahapatra. Handbook For The Selection And Evaluation Of Microprocessors For Airborne Systems. Technical report, Federal Aviation Administration - U.S. Department of Transportation, Air Traffic Organization NextGen & Operations Planning Office of Research and Technology Development Washington, DC 20591, Décembre 2011. (Cité page 17.)

[48] Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET Analysis of Multicore Architectures using UPPAAL. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, pages 103–113. Österreichische Computer Gesellschaft, Juillet 2010. (Cité page 55.)

[49] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. CoMPSoC : A Template for Composable and Predictable Multi-processor System on Chips. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1) :2 :1–2 :24, January 2009. (Cité page 46.)

[50] Damien Hardy. *Analyse pire cas pour processeur multi-cœurs disposant de caches partagés*. These, Université Rennes 1, December 2010. (Cité page 44.)

[51] GD Hutcheson. The economic implications of Moore's law. In *High Dielectric Constant Materials*, pages 1–30. Springer, 2005. (Cité page 17.)

[52] Jean-Bernard Itier. A380 integrated modular avionics. In *Proceedings of the ARTIST2 meeting on integrated modular avionics*, volume 1, pages 72–75, 2007. (Cité pages 16 et 25.)

[53] Xavier Jean, Marc Gatti, Guy Berthon, and Marc Fumey. MULCORs, The Use of MULticore proCessORS in Airborne Systems. Technical Report EASA 2011.OP.30, European Aviation Safety Agency, Décembre 2012. (Cité page 92.)

[54] V. Jegu, B. Triquet, F. Aspro, C. Pagetti, and F. Boniol. Method and device for loading and executing instructions with deterministic cycles in a multicore avionics system having a bus, the access time of which is unpredictable, Avril 2012. (Cité pages xv, 48 et 57.)

[55] Daniel Kästner, Reinhard Wilhelm, Reinhold Heckmann, Marc Schlickling, Markus Pister, Marek Jersak, Kai Richter, and Christian Ferdinand. Timing Validation of Automotive Software. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation*, volume 17 of *Communications in Computer and Information Science*, pages 93–107. Springer Berlin Heidelberg, 2009. (Cité page 36.)

[56] L.M. Kinnan. Use of multicore processors in avionics systems and its potential impact on implementation and certification. In *Digital Avionics Systems Conference, 2009. DASC '09. IEEE/AIAA 28th*, pages 1.E.4–1 –1.E.4–6, Octobre 2009. (Cité page 43.)

[57] R. Kirner and P. Puschner. Obstacles in Worst-Case Execution Time Analysis. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 333–339, Mai 2008. (Cité page 36.)

[58] Yan Li, V. Suhendra, Yun Liang, T. Mitra, and A. Roychoudhury. Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 57–67, Décembre 2009. (Cité page 44.)

[59] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '08, pages 137–146, New York, NY, USA, 2008. ACM. (Cité page 46.)

[60] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 12–21, 1999. (Cité page 37.)

[61] M. Masmano, I. Ripoll, A. Crespo, and J. J. Metge. XtratuM : a Hypervisor for Safety Critical Embedded Systems. In *11th Real-Time Linux Workshop*, 2009. (Cité page 176.)

[62] William P McNally. Will Commercial Specifications Meet Our Future Air Power Needs. Technical report, DTIC Document, 1997. (Cité page 17.)

[63] Thomas Moscibroda and Onur Mutlu. Memory performance attacks : denial of memory service in multi-core systems. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, pages 18 :1–18 :18, Berkeley, CA, USA, 2007. USENIX Association. (Cité page 44.)

[64] E. Normand. Single-event effects in avionics. *Nuclear Science, IEEE Transactions on*, 43(2) :461–474, Avril 1996. (Cité page 19.)

[65] J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pages 109–118, Juillet 2014. (Cité page 42.)

[66] Jan Nowotsch and Michael Paulitsch. Leveraging Multi-core Computing Architectures in Avionics. *European Dependable Computing Conference*, 0 :132–143, 2012. (Cité pages 43 et 55.)

[67] M. Paolieri, E. Quinones, F.J. Cazorla, and M. Valero. An Analyzable Memory Controller for Hard Real-Time CMPs. *Embedded Systems Letters, IEEE*, 1(4) :86–90, dec. 2009. (Cité page 47.)

[68] Niels Penneman, Danielius Kudinskas, Alasdair Rawsthorne, Bjorn De Sutter, and Koen De Bosschere. Formal virtualization requirements for the ARM architecture. *J. Syst. Archit.*, 59(3) :144–154, March 2013. (Cité pages 66 et 174.)

[69] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7) :412–421, July 1974. (Cité pages xv, 5, 6, 60, 66, 67, 68, 69, 72, 90, 148 et 174.)

[70] Vladimir Popović and Branko Vasić. Review of hazard analysis methods and their basic characteristics. *FME Transactions*, 36(4) :181–187, 2008. (Cité page 11.)

[71] D. Powell. Failure mode assumptions and assumption coverage. In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, pages 386–395, Juillet 1992. (Cité page 13.)

