UNIVERSITÉ DE MONTRÉAL

PROFILAGE, CARACTÉRISATION ET PARTITIONNEMENT FONCTIONNEL DANS UNE PLATE-FORME DE CONCEPTION DE SYSTÈMES EMBARQUÉS

LAURENT MOSS
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION DU DIPLÔME DE
PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
JUIN 2010

© Laurent Moss, 2010.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

PROFILAGE, CARACTÉRISATION ET PARTITIONNEMENT FONCTIONNEL DANS UNE PLATE-FORME DE CONCEPTION DE SYSTÈMES EMBARQUÉS

présentée par : MOSS, Laurent
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

Mme. NICOLESCU, Gabriela, Doct., présidente.
M. BOIS, Guy, Ph.D., membre et directeur de recherche.
M. ABOULHAMID, El Mostapha, Ph.D., membre et codirecteur de recherche.
M. BOLAND, Jean-François, Ph.D., membre.
M. PÉTROT, Frédéric, Ph.D., membre.

REMERCIEMENTS

Je tiens à remercier chaleureusement mon directeur de recherche, le professeur Guy Bois, dont les judicieux conseils, le constant appui et la grande disponibilité ont été indispensables à la réalisation de ce projet de recherche. Je remercie également mon co-directeur de recherche, le professeur El Mostapha Aboulhamid, dont les sages conseils ont alimenté ma réflexion tout au long de ce projet de recherche. Je remercie aussi les professeurs Gabriela Nicolescu et Yvon Savaria pour leurs pertinents conseils lors de mon examen général de synthèse. Je remercie les professeurs Jean-François Boland et Frédéric Pétrot, de même que les autres membres du jury, d'avoir pris le temps de réviser ma thèse.

Je remercie mes collègues Luc Filion, Maxime de Nanclas, Marc-André Cantin et Sébastien Fontaine pour leur précieuse collaboration dans la rédaction d'articles pour les conférences DVCON, DATE et RSP. Pour leur collaboration et leur soutien, je remercie aussi mes autres collègues du laboratoire de recherche en codesign CIRCUS et de Space Codesign, notamment Jérôme Chevalier, Benoit Pilote, Sylvain Goyette, Ahmed Faiz, Cédric Migliorini, Fatoumata-Lamaranah Bah, Sébastien Le Beux, Hubert Guérard, Mathieu McKinnon et Michel Rogers-Vallée.

Je remercie Réjean Lepage pour la qualité du support technique qu'il m'a offert dans l'utilisation des équipements du Groupe de Recherche en Microélectronique et Microsystèmes de l'École Polytechnique de Montréal. Je remercie Jeanne Daunais pour son soutien administratif, notamment en lien avec mes participations à diverses conférences.

Je remercie le Conseil de recherches en sciences naturelles et en génie, le Regroupement Stratégique en Microsystèmes du Québec et l'École Polytechnique de Montréal pour leur appui financier qui m'a permis de me consacrer entièrement à la réalisation de ce projet de recherche.

Finalement, je remercie mes parents, Philippe et Louise, pour leur appui indéfectible dans tous mes projets et plus particulièrement dans mes études universitaires et ce projet de recherche.

RÉSUMÉ

La complexité architecturale des systèmes embarqués augmente constamment et ceux-ci comprennent maintenant plusieurs processeurs, bus, périphériques et accélérateurs matériels. Les méthodologies présentement utilisées par l'industrie pour la conception des systèmes embarqués n'arrivent pas à suivre cette évolution. Des méthodologies de niveau système ont été proposées pour hausser le niveau d'abstraction de la conception des systèmes embarqués. Une telle méthodologie comporte une plate-forme virtuelle qui permet d'allouer des composants, d'y assigner la fonctionnalité de l'application et de simuler l'architecture résultante à un niveau transactionnel. Une méthodologie de niveau système peut accélérer la conception des systèmes embarqués en partant d'une spécification exécutable, en explorant automatiquement l'espace de conception et en synthétisant une architecture optimisée pour l'application.

Cependant, les méthodologies de niveau système existantes ont plusieurs lacunes. Elles supposent typiquement que l'application est modélisée avec un modèle de calcul restrictif et n'automatisent pas la synthèse des modules de l'application vers des blocs matériels. Elles n'intègrent pas un profilage non-intrusif de l'application ou d'une architecture qui l'implémente. Leurs méthodes d'estimation n'automatisent pas la caractérisation de l'application ou de la plate-forme. Ces méthodologies considèrent séparément les problèmes de l'allocation des processeurs, de l'assignation des tâches aux processeurs et du choix d'une topologie de communication.

Nous présentons une méthodologie de niveau système pour la conception, l'exploration architecturale et la synthèse des systèmes embarqués basée sur la technologie Space Codesign™ et sa plate-forme virtuelle SPACE. Cette méthodologie répond aux problématiques soulevées car elle combine un modèle de calcul plus expressif, une méthode de synthèse matérielle automatisée des modules d'une spécification SystemC, un profilage non-intrusif au niveau système, une méthode de caractérisation automatisée de l'application et du système d'exploitation temps-réel (RTOS), ainsi que des heuristiques pour une formulation unifiée du problème d'exploration architecturale.

Ainsi, nous avons défini pour notre méthodologie un nouveau modèle de calcul, les réseaux de processus temps-réel (RTPN) qui sont une extension des réseaux de processus Kahn. Cette extension permet de modéliser des aspects importants du traitement temps-réel tels que la scrutation, les senseurs échantillonnés, les périphériques d'entrée/sortie et les contraintes temps-réel. La sémantique dénotationnelle des RTPN est définie afin de vérifier si le raffinement d'une spécification exécutable SystemC vers une implémentation concrète est fonctionnellement correct.

Notre méthodologie inclut une méthode automatisée de raffinement des communications transactionnelles vers des protocoles précis au cycle et à la broche près ainsi que la génération automatique de blocs matériels pour les modules de l'application. Cette méthode permet, conjointement avec une méthode de génération de code embarqué incluant un RTOS, de générer une implémentation de l'application qui peut être simulée avec la plate-forme virtuelle ou synthétisée et exécutée sur la cible finale. Une nouvelle méthode de profilage au niveau système est appliquée à une telle simulation, ce qui permet d'extraire non-intrusivement des données sur la performance des modules, des processeurs, du RTOS, des bus et des mémoires.

Une nouvelle méthode automatisée permet de caractériser, par des simulations profilées, à la fois la fonctionnalité de l'application et les implémentations logicielles et matérielles de ses modules. Les périphériques et les bus de la plate-forme virtuelle ont également été caractérisés et une nouvelle méthode automatise la caractérisation du RTOS. Ces caractérisations configurent un simulateur de performance à haut niveau qui estime précisément et très rapidement la performance d'un ensemble d'architectures pour l'application en tenant compte de la contention sur les bus et de l'ordonnancement des tâches sur les processeurs. Cette caractérisation mène également à une estimation précise et rapide des besoins en ressources matérielles.

Nous présentons une formulation du problème d'exploration architecturale qui combine le partitionnement logiciel/matériel, l'allocation des processeurs, l'assignation des tâches aux processeurs et le choix d'une topologie de communication. L'exploration architecturale évalue les architectures selon des critères de performance et de coût matériel à l'aide de notre méthode d'estimation. Nous présentons pour la première fois une analyse combinatoire de ce problème et sa formulation comme un problème de recherche locale, pour la résolution duquel nous définissons des heuristiques basées sur un recuit simulé adaptatif et sur une recherche tabou réactive. L'architecture retenue par l'exploration architecturale peut ensuite être synthétisée vers une implémentation finale dans un flot de conception RTL bien établi. La méthodologie dans son ensemble est appliquée à trois études de cas : un système de guidage d'un astromobile, un décodeur JPEG avec détection de peau et un encodeur/décodeur WiMAX.

ABSTRACT

Embedded systems have increasingly complex architectures and are now composed of several processors, buses, peripherals and hardware accelerators. Embedded system design methodologies currently used in industry are not keeping up with this evolution. System-level methodologies have been proposed in order to raise the level of abstraction of embedded system design. Such a methodology includes a virtual platform in which components can be allocated while application tasks can be bound to allocated components for a transaction-level simulation of the resulting architecture. A system-level methodology can accelerate embedded system design by using an executable specification, automating design space exploration and synthesizing an optimized architecture for the application.

However, current system-level methodologies have several shortcomings. They typically assume that the application is modeled with a restrictive model of computation and do not automate the synthesis of hardware blocks from application modules. They do not support a non-intrusive profiling of the application or of an architecture implementing the application. Their estimation methods do not automate the characterization of the application or of the platform. These methodologies consider processor allocation, task binding to processors and the choice of a communication topology to be separate problems instead of being different aspects of a single problem.

We present a system-level methodology for the design, architectural exploration and synthesis of embedded systems based on the Space Codesign™ technology and its SPACE virtual platform. This methodology tackles these problems by combining a more expressive model of computation, a method for the automated synthesis of hardware blocks from a SystemC specification's modules, a non-intrusive system-level profiling, a method for the automated characterization of the application and of the real-time operating system (RTOS), as well as heuristics for a unified formulation of the architectural exploration problem.

We have thus defined for our methodology a novel model of computation, called real-time process networks (RTPN), which is an extension of Kahn process networks. This extension enables the modeling of important aspects of real-time processing, such as polling, sensor sampling, input/output peripherals and real-time constraints. We define the denotational semantics of RTPNs, which is used to verify the functional correctness of a refinement from a SystemC executable specification to a concrete implementation.

Our methodology includes an automated refinement from transaction-level communications to cycle- and pin-accurate protocols as well as an automated generation of hardware blocks from application modules. This method enables, when combined with an embedded

software generation method which includes a RTOS, the generation of an implementation of the application, which can be simulated with the virtual platform or synthesized and executed on the final target. A novel profiling method is applied to such simulations in order to non-intrusively extract data on the performance of modules, processors, RTOS, buses and memories.