[72] Power.org. *Power ISAVersion 2.06*, Juillet 2010. (Cité pages 102 et 127.)

[73] Power.org. *Power ISAVersion 2.07*, Mai 2013. (Cité page 69.)

[74] P.J. Prisaznuk. ARINC 653 role in Integrated Modular Avionics (IMA). In *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*, pages 1.E.5–1–1.E.5–10, Octobre 2008. (Cité page 22.)

[75] Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J. Cazorla. On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 8(4) :34 :1–34 :25, January 2012. (Cité page 42.)

[76] D. Regis, G. Hubert, F. Bayle, and M. Gatti. IC components reliability concerns for avionics end-users. In *Digital Avionics Systems Conference (DASC), 2013 IEEE/AIAA 32nd*, pages 2C2–1–2C2–9, Octobre 2013. (Cité page 19.)

[77] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing Predictability of Cache Replacement Policies. *Real-Time Syst.*, 37(2) :99–122, November 2007. (Cité page 127.)

[78] Anthony Roger and Vincent Brindejonc. Avoidance of dysfunctional behaviour of complex COTS used in an aeronautical context. In *Lambda-Mu*, 2014. (Cité page 41.)

[79] John Rushby. Partitioning in Avionics Architectures : Requirements, Mechanisms, and Assurance. Technical report, Computer Science Laboratory,SRI International, Menlo Park, 94025 USA, Mars 1999. (Cité pages 14 et 23.)

[80] Rusty Russell. Ambition, Hubris and Virtual I/O. Blog post, Mai 2007. http://ozlabs.org/ rusty/index.cgi/tech/2007-05-21.html. (Cité page 180.)

[81] Rusty Russell. Virtio : Towards a De-facto Standard for Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.*, 42(5) :95–103, July 2008. (Cité page 179.)

[82] C.E. Salloum, M. Elshuber, O. Hoftberger, H. Isakovic, and A. Wasicek. The ACROSS MPSoC – A New Generation of Multi-core Processors Designed for Safety-Critical Embedded Systems. In *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 105–113, Septembre 2012. (Cité page 46.)

[83] Martin Schoeberl and Peter Puschner. Is Chip-Multiprocessing the End of Real-Time Scheduling ? In *Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Dublin, Ireland, Juillet 2009. OCG. (Cité page 46.)

[84] Freescale Semiconductors. Freescale Collaborates with Avionics Manufacturers to Facilitate Their Certification of Systems Using Multicore Processors. Press Release, Septembre 2011. (Cité pages 17 et 92.)

[85] Freescale Semiconductors. Freescale Product Longevity Program Device List, Juin 2014. (Cité pages xvii, 17, 94, 162, 163 et 164.)

[86] Lui Sha, Marco Caccamo, Renato Mancuso, Jung-Eun Kim, Man-Ki Yoon, Rodolfo Pellizzoni, Heechul Yun, Russel Kegley, Dennis Perlman, Greg Arundale, et al. Single Core Equivalent Virtual Machines for Hard Real-ÂTime Computing on Multicore Processors. Technical report, Illinois University, Novembre 2014. (Cité page 46.)

[87] H. Shah, A. Raabe, and A. Knoll. Priority division : A high-speed shared-memory bus arbitration with bounded latency. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1 –4, Mars 2011. (Cité page 47.)

[88] Hardik Shah, Andreas Raabe, and Alois Knoll. Challenges of wcet analysis in cots multi-core due to different levels of abstraction. In *Workshop on High-performance and Real-time Embedded Systems (HiRES 2013)*, 2013. (Cité page 43.)

[89] James E. Smith and Ravi Nair. The Architecture of Virtual Machines. *Computer*, 38 :32–38, Mai 2005. (Cité pages xvi, 170 et 172.)

[90] Jim Smith and Ravi Nair. *Virtual Machines : Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. (Cité pages 70 et 73.)

[91] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. The impact of technology scaling on lifetime reliability. In *2004 International Conference on Dependable Systems and Networks*, pages 177–186, Juin 2004. (Cité page 19.)

[92] G.M. Swift, F.F. Fannanesh, S.M. Guertin, F. Irom, and D.G. Millward. Single-event upset in the PowerPC750 microprocessor. *IEEE Transactions on Nuclear Science*, 48(6) :1822–1827, Décembre 2001. (Cité page 19.)

[93] Texas-Instruments. *TMS320C6678 - Multicore Fixed and Floating-Point Digital Signal Processor*. Texas Instruments, Février 2012. (Cité page 41.)

[94] Theo Ungerer, Francisco Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quinones, Mike Gerdes, Marco Paolieri, Julian Wolf, Hugues Casse, Sascha Uhrig, Irakli Guliashvili, Michael Houston, Floria Kluge, Stefan Metzlaff, and Jorg Mische. Merasa : Multicore Execution of Hard Real-Time Applications Supporting Analyzability. *IEEE Micro*, 30 :66–75, 2010. (Cité page 46.)

[95] VmWare. Understanding Full Virtualization, Paravirtualization, and Hardware Assist. White paper, Octobre 2007. (Cité page 173.)