A novel automated method characterizes, through profiled simulations, both the application functionality and the software and hardware implementations of its modules. The devices and buses of the virtual platform have also been characterized and a novel method automates the characterization of the RTOS. These characterizations configure a high-level performance simulator for an accurate and very fast estimation of the performance of several candidate architectures for the application, taking into account bus contention and task scheduling on processors. This characterization also powers a fast and accurate estimation of required hardware resources.

A formulation of the architectural exploration problem is given such that it combines hardware/software partitioning, processor allocation, task binding on processors and the selection of a communication topology. This architectural exploration evaluates architectures for criteria of performance and hardware cost with our estimation method. We present for the first time a combinatorial analysis of this problem and its formulation as a local search problem, for which heuristics based on adaptative simulated annealing and reactive tabu search are defined. The architecture selected by the architectural exploration can then be synthesized towards a final implementation in a well-established RTL design flow. The methodology as a whole has been applied to three case studies: a rover guiding system, a JPEG decoder with skin detection and a WiMAX encoder/decoder.

TABLE DES MATIÈRES

REMERCIEMENTS iii

RÉSUMÉ iv

ABSTRACT vi

TABLE DES MATIÈRES viii

LISTE DES TABLEAUX xiv

LISTE DES FIGURES xvi

LISTE DES ANNEXES xix

LISTE DES SIGLES ET ABRÉVIATIONS xx

CHAPITRE 1 INTRODUCTION 1
- 1.1 Mise en contexte et problématique 1
- 1.2 Objectif 2
- 1.3 Contributions 3

CHAPITRE 2 REVUE DE LITTÉRATURE 6
- 2.1 Modèles de calcul parallèles 6
  - 2.1.1 Flots de données synchrones (SDF) 7
  - 2.1.2 Flots de données cyclo-statiques (CSDF) 7
  - 2.1.3 Flots de données booléens (BDF) 8
  - 2.1.4 Réseaux de processus Kahn (KPN) 8
    - 2.1.4.1 Sémantique dénontationnelle 9
    - 2.1.4.2 Limites et extensions 10
  - 2.1.5 Modèles temporisés de flots de données 11
  - 2.1.6 Réseaux de processus temps-réel 12
- 2.2 Synthèse matérielle au niveau système 13
  - 2.2.1 Intégration d'un bloc matériel existant 13
  - 2.2.2 Synthèse comportementale d'un bloc matériel 13
  - 2.2.3 Raffinement des communications 14

2.2.4 Raffinement des communications et synthèse matérielle 15
2.3 Profilage de performance 16
2.3.1 Profilage d'un code logiciel 16
2.3.1.1 Profilage par exécution sur FPGA 16
2.3.1.2 Profilage par simulation sur ISS 17
2.3.2 Profilage d'une plate-forme logicielle/matérielle 18
2.3.3 Profilage des changements de contexte 19
2.3.4 Profilage des arguments et des valeurs de retour 19
2.3.5 Portabilité du profilage 20
2.4 Estimation du temps d'exécution 20
2.4.1 Analyse statique 20
2.4.1.1 Estimation par ordonnancement statique 20
2.4.1.2 Estimation statique du WCET 22
2.4.2 Annotation temporelle d'une simulation TLM 24
2.4.2.1 Annotation temporelle par analyse statique 25
2.4.2.2 Annotation temporelle par analyse dynamique 27
2.4.3 Évaluation d'une trace dynamique 29
2.4.3.1 Évaluation par analyse statique 29
2.4.3.2 Évaluation par analyse dynamique 30
2.5 Algorithmes d'exploration architecturale 33
2.5.1 Partitionnement logiciel/matériel avec un processeur 34
2.5.2 Exploration architecturale avec plusieurs processeurs 35
2.5.2.1 Nombre fixe de processeurs 36
2.5.2.2 Exploration de l'allocation des processeurs 37
2.5.2.3 Allocation des processeurs et topologie de communications . . . 39
2.5.3 Algorithmes de recherche locale 40
2.6 Méthodologies intégrées 42
2.6.1 SoCDAL 42
2.6.2 PeaCE 43
2.6.3 Koski 45
2.6.4 Daedalus 46
2.6.5 SystemCoDesigner 48

CHAPITRE 3 PRÉSENTATION DE LA MÉTHODOLOGIE 50
3.1 Aperçu général 50
3.2 Présentation des éléments de la méthodologie 52

3.2.1 Modèle de calcul RTPN 52
3.2.2 Spécification exécutable parallèle 55
3.2.3 Plate-forme virtuelle SPACE 57
3.2.4 Synthèse d'architecture 58
3.2.5 Synthèse des modules 59
3.2.5.1 Synthèse logicielle 60
3.2.5.2 Synthèse des modules matériels 61
3.2.6 Synthèse du système 61
3.2.7 Synthèse de plate-forme et synthèse logique 63
3.2.8 Profilage au niveau système 63
3.2.9 Caractérisation et estimation 63
3.2.10 Exploration architecturale 64

CHAPITRE 4 RÉSEAUX DE PROCESSUS TEMPS-RÉEL 65
4.1 Définition des réseaux de processus temps-réel 65
4.1.1 Évènements et relation de précédence 66
4.1.2 Types de canaux et de réseaux de processus 67
4.2 Sémantique dénotationnelle des RTPN 69
4.2.1 Ordonnancements équivalents à un KPN classique 70
4.2.2 Classes d'ordonnancements équivalents 72
4.2.3 Représentation d'un RTPN comme un KPN paramétré 73
4.3 Applications aux systèmes embarqués 75
4.3.1 Mémoire partagée 75
4.3.2 Senseur échantillonné 75
4.3.3 Périphériques d'entrée et de sortie 76
4.3.4 Contraintes temps-réel 76
4.3.5 Modélisation d'une application SPACE par un RTPN 77

CHAPITRE 5 RAFFINEMENT DES COMMUNICATIONS ET SYNTHÈSE DU MATÉRIEL 79
5.1 Sérialisation des types de données 80
5.1.1 Sérialisation triviale 80
5.1.2 Sérialisation standardisée 80
5.2 Raffinement des communications 82
5.2.1 Spécification exécutable au niveau TLM 82
5.2.2 Raffinement du protocole de communications 84
5.2.3 Synthèse d'interface 85

5.3 Synthèse du matériel 85
5.3.1 Synthèse comportementale 85
5.3.2 Synthèse de la plate-forme 89

CHAPITRE 6 PROFILAGE AU NIVEAU SYSTÈME 90
6.1 Profilage non-intrusif de code logiciel 91
6.1.1 Fonctionnement général du profileur d'ISS 92
6.1.2 Implémentation du profileur d'ISS 93
6.1.2.1 Implémentation du profileur d'ISS pour le MicroBlaze 94
6.1.2.2 Implémentation générique avec GDB 95
6.1.3 Profilage du RTOS et de l'API logicielle de SPACE 97
6.1.3.1 Algorithmes du profileur d'ISS SPACE 97
6.1.3.2 Implémentation pour μCOS/II 99
6.1.4 Profilage exhaustif du code logiciel 100
6.2 Profilage au niveau système dans SPACE 100
6.2.1 Instrumentation de la plate-forme SPACE 101
6.2.2 Informations et métriques extraites du profilage 103
6.2.2.1 Exécution des modules et du RTOS 103
6.2.2.2 Communication des modules de bout en bout 104
6.2.2.3 Transferts sur le bus 105
6.2.2.4 Accès à la mémoire 106

CHAPITRE 7 CARACTÉRISATION ET ESTIMATION 108
7.1 Définition des métriques 108
7.2 Performance et validité fonctionnelle 109
7.2.1 Caractérisation de la fonctionnalité de l'application 110
7.2.1.1 Définition de la trace fonctionnelle 110
7.2.1.2 Extraction d'un CPG par profilage 112
7.2.2 Caractérisation du temps d'exécution des modules 115
7.2.2.1 Définition des fonctions de caractérisation des modules 116
7.2.2.2 Profilage d'une implémentation d'un module 117
7.2.2.3 Analyse des enregistrements de profilage 117
7.2.3 Caractérisation de la plate-forme 119
7.2.3.1 Caractérisation des périphériques 119
7.2.3.2 Caractérisation des adaptateurs de bus 120
7.2.3.3 Caractérisation des bus et des ponts 120
7.2.3.4 Caractérisation du RTOS et de l'API logicielle 121

7.2.4 Estimation par une simulation de performance 124
7.2.4.1 Fonctionnement général du simulateur de performance 125
7.2.4.2 Implémentation du simulateur de performance 126
7.2.4.3 Validité de l'estimation 127
7.3 Quantité de ressources matérielles 128
7.3.1 Caractérisation des modules matériels 129
7.3.2 Caractérisation des modules logiciels 130
7.3.3 Caractérisation de la plate-forme 131
7.3.4 Estimation des quantités des ressources matérielles 133

CHAPITRE 8 EXPLORATION ARCHITECTURALE 134
8.1 Définition de l'espace de recherche 134
8.2 Problèmes d'exploration architecturale 136
8.3 Fonctions objectives 139
8.4 Algorithmes d'exploration architecturale 141
8.4.1 Complexité de l'exploration architecturale 142
8.4.2 Algorithme de parcours en profondeur 142
8.4.3 Algorithme glouton 144
8.4.4 Recherche locale 146
8.4.4.1 Définition du voisinage 147
8.4.4.2 Marche aléatoire 148
8.4.4.3 Descente 148
8.4.4.4 Recuit simulé adaptatif 150
8.4.4.5 Recherche tabou 153
8.4.4.6 Recherche tabou réactive 155

CHAPITRE 9 RÉSULTATS ET DISCUSSION 159
9.1 Présentation des études de cas 159
9.1.1 Système de guidage d'un rover 159
9.1.2 Décodeur JPEG avec détection de la peau 160
9.1.3 Encodeur/décodeur WIMAX 161
9.2 Modéles RTPN 162
9.3 Synthèse matérielle et des communications 163
9.3.1 Synthèse des modules matériels 163
9.3.2 Simulation à différents niveaux 167
9.3.3 Synthèse sur FPGA 168
9.4 Profilage au niveau système 170

9.4.1 Architectures testées 170
9.4.2 Impact du profilage 171
9.5 Caractérisation et estimation 174
9.5.1 Estimation de la quantité de ressources matérielles 174
9.5.2 Estimation du temps d'exécution 177
9.6 Exploration architecturale 179
9.6.1 Espace de recherche et analyse combinatoire 183
9.6.2 Comparaisons des algorithmes de recherche locale 184
9.6.3 Caractéristiques des solutions obtenues 189

CHAPITRE 10 CONCLUSION 191
10.1 Synthèse des travaux 191
10.2 Limites et améliorations futures 192
10.2.1 Équivalence des ordonnancements 192
10.2.2 Consommation de puissance et d'énergie 193
10.2.3 Temps de caractérisation des modules logiciels 195
10.2.4 Généralisation de l'exploration architecturale 196
10.2.5 Performance du logiciel embarqué 197
10.2.6 Évaluation industrielle de la méthodologie 198

RÉFÉRENCES 200

ANNEXES 219

LISTE DES TABLEAUX

Tableau 3.1 Paramètres des fonctions de communication de SPACE 56
Tableau 3.2 Conversion des types de données SystemC pour un logiciel embarqué . 61
Tableau 4.1 Définition des canaux Kahn, POLL, N-BREG et N-REG 68
Tableau 4.2 Conditions à respecter pour qu'un canal se comporte comme un canal Kahn 71
Tableau 5.1 Conversion, pour différents types de données, de la valeur de 32 bits `0x0A0B0C0D` d'une représentation *big endian* vers une représentation *little endian* 81
Tableau 7.1 Définition des attributs des opérations de communication 113
Tableau 7.2 Attributs des enregistrements de communication de bout en bout utilisés pour l'extraction du CPG 113
Tableau 7.3 Fonctions de caractérisation du temps d'exécution des segments de programme d'un module m 116
Tableau 7.4 Paramètres de performance pour différents périphériques avec interface OPB 120
Tableau 7.5 Paramètres de performance du bus OPB et du pont OPB-OPB 122
Tableau 7.6 Paramètres de performance du RTOS μCOS II et de l'API logicielle . . 123
Tableau 7.7 Paramètres des composants RTL d'un FPGA Virtex ciblé par SPACE 132
Tableau 8.1 Définitions des métriques appliquées aux modules 145
Tableau 8.2 Définitions des attributs pour la recherche tabou 155
Tableau 9.1 Ressources matérielles utilisées par les modules du décodeur JPEG . . 164
Tableau 9.2 Ressources matérielles utilisées par les modules du rover 165
Tableau 9.3 Ressources matérielles utilisées par les modules du codec WiMAX . . . 166
Tableau 9.4 Simulation à différents niveaux du système de guidage du rover 167
Tableau 9.5 Simulation à différents niveaux du décodeur JPEG 167
Tableau 9.6 Simulation à différents niveaux de l'encodeur/décodeur WiMAX 168
Tableau 9.7 Ressources matérielles et latence pour l'architecture de la figure 9.5 . . 170
Tableau 9.8 Simulation avec et sans profilage des architectures du rover 172
Tableau 9.9 Simulation avec et sans profilage des architectures du décodeur JPEG 172
Tableau 9.10 Simulation avec et sans profilage des architectures du codec WiMAX . 172
Tableau 9.11 WCT nécessaire à l'estimation et à la mesure par simulation du temps d'exécution des architectures du système de guidage du rover 180

Tableau 9.12 WCT nécessaire à l'estimation et à la mesure par simulation du temps d'exécution des architectures du décodeur JPEG 180
Tableau 9.13 WCT nécessaire à l'estimation et à la mesure par simulation du temps d'exécution des architectures de l'encodeur/décodeur WiMAX 180
Tableau 9.14 FPGA Virtex ciblés par les tests d'exploration architecturale 182
Tableau 9.15 Plate-forme cibles et contraintes T_{MAX} pour le décodeur JPEG avec détection de peau 182
Tableau 9.16 Plate-forme cibles et contraintes T_{MAX} pour l'encodeur/décodeur WiMAX 182
Tableau 9.17 Disponibilité des bascules et LUTs et contraintes T_{MAX} pour le système de guidage du rover 183
Tableau C.1 Bornes inférieures de $E_2(n,d)$ pour $0 \leq n \leq 10$ et $0 \leq d \leq 4$ 244

LISTE DES FIGURES

Figure 2.1 Exemple de graphe orienté $G = (V, A)$ représentant un réseau de processus 7
Figure 3.1 Aperçu général de la méthodologie proposée 53
Figure 3.2 Évolution d'une application à différentes étapes de la méthodologie . . 54
Figure 3.3 Structure d'un logiciel embarqué synthétisé pour un processeur dans SPACE 60
Figure 4.1 Exemple de réseau de processus 66
Figure 4.2 Évènements des processus a et b selon une relation de précédence. . . . 69
Figure 4.3 Évènements des processus a et b selon un ordonnancement t. 69
Figure 4.4 Représentation d'un canal RTPN comme un processus KPN paramétré par des séquences de bits de lecture et d'écriture 74
Figure 4.5 KPN paramétré équivalent au RTPN de la figure 4.1 74
Figure 4.6 Producteur et consommateur communiquant via une mémoire partagée. 76
Figure 4.7 Modélisation d'un système embarqué par un RTPN. 78
Figure 4.8 Spécification exécutable parallèle d'un système embarqué avec SPACE. 78
Figure 4.9 Modélisation d'une écriture rendez-vous avec un canal d'acquittement. 78
Figure 5.1 Flot de sérialisation standardisée 82
Figure 5.2 Flot automatisé pour le raffinement des communications et la synthèse du matériel 83
Figure 5.3 Exemple d'un fil d'exécution d'un module comportemental 84
Figure 5.4 Un exemple de structure de données à transférer 86
Figure 5.5 Un exemple d'une sérialisation standardisée au niveau RTL 86
Figure 5.6 Un exemple d'une désérialisation standardisée au niveau RTL 87
Figure 6.1 Fonctionnement général du profileur d'ISS 92
Figure 6.2 Implémentation générique du profileur d'ISS avec client/serveur GDB . 95
Figure 6.3 Le profileur d'ISS SPACE surveille la mémoire et les registres pour détecter les communications et les changements de contexte 97
Figure 6.4 Architecture générale du co-profilage 101
Figure 6.5 Instrumentation de la plate-forme SPACE. Les lignes pointillées représentent les accès faits par le profilage 101
Figure 6.6 Fichier de métriques de performance produit par le profilage au niveau système pour le module Y2R de l'exemple du JPEG 104
Figure 6.7 Charge des processeurs pour l'exemple du JPEG 105

Figure 6.8 Diagramme de Gantt des opérations de calcul et de communication du quantificateur inverse dans l'exemple du JPEG 106
Figure 6.9 Diagramme de Gantt des transferts sur le bus et métriques sur la charge du bus pour l'exemple du JPEG 107
Figure 7.1 Méthode de caractérisation et d'estimation de la performance et du nombre de violations fonctionnelles 111
Figure 7.2 Opérations du module M_2 114
Figure 7.3 Opérations du module M_3 114
Figure 7.4 CPG des communications du périphérique P_1 et des modules M_2 et M_3 114
Figure 7.5 Méthode de caractérisation et d'estimation de la quantité de ressources matérielles 129
Figure 8.1 Comparaison de la croissance du nombre $E(n)$ d'architectures pour n modules avec celle de 2^n et $n!$ 143
Figure 8.2 Fonctionnement général de l'algorithme de parcours en profondeur . . 143
Figure 8.3 Fonctionnement général de l'algorithme glouton 146
Figure 8.4 Fonctionnement général de l'algorithme de descente 149
Figure 8.5 Fonctionnement général de l'algorithme de recuit simulé 151
Figure 8.6 Fonctionnement général de l'algorithme de recherche tabou 153
Figure 8.7 Fonctionnement général de l'algorithme de recherche tabou réactive . . 156
Figure 9.1 Schéma de la spécification exécutable du système de guidage du rover . 159
Figure 9.2 Schéma des modules et périphériques du décodeur JPEG 161
Figure 9.3 Schéma des modules de l'encodeur/décodeur WiMAX 162
Figure 9.4 Modèle RTPN du rover 163
Figure 9.5 L'architecture cible pour le système de guidage 169
Figure 9.6 Comparaison du nombre estimé et mesuré de bascules 176
Figure 9.7 Comparaison du nombre estimé et mesuré de LUTs 176
Figure 9.8 Comparaison du nombre estimé et mesuré de mémoires BRAMs 176
Figure 9.9 Comparaison du nombre estimé et mesuré de multiplieurs 177
Figure 9.10 Comparaison du temps pris par l'estimation, la synthèse logique et le placement 177
Figure 9.11 Comparaison du temps d'exécution estimé et simulé pour 54 architectures du système de guidage du rover 179
Figure 9.12 Comparaison du temps d'exécution estimé et simulé pour 31 architectures du décodeur JPEG avec détection de peau 179
Figure 9.13 Comparaison du temps d'exécution estimé et simulé pour 32 architectures de l'encodeur/décodeur WiMAX 180

Figure 9.14 Comparaison des valeurs objectives obtenues par les algorithmes de recherche locale pour la maximisation de la performance pour différentes configurations . . . 185
Figure 9.15 Comparaison du WCT (en secondes) pour les algorithmes de recherche locale pour la maximisation de la performance pour différentes configurations . . . 186
Figure 9.16 Comparaison des valeurs objectives obtenues par les algorithmes de recherche locale pour la minimisation de la quantité de ressources matérielles pour différentes configurations . . . 186
Figure 9.17 Comparaison du WCT (en secondes) pour les algorithmes de recherche locale pour la minimisation de la quantité de ressources matérielles pour différentes configurations . . . 187
Figure 9.18 Comparaison des valeurs objectives selon le nombre d'itérations pour les algorithmes de recherche locale pour la maximisation de la performance du décodeur JPEG sur différentes plates-formes cibles . . . 188
Figure 9.19 Comparaison des valeurs objectives selon le nombre d'itérations pour les algorithmes de recherche locale pour la minimisation de la quantité de ressources matérielles du décodeur JPEG sur différentes plates-formes cibles . . . 188
Figure 10.1 Un processus RTPN qui utilise une lecture non-bloquante comme une lecture bloquante . . . 194
Figure A.1 Un processus p avec un canal c d'entrée, un signal a d'arbitrage, un signal b d'ordonnancement, un signal r de requête et des signaux x,y et z de données . . . 221
Figure A.2 Modèle général d'un RTPN en tant que DEPN . . . 222
Figure A.3 Modèle d'un DEPN non déterministe avec $x_1 = y_1 = x_2 = y_2 = o$ . . . 223
Figure A.4 Modèle du RTPN après composition de l'arbitre avec les canaux . . . 223
Figure A.5 Modèle du RTPN après composition de l'ordonnanceur avec les processus 224
Figure C.1 Comparaison de la croissance de $E(n)$ avec 2^n, $B(n)$ et $n!$ . . . 236
Figure C.2 Définition combinatoire de f_1 dans Maple . . . 237

LISTE DES ANNEXES

Annexe A TRANSFORMATION D'UN RTPN EN DEPN 219
Annexe B PROCÉDURE DE CARACTÉRISATION DU RTOS ET DE L'API LOGICIELLE 226
Annexe C COMPLEXITÉ COMBINATOIRE DE L'EXPLORATION ARCHITECTURALE 231
Annexe D COMPLEXITÉ ALGORITHMIQUE DE L'EXPLORATION ARCHITECTURALE 245
Annexe E GÉNÉRATION ALÉATOIRE D'UNE ARCHITECTURE 247
Annexe F DÉTAILS DES ÉTUDES DE CAS 249

LISTE DES SIGLES ET ABRÉVIATIONS

ABI	Application Binary Interface
ALU	Arithmetic and Logic Unit
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASIC	Application-Specific Integrated Circuit
AST	Abstract Syntax Tree
BCA	Bus Cycle-Accurate
BDF	Boolean Data Flow
BRAM	Block Random Access Memory
CFG	Control Flow Graph
CPG	Communication Precedence Graph
CSDF	Cvclo-Static Data Flow
DAG	Directed Acyclic Graph
DCT	Discrete Cosine Transform
DEPN	Discrete Event Process Networks
DSP	Digital Signal Processing
EDK	Embedded Development Kit
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
FSL	Fast Simplex Link
GDB	GNU Debugger
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HDL	Hardware Description Language
ID	Identifier
IDCT	Inverse Discrete Cosine Transform
IEEE	Institute of Electrical and Electronics Engineers
ILP	Integer Linear Programming
IP	Intellectual Property
IR	Intermediate Representation
ISR	Interrupt Service Routine
ISS	Instruction Set Simulator
JPEG	Joint Photographic Experts Group

KPN	Kahn Process Network
LFSR	Linear Feedback Shift Register
LUT	Look-up Table
NP	Nondeterministic Polynomial
OPB	On-chip Peripheral Bus
PC	Program Counter
PLB	Processor Local Bus
PN	Process Network
PTAS	Polynomial-Time Approximation Scheme
RAM	Random Access Memory
RGB	Red Green Blue
RLE	Run-Length Encoding
RPN	Reactive Process Networks
RTL	Register Transfer Level
RTOS	Real-Time Operating System
RTPN	Real-Time Process Network
SDF	Synchronous Data Flow
SDRAM	Synchronous Dynamic Random Access Memory
SoC	System-on-Chip
SRAM	Static Random Access Memory
SW	Software
TIG	Task Interaction Graph
TLM	Transaction-Level Modeling
TPG	Task Precedence Graph
UART	Universal Asynchronous Receiver-Transmitter
UML	Unified Modeling Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
WCET	Worst-Case Execution Time
WCT	Wall Clock Time
WiMAX	Worldwide Interoperability for Microwave Access

CHAPITRE 1

INTRODUCTION

1.1 Mise en contexte et problématique

La complexité et les requis de performance des systèmes embarqués augmentent constamment. Pour répondre à ces requis, les systèmes embarqués sont maintenant des systèmes sur puce (SoC), qui peuvent intégrer plusieurs processeurs, bus, périphériques et accélérateurs matériels sur une seule et même puce, le tout étant relié par une topologie de communications. Cette évolution a notamment pour conséquence d'augmenter grandement l'importance du développement logiciel dans la conception des systèmes embarqués. Ainsi, il est estimé que le développement des logiciels embarqués compte pour près de 60% du coût de développement des systèmes embarqués utilisant une technologie 90nm (International Business Strategies Inc., 2004). De plus, la grande majorité des concepteurs de systèmes embarqués rapportent que leur équipe de travail consacre autant sinon plus d'argent, de temps et de ressources humaines aux aspects logiciels de leurs projets qu'à leurs aspects matériels (CMP Media LLC, 2006).

Les méthodologies au niveau transfert de registres (RTL) présentement utilisées par l'industrie pour la conception des systèmes embarqués n'arrivent pas à suivre cette évolution vers des architectures plus complexes et ayant un plus grand contenu logiciel. Ainsi, la grande majorité des projets de conception de systèmes embarqués respectent moins de 90% de leurs requis de fonctionnalité et de performance alors qu'environ 30% des projets respectent moins de 50% de ces requis (Krasner, 2003). De plus, la majorité des projets de conception de systèmes embarqués sont soit annulés, soit livrés avec un retard moyen de quatre mois (Krasner, 2003; CMP Media LLC, 2006). Étant donné que les systèmes embarqués sont fréquemment des produits avec une courte durée de vie économique, de tels retards sont lourds de conséquences et impliquent une réduction importante des ventes du produit à la fois par une réduction de la période de vente et par une part de marché réduite (Rowe, 2010). L'annonce d'un retard dans l'introduction d'un nouveau produit a donc un impact négatif significatif sur la valeur marchande de la compagnie qui développe ce produit (Hendricks et Singhal, 1997).

L'activité technique qui cause le plus souvent des retards dans les projets de systèmes embarqués est la conception et la spécification de l'architecture du système. Ainsi, dans près de la moitié des cas, cette activité est la cause des retards dans le développement du

logiciel embarqué (Volckmann *et al.*, 2008). La conception et la spécification de l'architecture comptent pour 25% du coût associé au développement du matériel d'un système utilisant une technologie 90nm et cette proportion suit une tendance à la hausse (International Business Strategies Inc., 2004). La réduction du temps de mise en marché des systèmes embarqués nécessite donc des méthodes qui permettent de faciliter la conception et la spécification de l'architecture du système et d'accélérer le développement du logiciel embarqué pour cette architecture.

1.2 Objectif

Afin de s'attaquer à ces problèmes, il a été proposé de hausser le niveau d'abstraction de la conception des systèmes embarqués à l'aide de méthodologies de niveau système (Bailey *et al.*, 2007). Ainsi, ces méthodologies visent une conception de haut niveau qui sépare la spécification de la fonctionnalité et de l'architecture, et qui sépare également la réalisation des calculs et des communications (Keutzer *et al.*, 2000). Grâce à une plate-forme virtuelle qui permet d'assigner la fonctionnalité de l'application à un modèle fonctionnel d'une architecture logicielle/matérielle, il est possible de simuler cette implémentation à haut niveau et d'avoir un aperçu de son comportement et de sa performance. Cela comporte deux avantages. Premièrement, il devient alors possible de débuter le développement du logiciel avant que l'architecture matérielle ait été fixée, ce qui réduit les risques de dépassement d'échéancier. Deuxièmement, il est possible de faire une exploration architecturale de l'application tôt dans le processus de développement, c'est-à-dire de tester plusieurs architectures possibles pour l'application et de sélectionner la meilleure à la lumière des résultats obtenus. Cela évite de développer une implémentation complète sur une architecture qui se révèle finalement inadéquate (Bailey *et al.*, 2007).

Il est possible de réaliser le plein potentiel des méthodologies de niveau système en définissant une méthodologie intégrée de conception, d'exploration architecturale et de synthèse des systèmes embarqués. Une telle méthodologie vise autant que possible à hausser le niveau d'abstraction de la conception tout en automatisant le raffinement vers une implémentation finale. Elle doit donc inclure les éléments suivants :

1. Un modèle de calcul formel qui permet de définir la fonctionnalité de l'application et de s'assurer que sa fonctionnalité est préservée lors de son raffinement vers une implémentation.

2. Une synthèse automatisée, pour une architecture donnée, des modules de l'application vers un logiciel embarqué et des accélérateurs matériels de même qu'un raffinement automatisé des communications entre modules vers des protocoles et des liens de communications concrets. Cela permet d'éviter la duplication des efforts de programmation entre le niveau

système et le niveau de l'implémentation, qui serait autrement une barrière significative à l'adoption des méthodologies de niveau système.

3. Un profilage au niveau système qui inclut les modules de l'application, le système d'exploitation temps-réel (RTOS), les processeurs et les bus. En effet, le manque de visibilité au niveau de l'ensemble du système a été identifié comme une cause majeure des retards dans les projets de systèmes embarqués (Krasner, 2003).

4. Une méthode d'estimation rapide du temps d'exécution et des ressources matérielles, qui permet d'évaluer rapidement différentes architectures sans avoir à réaliser à chaque fois une simulation précise au cycle près ou une synthèse logique. Une telle méthode demande généralement une caractérisation de l'application et celle-ci doit être automatisée afin d'accélérer le développement et de pouvoir rapidement re-caractériser l'application si elle évolue au cours du projet.

5. Une méthode d'exploration architecturale capable d'évaluer automatiquement un grand nombre d'architectures possibles et de sélectionner celle qui est la meilleure selon des critères de performance et de coût matériel.

L'objectif de cette thèse est de présenter, à l'aide de la technologie Space Codesign™ et de sa plate-forme virtuelle SPACE (Filion *et al.*, 2007; Bois *et al.*, 2010), la mise en oeuvre de ces différents éléments de même que leur intégration en une méthodologie de conception, d'exploration architecturale et de synthèse des systèmes embarqués.

1.3 Contributions

Cette thèse effectue les contributions suivantes à l'état de l'art :

1. Une nouvelle méthodologie intégrée de conception, d'exploration architecturale et de synthèse des systèmes embarqués qui est basée sur les éléments ci-dessous.

2. Un nouveau modèle de calcul, les réseaux de processus temps-réel (RTPN), qui est une extension des réseaux de processus Kahn. Cette extension permet de modéliser des aspects importants du traitement temps-réel tels que la scrutation, les senseurs échantillonnés, les périphériques d'entrée/sortie et les contraintes temps-réel. La sémantique dénotationnelle des RTPN est définie afin de vérifier si le raffinement d'une spécification exécutable SystemC vers une implémentation concrète est fonctionnellement correct. Ainsi, on montre sous quelles conditions deux ordonnancements d'un RTPN sont fonctionnellement équivalents et comment un tel ordonnancement peut être caractérisé par un ensemble de séquences de bits.

3. Une méthode qui automatise le raffinement des communications transactionnelles vers des protocoles précis au cycle et à la broche près ainsi que la génération des blocs matériels pour les modules de l'application. Ce raffinement inclut une sérialisation standardisée des

types de données abstraits vers une représentation précise au bit près qui est compatible avec les implémentations logicielles et matérielles des modules. Cette méthode permet, conjointement avec une méthode de génération de code logiciel embarqué incluant un RTOS, de générer, pour une architecture donnée, une implémentation de l'application qui peut être simulée avec la plate-forme virtuelle ou synthétisée et exécutée sur la cible finale.

4. Une nouvelle méthode de profilage non-intrusif d'un logiciel embarqué qui permet de profiler le temps à l'entrée et à la sortie de chaque appel de fonction de même que la valeur des arguments passés en paramètre et de la valeur de retour. Ce profilage du logiciel est appliqué à une nouvelle méthode de profilage au niveau système qui permet d'extraire non-intrusivement des données sur la performance des modules logiciels et matériels, des processeurs, du RTOS, des bus et des mémoires à partir d'une simulation du système.

5. Une nouvelle méthode automatisée qui permet de caractériser, par des simulations profilées, à la fois la fonctionnalité de l'application et les implémentations logicielles et matérielles de ses modules. Les périphériques et les bus de la plate-forme virtuelle ont également été caractérisés et une nouvelle méthode automatise la caractérisation du RTOS. Ces caractérisations configurent un simulateur de performance à haut niveau qui estime précisément et très rapidement la performance d'un ensemble d'architectures pour l'application en tenant compte de la contention sur les bus et de l'ordonnancement des tâches sur les processeurs. Cette caractérisation mène également à une estimation précise et rapide des besoins en ressources matérielles.

6. Une nouvelle formulation du problème d'exploration architecturale qui combine le partitionnement logiciel/matériel, l'allocation des processeurs, l'assignation des tâches aux processeurs et le choix d'une topologie de communication. L'exploration architecturale évalue les architectures selon des critères de performance et de coût matériel à l'aide de notre méthode d'estimation. Nous présentons pour la première fois une analyse combinatoire de ce problème et sa formulation comme un problème de recherche locale. Nous définissons également les heuristiques suivantes pour la résolution de ce problème : un parcours en profondeur, une marche aléatoire, un recuit simulé adaptatif et une recherche tabou réactive. L'architecture retenue par l'exploration architecturale peut ensuite être synthétisée vers une implémentation finale dans un flot de conception RTL bien établi.

La méthodologie dans son ensemble est appliquée à trois études de cas : un système de guidage d'un astromobile, un décodeur JPEG avec détection de peau et un encodeur/décodeur WiMAX. Les résultats montrent que notre méthode de synthèse matérielle s'applique à des modules aussi complexes qu'un décodeur Reed-Solomon ou Huffman. De plus, il est montré qu'il est possible d'implémenter un profilage non-intrusif au niveau système avec un impact minime sur la vitesse de simulation. La méthode d'estimation permet d'évaluer le

temps d'exécution d'un ensemble d'architectures avec une précision de 8% et une vitesse de 400 à 48000 fois plus rapide qu'une simulation complète. La méthode d'estimation permet également d'estimer la quantité de ressources matérielles avec une précision de 20% et avec une vitesse 200000 fois plus rapide que la synthèse logique avec placement. Finalement, la recherche tabou réactive obtient systématiquement de meilleurs résultats que la marche aléatoire et le recuit simulé adaptatif. Dans les cas où il a été possible de trouver une solution optimale avec un parcours en profondeur, la recherche tabou réactive a également pu trouver cette solution optimale.

Cette thèse est structurée comme suit. Le chapitre 2 présente une revue de littérature afin de situer nos contributions par rapport à l'état de l'art. Le chapitre 3 présente dans son ensemble notre méthodologie intégrée de conception, d'exploration architecturale et de synthèse des systèmes embarqués. Le chapitre 4 décrit les réseaux de processus temps-réel. Le chapitre 5 présente notre méthode automatisée de synthèse matérielle. Le chapitre 6 décrit notre méthode de profilage au niveau système. Le chapitre 7 présente la méthode de caractérisation et d'estimation du temps d'exécution et de la quantité de ressources matérielles. Le chapitre 8 présente la formulation du problème d'exploration architecturale et différents algorithmes pour le résoudre. Le chapitre 9 applique cette méthodologie aux études de cas et en présente les résultats. Finalement, le chapitre 10 effectue une synthèse des travaux et propose des améliorations futures à la méthodologie.

CHAPITRE 2

REVUE DE LITTÉRATURE

Ce chapitre passe en revue les travaux pertinents à notre méthodologie et à ses différentes étapes et décrit notre contribution par rapport à ces travaux. On passe d'abord en revue les travaux portant sur les modèles de calcul parallèles, puis sur la synthèse matérielle au niveau système. Le chapitre se poursuit par une revue du profilage de performance et des méthodes d'estimation du temps d'exécution. On passe ensuite en revue les algorithmes d'exploration architecturale. Finalement, notre méthodologie est comparée avec d'autres travaux présentant des méthodologies intégrées de conception, d'exploration architecturale et de synthèse des systèmes embarqués

2.1 Modèles de calcul parallèles

Les systèmes embarqués sont fréquemment spécifiés sous la forme d'un ensemble de tâches parallèles ou implémentés par une architecture comprenant plusieurs processeurs et accélérateurs matériels s'exécutant en parallèle. Il est cependant difficile de raisonner sur le parallélisme au niveau des langages de programmation impératifs couramment utilisés pour les spécifications exécutables et les logiciels embarqués (Lee, 2006). Les modèles de calcul parallèles sont des formalismes mathématiques qui permettent d'analyser les systèmes embarqués à un plus haut niveau d'abstraction et aident notamment au raffinement d'une spécification vers une implémentation conforme à celle-ci. Plus le pouvoir expressif d'un modèle de calcul est élevé, plus grand est l'ensemble des programmes parallèles qui peuvent être exprimés à l'aide de ce modèle de calcul, mais plus la capacité d'analyse statique de ces programmes parallèles devient réduite. Différents modèles de calcul représentent différents compromis entre le pouvoir expressif et la capacité d'analyse statique.

Les modèles de calculs les plus souvent utilisés pour l'exploration architecturale et la synthèse des systèmes embarqués sont les modèles de réseaux de processus. Tel qu'illustré à la figure 2.1, un réseau de processus est un graphe orienté $G = (V, A)$ tel que les noeuds V sont un ensemble de processus et les arcs A sont un ensemble de canaux FIFO. Dans la sémantique opérationnelle de ces modèles de calcul, les processus communiquent entre eux uniquement au moyen de ces canaux et non au moyen de variables partagées. Les principaux modèles de calcul de réseaux de processus sont présentés ici par ordre croissant de pouvoir expressif.

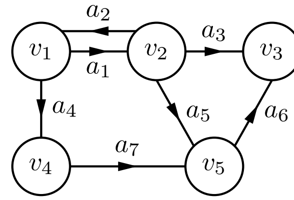

Figure 2.1 Exemple de graphe orienté $G = (V, A)$ représentant un réseau de processus

2.1.1 Flots de données synchrones (SDF)

Dans le modèle des flots de données synchrones (SDF : Synchronous Data Flow) (Lee et Messerschmitt, 1987), chaque processus répète toujours la même série d'opérations, qui représente une exécution du processus. Un processus lit d'abord un nombre fixe de jetons sur ses canaux d'entrée, effectue ensuite un traitement déterministe, puis écrit un nombre fixe de jetons sur ses canaux de sorties. De plus, toutes les lectures des processus sont bloquantes : un processus n'effectue pas son traitement tant que tous les jetons qu'il doit lire ne sont pas disponibles. Soit un canal qui va d'un processus a à un processus b et soit n_a le nombre de jetons écrits dans ce canal à chaque exécution de a et n_b le nombre de jetons lus dans ce canal à chaque exécution de b. On définit pour ce canal une équation d'équilibre $n_a e_a = n_b e_b$ tel que le nombre de jetons écrits dans le canal après e_a exécutions de a est égal au nombre de jetons lus dans le canal après e_b exécutions de b. En définissant une telle équation pour chacun des canaux d'un réseau SDF, on obtient un ensemble d'équations d'équilibre. Une solution à ce système d'équations est un vecteur de valeurs $\vec{e}$ qui indique combien de fois chacun des processus doit s'exécuter pour ramener le réseau SDF à son état initial, ce qui constitue une itération du réseau SDF. Si il existe un tel vecteur $\vec{e}$ non nul, alors ce vecteur peut être utilisé pour déterminer dans quel ordre doivent s'exécuter les processus et ainsi réaliser un ordonnancement statique du graphe SDF. Un cas particulier des réseaux SDF sont les SDF homogènes : à chacune de ses exécutions, chaque processus lit un seul jeton sur chacun de ses canaux d'entrée et écrit un seul jeton sur chacun de ses canaux de sortie.

2.1.2 Flots de données cyclo-statiques (CSDF)

Une extension des réseaux SDF est le modèle des flots de données cyclo-statiques (CSDF : Cyclo-Static Data Flow) (Bilsen *et al.*, 1996). Dans un réseau CSDF, chaque processus a une liste fixe, finie, ordonnée et circulaire de modes d'exécution SDF. Les modes d'exécution peuvent différer entre eux par le nombre de jetons lus ou écrits ou par le traitement effectué. Après chacune de ses exécutions, chaque processus passe au mode d'exécution suivant selon sa liste circulaire. De manière analogue au modèle SDF, il est possible de définir une itération

comme un ensemble d'exécutions qui ramène le réseau à son état initial. Il est également possible de trouver un ordonnancement statique pour un réseau CSDF, mais celui-ci sera généralement plus long que pour un réseau SDF étant donné qu'une itération du réseau doit ramener chaque processus à son mode d'exécution initial.

2.1.3 Flots de données booléens (BDF)

Les modèles SDF et CSDF ne sont pas Turing-complets (en d'autres termes, ils ne peuvent pas simuler une machine de Turing). C'est pourquoi il est possible de les ordonnancer statiquement et de déterminer statiquement si un tel réseau de processus peut s'exécuter avec une quantité bornée de mémoire. Cependant, leur pouvoir expressif est limité, car ils ne peuvent pas exprimer des flots de données conditionnels ou avec un nombre d'itérations dépendant des données. Le modèle des flots de données booléens (BDF : Boolean Data Flow) (Buck, 1993) étend ces modèles en ajoutant deux processus qui, selon une valeur booléenne de contrôle, peuvent respectivement chacun choisir de lire un jeton dans un de ses deux canaux d'entrées ou d'écrire un jeton dans un de ses deux canaux de sortie. Cette extension suffit à rendre le modèle BDF Turing-complet et à lui permettre de modéliser les flots de données conditionnels et les itérations dont le nombre dépend des données. Cependant, il n'est pas possible d'ordonnancer statiquement un réseau BDF dans le cas général et il faut alors avoir recours à un ordonnancement dynamique, où l'ordre d'exécution des processus est déterminé au fur et à mesure que ceux-ci s'exécutent sur le flot de données.

2.1.4 Réseaux de processus Kahn (KPN)

Les réseaux de processus Kahn (KPN : Kahn Process Networks) (Kahn, 1974) sont un modèle de calcul encore plus général. Dans un KPN, chaque processus est séquentiel, déterministe et peut effectuer dans un ordre arbitraire des traitements, des lectures sur des canaux d'entrée ou des écritures sur des canaux de sortie. Les canaux sont des FIFO sans perte de taille infinie avec une lecture bloquante et une écriture non-bloquante. Un processus ne peut pas vérifier le nombre de jetons présents dans un canal avant de décider d'effectuer ou non une lecture dans ce canal. La sémantique opérationnelle des KPN consiste essentiellement à ce que chaque processus s'exécute jusqu'à ce qu'une de ses lectures bloque en raison d'un manque de jetons sur un canal d'entrée, auquel cas le processus attend l'arrivée de nouveaux jetons avant de poursuivre son exécution.

2.1.4.1 Sémantique dénontationnelle

Le modèle des KPN est également doté d'une sémantique dénotationnelle, qui permet d'exprimer le comportement du réseau de processus dans son ensemble à partir du comportement de chacun de ses processus et des connexions entre ceux-ci. Soit une séquence arbitraire finie ou infinie de jetons $X = (x_1, x_2, \dots)$ et soit S l'ensemble de ces séquences. Un processus avec i canaux d'entrée et o canaux de sorties est défini comme une fonction $f : S^i \rightarrow S^o$ de ses séquences de jetons d'entrée vers des séquences de jetons de sortie. Ainsi, contrairement aux processus SDF et BDF, qui sont simplement des fonctions des jetons d'entrée vers des jetons de sortie, les processus KPN peuvent conserver un état et changer celui-ci selon les valeurs des jetons lus. En appliquant la fonction $f : S^i \rightarrow S^o$ propre à chaque processus du réseau, on obtient un ensemble d'équations portant sur les séquences de jetons transitant sur chaque canal du réseau. Ce système d'équations n'a pas nécessairement une solution directe étant donné qu'il peut y avoir des dépendances cycliques, tel que par exemple $Y = f(X)$ et $X = g(Y)$ qui implique $X = g(f(X))$. Une solution à un tel système d'équations est un point fixe. Si les fonctions de chaque processus sont continues au sens de Scott (Abramsky et Jung, 1994) (essentiellement, si leur comportement sur des séquences finies approxime leur comportement sur des séquences infinies), alors le théorème du point fixe de Kleene garantit l'existence d'au moins un point fixe qui est le plus petit point fixe (dont les séquences sont des préfixes de tous les autres points fixes) (Abramsky et Jung, 1994; Kahn, 1974).

La sémantique dénotationnelle des KPN définit le comportement d'un KPN comme son plus petit point fixe. L'existence et l'unicité de celui-ci implique qu'un KPN a un comportement déterministe qui dépend uniquement des fonctions calculées par ses processus et des connexions entre ceux-ci. De plus, le théorème du point fixe de Kleene est constructif : il indique que le plus petit point fixe peut être trouvé en initialisant toutes les séquences par la séquence vide et en appliquant itérativement les fonctions des différents processus. Cette procédure correspond à la sémantique opérationnelle des KPN : cela indique que celle-ci est conforme à la sémantique dénotationnelle des KPN et garantit elle aussi un comportement déterministe du réseau de processus (Lynch et Stark, 1989). En particulier, la fonctionnalité réalisée par un KPN ne dépend pas de l'ordonnancement des processus du KPN. Différentes implémentations d'un système spécifié sous la forme d'un KPN peuvent faire varier cet ordonnancement, par exemple en faisant varier le temps de calcul des processus ou le temps de communication des canaux sans affecter la fonctionnalité du système. Cette propriété formelle est très intéressante pour une méthodologie d'exploration architecturale, qui fera nécessairement varier ces paramètres en comparant par exemple l'exécution d'un KPN sur un seul processeur, sur plusieurs processeurs ou à l'aide d'accélérateurs matériels. C'est pourquoi plusieurs des méthodologies passées en revue à la section 2.6 utilisent les KPN ou un

sous-ensemble de ceux-ci tel que SDF ou CSDF.

2.1.4.2 Limites et extensions

Cependant, les KPN ont plusieurs restrictions qui nuisent à leur implémentation pratique. D'abord, ils supposent l'existence de canaux de taille infinie alors qu'une implémentation réelle aura seulement des canaux de taille finie à sa disposition. De plus, comme les KPN sont Turing-complets, il est généralement impossible de déterminer statiquement si l'exécution d'un KPN amènera un canal à contenir un nombre non borné de jetons. Une solution retenue par diverses interfaces de programmation (API) pour la spécification des systèmes embarqués se basant sur les KPN, tels que YAPI (de Kock *et al.*, 2000), TTL (van der Wolf *et al.*, 2004) et WFIFO (Huang *et al.*, 2007), est de borner la taille des canaux et de rendre l'écriture bloquante lorsqu'un canal est plein. Cependant, cela peut créer un interblocage artificiel si la taille des canaux est insuffisante. Il existe des algorithmes d'ordonnancement dynamique qui permettent de détecter ces interblocages artificiels et d'augmenter progressivement la taille des canaux afin de les résoudre (Parks, 1995; Geilen et Basten, 2003). Par contre, ces algorithmes peuvent avoir à augmenter infiniment la taille d'un canal et ils sont mieux adaptés à la simulation d'un KPN sur un ordinateur hôte que pour son implémentation dans un système sur puce, où la taille des canaux peut difficilement être modifiée dynamiquement. La solution retenue dans l'API de SPACE (Chevalier *et al.*, 2006; Bois *et al.*, 2010) basée sur les KPN est plutôt de fournir des fonctions explicites d'écriture bloquante (en plus des fonctions d'écriture non-bloquante) qui permettent de contrôler le flot de communications. Une telle écriture bloquante attend un acquittement du lecteur avant de se terminer. Cet acquittement est automatiquement envoyé par le lecteur lorsqu'il lit un message envoyé par une écriture bloquante. Le concepteur de l'application embarquée peut ainsi rendre bloquantes certaines écritures au niveau de la spécification afin qu'elle puisse s'exécuter avec des canaux bornés sans interblocage artificiel.

Une autre limite des KPN est qu'ils ne peuvent pas exprimer des concepts importants du traitement temps-réel tels que les timeouts, les interruptions, la scrutation (polling) ou les contraintes temps-réel étant donné l'absence d'une opération de lecture non-bloquante (Panangaden et Stark, 1988). Ainsi, les API de spécification de systèmes embarqués YAPI, TTL, WFIFO et SPACE de même que le modèle de calcul FunState (Strehl *et al.*, 2001) étendent la sémantique opérationnelle des KPN en permettant également les lectures non-bloquantes sur un canal. Le modèle de calcul des réseaux de processus réactifs (RPN : Reactive Process Networks) (Geilen et Basten, 2004) étend aussi le modèle KPN en y ajoutant le traitement d'évènements sporadiques. Ainsi, un RPN vérifie régulièrement la présence d'un tel évènement sporadique, ce qui correspond à une lecture non-bloquante, et modifie la configuration

du réseau de processus en réponse à cet évènement. Bien que ces travaux étendent la sémantique opérationnelle des KPN, ils n'étendent pas la sémantique dénotationnelle des KPN et celle-ci ne s'applique pas à ces modèles avec lecture non-bloquante. En effet, les KPN sont un modèle de calcul non temporisé et les aspects qui dépendent du temps ou de l'ordonnancement sont équivalents à des processus non-déterministes dans un réseau de processus non temporisé (Panangaden et Stark, 1988; Panangaden et Shanbhogue, 1992). Un processus non-déterministe n'est pas une fonction de ses séquences d'entrées vers des séquences de sortie, mais représente plutôt une relation entre celles-ci. Or il a été démontré que de connaître la relation entre les entrées et les sorties de chaque processus et les connexions entre ceux-ci n'est pas suffisant pour déterminer la relation entre les entrées et les sorties d'un réseau de processus (Brock et Ackerman, 1981). Cela explique pourquoi la sémantique dénotationnelle des KPN ne peut être étendue à un modèle qui permet les lectures non-bloquantes tout en demeurant non temporisé.

2.1.5 Modèles temporisés de flots de données

Une solution au manque d'expressivité des KPN est d'étendre ce modèle en lui ajoutant une notion explicite de temps. Ainsi, dans les modèles de réseaux de processus à évènements discrets (DEPN : Discrete Event Process Networks) (Yates, 1993; Lee, 1999; Liu et Lee, 2008), les processus traitent des séquences de jetons temporisés $E = (e_1, e_2, ..)$ tel que chaque jeton $e_i = (t_i, v_i)$ a une valeur v_i et apparaît au temps t_i. Il est généralement exigé que les jetons apparaissent par ordre chronologique ($i < j$ implique $t_i < t_j$) dans chaque séquence. Un type de processus d'un intérêt particulier pour le modèle des DEPN est le processus delta-causal, qui impose un délai d'au moins $\Delta > 0$ entre le temps où un jeton apparait dans une séquence d'entrée et le temps où la séquence de sortie est modifiée en réaction à ce jeton. On peut définir, comme pour les KPN, un ensemble d'équations reliant les séquences d'entrée et de sortie des différents processus et, si chacun d'entre eux est delta-causal, alors le théorème Banach du point fixe (Yates, 1993; Lee, 1999) garantit l'existence et l'unicité d'un point fixe qui est la solution de ce système d'équations. La sémantique dénotationnelle des DEPN définit ce point fixe unique comme le comportement du réseau de processus.

Cette sémantique dénotationnelle vient cependant avec un prix : chaque processus d'un DEPN doit spécifier non seulement sa fonctionnalité mais également sa performance exacte. Cela est peu pratique pour la spécification des systèmes embarqués : la performance exacte de chaque processus n'est généralement pas connue au moment de la spécification et une exploration architecturale subséquente vise justement à faire varier ces performances. Or toute variation de performance transforme un DEPN en un DEPN différent : il n'est pas possible, au niveau de la sémantique dénotationnelle des DEPN, de comparer entre elles

deux implémentations d'une même spécification embarquée ou de vérifier que les raffinements effectués par l'exploration architecturale préservent la fonctionnalité de la spécification.

2.1.6 Réseaux de processus temps-réel

La contribution du chapitre 4 de cette thèse est de définir un nouveau modèle de calcul, les réseaux de processus temps réel (RTPN : Real-Time Process Networks) qui permettent de relier la sémantique opérationnelle des API de spécification des systèmes embarqués avec la sémantique dénotationnelle des KPN et des DEPN. Ainsi, les processus d'un RTPN sont des processus déterministes et séquentiels non temporisés tout comme ceux des KPN. La différence se trouve au niveau des canaux qui relient les processus : certains canaux peuvent admettre des lectures non bloquantes, comme les API de spécification des systèmes embarqués. Une extension par rapport à ces API est que certains canaux peuvent également être de taille finie et admettre une écriture non-bloquante destructive lorsque le canal est plein, ce qui permet de modéliser les senseurs échantillonnés et les périphériques d'entrée/sortie. En présence de tels canaux, le comportement du RTPN dépend de l'ordonnancement des processus, plus précisément de l'ordre dans lequel les différents processus lisent et écrivent dans ces canaux. Un ordonnancement associe un temps à chacune des opérations du RTPN et on associe un DEPN équivalent à chacun de ces ordonnancements. On démontre également, via le théorème Banach du point fixe, que chaque DEPN associé à un RTPN donné a un point fixe unique (et donc un comportement unique) si chaque canal du RTPN a un comportement delta-causal.

On caractérise également les différents ordonnancements par des séquences de bits associées aux ports de lecture et d'écriture de chaque canal. Deux ordonnancements d'un RTPN sont fonctionnellement équivalents si et seulement si ils génèrent les mêmes séquences de bits pour chaque port du RTPN. Cette définition crée des classes d'équivalences entre les ordonnancements qui permettent de comparer entre elles deux implémentations fonctionnellement équivalentes mais avec des performances différentes. Un RTPN peut également être modélisé par un KPN paramétré par ces séquences de bits. Cette modélisation du RTPN sous la forme d'un KPN paramétré démontre l'existence et l'unicité d'un comportement du RTPN (qui est le plus petit point fixe du KPN paramétré) pour un ensemble donné de séquences de bits, ce qui permet d'isoler les effets de l'ordonnancement et des aspects dépendant du temps.

À notre connaissance, il s'agit des premiers travaux qui relient ainsi la sémantique opérationnelle des API de spécification des systèmes embarqués à la sémantique dénotationnelle des KPN et des DEPN.

2.2 Synthèse matérielle au niveau système

Dans les méthodologies de conception de systèmes embarqués au niveau système, la spécification des processus ou modules de l'application se fait généralement à un haut niveau d'abstraction : les calculs des modules se font à un niveau comportemental alors que leurs communications sont à un niveau transactionnel, sans modéliser cycle par cycle l'état du module ou la valeur des signaux sur chacune des broches d'entrée et de sortie. L'implémentation finale du module devra par contre communiquer avec un protocole précis au cycle et à la broche près alors que les opérations du module seront ordonnancées au cycle près, que ce soit directement dans le cas d'une implémentation matérielle ou au travers du jeu d'instructions du processeur dans le cas d'une implémentation logicielle. Il y a un écart sémantique entre le niveau d'abstraction de la spécification et de l'implémentation autant pour ce qui est des communications que des calculs. Des méthodes de raffinement et de synthèse visent à combler cet écart et on cherche à les automatiser autant que possible afin d'accélérer la conception et l'implémentation des systèmes embarqués. Étant donné que la synthèse logicielle des modules spécifiés avec SPACE a déjà été décrite dans (Chevalier *et al.*, 2006), le chapitre 5 de cette thèse se concentre plutôt sur le raffinement des communications et la synthèse pour l'implémentation matérielle des modules. On garde néanmoins en tête le fait que des modules implémentés en matériel doivent pouvoir communiquer correctement avec des modules implémentés en logiciel.

2.2.1 Intégration d'un bloc matériel existant

Une façon d'aborder ce problème est de supposer qu'il existe déjà un bloc matériel (un IP) qui est capable d'implémenter en matériel les fonctionnalités du module de la spécification. Il faut alors générer pour cet IP un adaptateur et un contrôleur qui rendent compatibles les communications et l'exécution de cet IP avec le reste du système embarqué. Ainsi, (Jung *et al.*, 2008) et ESPAM (Nikolov *et al.*, 2008a) génèrent de tels adaptateurs et contrôleurs qui permettent d'intégrer des IP lors de la synthèse d'une spécification vers une implémentation embarquée, en supposant que ces IP suivent respectivement la sémantique des modèles de calcul SDF et KPN. Le défaut de cette approche est qu'il n'est pas garanti qu'il existe déjà un IP pour un module donné et que, même si un tel IP existe, il n'est pas nécessairement conforme au modèle de calcul retenu.

2.2.2 Synthèse comportementale d'un bloc matériel

Une autre façon d'aborder ce problème est de synthétiser directement un IP sur mesure à partir de la spécification du module tout en s'assurant que l'IP synthétisé soit conforme par

construction avec le protocole de communications et le modèle de calcul du système embarqué. La synthèse comportementale permet de générer un bloc matériel au niveau RTL à partir d'un code séquentiel C ou SystemC (Coussy *et al.*, 2009). Il existe plusieurs outils commerciaux de synthèse comportementale, notamment Cynthesizer (Meredith, 2008), Catapult (Bollaert, 2008), C-to-Silicon (Cadence Design Systems, Inc., 2008) et AutoESL (Zhang *et al.*, 2008). Cependant, le code C ou SystemC fourni en entrée à ces outils doit préalablement être modifié ou annoté afin de définir une interface de communication précise au cycle et à la broche près, afin que le bloc RTL généré utilise le même protocole que le reste du système. Ainsi, ces outils comblent l'écart sémantique au niveau des calculs mais non au niveau des communications : un raffinement des communications et une synthèse d'interface doivent être réalisés avant la synthèse comportementale.

2.2.3 Raffinement des communications

Des méthodes pour automatiser le raffinement des communications d'un niveau transactionnel vers un protocole précis au cycle et à la broche près ont été présentées dans (Abdi *et al.*, 2003; Klingauf et Gunzel, 2005; Gerstlauer *et al.*, 2007). Dans (Klingauf et Gunzel, 2005), des transacteurs traduisent lors de la simulation un protocole TLM utilisant des canaux FIFO abstraits en un protocole RTL utilisant un bus partagé. Ces transacteurs exigent cependant l'écriture préalable de fonctions pour la sérialisation et la désérialisation des types de données abstraits de C/C++ en des représentations précises au bit près pour la transmission sur le bus. Cela permet néanmoins de souligner l'importance de la sérialisation dans le raffinement des communications : la spécification des systèmes embarqués à un niveau comportemental implique l'utilisation de types de données abstraits, incluant des structures de données définies par l'utilisateur, alors que les communications au niveau RTL impliquent des vecteurs de bits. Il existe plusieurs manières de représenter un même type de données par un vecteur de bits et il faut s'assurer que la représentation choisie pour les communications soit compatible avec celle des autres modules du système, notamment ceux qui sont implémentés en logiciel. Ainsi, (Abdi *et al.*, 2003) automatise la génération de fonctions de sérialisation et de désérialisation pour des types de données SpecC (Gajski *et al.*, 2000) et ces fonctions sont utilisées dans une synthèse d'interface qui remplace les communications au niveau TLM par des communications au niveau RTL. Dans (Gerstlauer *et al.*, 2007), cette sérialisation est étendue afin de permettre de spécifier un ordre des octets (*endianness*) que doivent respecter les vecteurs de bits représentant les types de données. Notre raffinement des communications et notre synthèse d'interface sont également automatisées, mais elles se basent plutôt sur le langage SystemC (IEEE, 2005) qui est supporté par une plus large gamme d'outils de synthèse comportementale. De plus, un avantage de notre sérialisation par rapport à (Abdi *et al.*,

2003) et (Gerstlauer *et al.*, 2007) est qu'elle permet d'aligner la représentation sérialisée avec celle utilisée par le compilateur C/C++ du processeur cible, ce qui permet de minimiser le temps pris par la sérialisation dans le logiciel embarqué.

2.2.4 Raffinement des communications et synthèse matérielle

Des travaux récents tentent de combiner le raffinement des communications, la synthèse d'interface et la synthèse matérielle. Ainsi, System-on-Chip Environment (SCE) (Domer *et al.*, 2008) amène les communications des modules du niveau TLM au niveau RTL à l'aide des méthodes de (Gerstlauer *et al.*, 2007), puis permet de raffiner interactivement les calculs des modules d'un niveau comportemental vers un niveau RTL à l'aide de l'outil présenté dans (Shin *et al.*, 2008). Un tel raffinement interactif implique la participation de l'utilisateur aux opérations d'allocation, d'ordonnancement et d'assignation alors que ces opérations sont automatisées dans les outils commerciaux de synthèse comportementale. Notre synthèse matérielle est ainsi complètement automatisée grâce à l'utilisation d'un tel synthétiseur comportemental.

Dans (Tibboel *et al.*, 2007), le protocole de communications de TTL est raffiné du niveau TLM vers un protocole précis au cycle et à la broche près sur un bus AMBA (Flynn, 1997) à l'aide de transacteurs et d'adaptateurs de bus AMBA. Cette synthèse d'interface peut ensuite être suivie d'une synthèse comportementale du module à l'aide des outils Cynthesizer ou Catapult. Cependant, cette synthèse d'interface a plusieurs limites. D'abord, elle exige que les modèles TLM transfèrent seulement des types de données primitifs C ou SystemC. Ensuite, l'utilisateur doit manuellement spécifier la représentation précise au bit près de ces types de données et modifier le code du module pour remplacer le port du canal TLM par celui du transacteur TLM-RTL. De plus, le lien avec l'outil de synthèse comportementale (présentation du code dans un format acceptable à l'outil et création d'un projet dans l'outil pour la synthèse d'un module) y est manuel alors qu'il est automatisé dans notre méthode.

Dans la méthodologie d'exploration architecturale et de synthèse des systèmes embarqués SystemCoDesigner (Keinert *et al.*, 2009), Cynthesizer peut être utilisé pour obtenir une implémentation matérielle des modules. Ces modules sont spécifiés sous la forme d'acteurs à l'aide de la librairie SysteMoC (Falk *et al.*, 2006) basée sur le modèle de calcul FunState. Chaque acteur est spécifié comme une machine à états finis et effectue ses calculs ainsi que ses communications au travers de canaux FIFO lors des transitions entre les états. Les transitions entre les états sont activées lorsqu'une condition de garde est remplie, par exemple lorsqu'un canal d'entrée contient un nombre suffisant de jetons. Cette spécification sous la forme d'une machine à états finis est ainsi plus près du niveau RTL et plus restrictive que le code séquentiel C/C++ ou SystemC qui est accepté par notre méthode. Avant la synthèse

comportementale avec Cynthesizer, chaque acteur SysteMoC est raffiné par SystemCoDesigner en un module SystemC. Cela implique notamment un raffinement des communications par FIFO abstraites de SysteMoC vers des communications SystemC au niveau RTL via une synthèse d'interface. Cependant, ces communications semblent supporter uniquement les types de données primitifs de C/C++ et aucune discussion n'est faite quant à la sérialisation des types de données.

Alors que (Sarkar *et al.*, 2009) compare les implémentations RTL d'un module obtenues par trois outils de synthèse comportementale à une implémentation RTL codée à la main pour le même module, nos travaux effectuent une telle comparaison pour un flot de synthèse matérielle qui comprend non seulement la synthèse comportementale, mais également le raffinement des communications et la synthèse d'interface.

2.3 Profilage de performance

2.3.1 Profilage d'un code logiciel

Le profileur de code logiciel le plus connu est Gprof (Graham *et al.*, 1982). Il procède par une instrumentation du code logiciel au point d'entrée et aux points de sortie de chaque fonction et par un échantillonnage périodique de la valeur du compteur de programme (PC : Program Counter). Cela permet de générer un graphe d'appel de fonctions et d'estimer le temps d'exécution de chaque fonction du code logiciel. Cependant, ce profilage est intrusif dans le sens où il implique l'ajout d'instructions de profilage et d'échantillonnage au code logiciel. Cela modifie alors le temps d'exécution, la taille en mémoire et possiblement le comportement (McDowell et Helmbold, 1989) de ce logiciel et cela biaise donc les résultats obtenus par ce profilage. De plus, l'échantillonnage introduit une erreur supplémentaire à celle causée par l'instrumentation intrusive. Les compteurs de performance intégrés à certains processeurs peuvent être utilisés pour échantillonner le PC lors du traitement d'une interruption produite par le débordement d'un tel compteur, mais les erreurs d'instrumentation et d'échantillonnage demeurent (Sprunt, 2002).

2.3.1.1 Profilage par exécution sur FPGA

Une solution à ce problème est d'examiner et d'analyser la valeur du PC non pas à partir du logiciel qui s'exécute sur le processeur, mais plutôt à partir de l'extérieur du processeur. Cela peut être fait alors que le logiciel est simulé sur un ISS ou exécuté sur un processeur dans l'implémentation cible. Cela permet de diminuer ou même d'éliminer l'instrumentation du code logiciel. Ainsi, SnoopP (Shannon et Chow, 2004), COMET Profiler (Finc et Zemva, 2005) et (Hough *et al.*, 2007) implémentent sur FPGA des profileurs pour respectivement

les processeurs MicroBlaze (Xilinx Inc., 2005), Nios (Altera Corp., 2004) et LEON2 (Gaisler, 2003). Ces profileurs sont des blocs matériels qui prennent en entrée le signal d'horloge du système et un signal correspondant au PC du processeur afin de compter le nombre de cycles passés dans des plages d'instructions définies par l'utilisateur. Ces profileurs considèrent qu'une plage s'exécute lorsque le PC est compris entre l'adresse de début et de fin de la plage dans le code assembleur. Cela permet de réaliser un profilage non-intrusif de ces plages de code. Par contre, cela pourrait donner un portrait incomplet du temps d'exécution attribuable à une plage si elle appelle des fonctions dont les instructions se trouvent à l'extérieur de cette plage. Il serait possible de profiler également les plages d'instructions correspondant aux fonctions appelées par la plage parente, mais rien ne garantit que ces fonctions sont seulement appelées à partir de cette plage parente. Le profileur AddressTracer (Saad *et al.*, 2009) s'attaque à ce problème en considérant qu'une plage (dans ce cas-ci, une fonction) commence son exécution lorsque le PC d'un MicroBlaze sur FPGA est égal à l'adresse de début de la fonction et se termine lorsque le PC est égal à l'adresse de fin. Cela permet de profiler exactement le temps d'exécution d'une fonction ayant un seul point de sortie, mais corrompt le profilage si la fonction en a plusieurs.

2.3.1.2 Profilage par simulation sur ISS

Les profileurs sur FPGA, étant intégrés sur puce, sont limités dans la quantité d'informations qu'ils peuvent recueillir et le nombre de plages de code qu'ils peuvent profiler. Le profilage par simulation sur ISS peut effectuer un profilage plus détaillé étant donné que la simulation s'exécute sur un système hôte avec bien plus de mémoire disponible qu'un FPGA. Ainsi, (Kwon *et al.*, 2004) et (Pimentel *et al.*, 2008) profilent le temps d'exécution des fonctions par une simulation du logiciel cible sur respectivement un ISS ARM (ARM Ltd., 2010) et un ISS SimpleScalar (Austin *et al.*, 2002). Le profilage de (Kwon *et al.*, 2004) supporte également les fonctions avec plusieurs points de sortie. Cependant, ces deux méthodes demeurent intrusives car le logiciel est instrumenté au niveau du code source (Kwon *et al.*, 2004) ou assembleur (Pimentel *et al.*