[96] VmWare. Virtualization : architectural considerations and other evaluation criteria. White paper, Octobre 2007. (Cité page 179.)

[97] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Fifth International Conference on Quality Software. (QSIC 2005).*, pages 295 – 303, september 2005. (Cité page 37.)

[98] Joshua White and Adam Pilbeam. A Survey of Virtualization Technologies With Performance Testing. *CoRR*, abs/1010.3233, 2010. (Cité pages 174 et 175.)

[99] Matthew M. Wilding, David S. Hardin, and David A. Greve. Invariant Performance : A Statement of Task Isolation Useful for Embedded Application Integration. In *Proceedings of the conference on Dependable Computing for Critical Applications*, DCCA '99, pages 287–, Washington, DC, USA, 1999. IEEE Computer Society. (Cité page 23.)

[100] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution-Time problem overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3) :36 :1–36 :53, May 2008. (Cité pages 34, 36 et 53.)

[101] Zheng Pei Wu, Y. Krish, and R. Pellizzoni. Worst Case Analysis of DRAM Latency in Multi-requestor Systems. In *IEEE 34th Real-Time Systems Symposium (RTSS)*, pages 372–383, Décembre 2013. (Cité page 44.)

[102] Yina Zhang. Evaluation of Methods for Dynamic Time Analysis for CC-Systems AB. Technical Report, Mälardalen University, Août 2005. (Cité page 37.)

Maîtrise de la couche hyperviseur sur les processeurs multi-cœurs COTS dans un contexte avionique

Xavier JEAN

RESUMÉ : Nous nous intéressons dans cette thèse à la maîtrise de processeurs multi-cœurs COTS dans le but de les rendre utilisables dans des équipements avioniques, qui ont des exigences temps-réel dures. L'objectif est de permettre l'application de méthodes connues d'évaluation de pire temps d'exécution (WCET) sur un ensemble de tâches représentatif d'applications avioniques.

Au cours de leur exécution, les tâches exécutées sur différents cœurs vont accéder simultanément à des ressources matérielles qui sont partagées entre les cœurs, en particulier la mémoire principale. Cela pourra entraîner des mises en attente de certains accès que l'on qualifie d'interférences. Ces interférences peuvent avoir un impact élevé sur le temps d'exécution du logiciel embarqué. Sur un processeur COTS, qui est acheté dans le commerce et vise un marché plus large que l'avionque, cet impact n'est pas borné.

Nous cherchons à garantir l'absence d'interférences grâce à des moyens logiciels, dans la mesure où les processeurs COTS ne proposent pas de mécanismes adéquats au niveau matériel. Nous cherchons à étendre des concepts de logiciel déterministe de telle sorte à les rendre compatibles avec un objectif de réutilisation de logiciel existant. A cet effet, nous introduisons la notion de logiciel de contrôle, qui est un élément fonctionnellement neutre, répliqué sur tous les cœurs, et qui contrôle les dates des accès des cœurs aux ressources communes de telle sorte à offrir une isolation temporelle entre ces accès. Nous étudions dans cette thèse le problème de faisabilité d'un logiciel de contrôle sur un processeur COTS, et de son efficacité vis à vis d'applications avioniques.

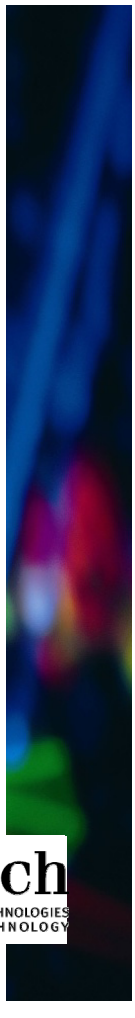
MOTS-CLEFS : Processeur multi-cœurs, logiciel de contrôle, virtualisation, hyperviseur, Avionique Modulaire Intégrée

ABSTRACT : We focus in this thesis on issues related to COTS multi-core processors mastering, especially regarding hard real-time constraints, in order to enable their usage in future avionics equipment. We aim at applying existing Worst Case Execution Time (WCET) evaluation methods on a set of tasks similar to those we can find in avionics software.

At runtime, tasks executed among different cores are likely to access hardware resources at the same time, e.g. the main memory. It may lead to additional delays due to hardware contention, called “interferences”. Interferences slow down embedded software within ranges that may be important. Additionnally, no bound has been established for their impact on WCET when using COTS processors, that target larger markets than avionics.

We try to provide guarantees that all interferences are eliminated through software, as COTS processors do not provide adequate mechanisms at hardware level. We extend deterministic software concepts that have been developed in the state of the art, in order to make them compliant with the use of legacy software. We introduce the concept of "control software", which is functionnaly neutral, is replicated among all cores, and performs active control of core’s accesses to shared resources, so that concurrent accesses are temporally isolated. We formalize and study in this thesis the problem of control software feasibility on COTS processors, and questions of efficiency with regard to legacy avionics software.

KEY-WORDS : Multi-core processors, control software, virtualization, hypervisor, Integrated Modular Avionics



TELECOM ParisTech

EDITE

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY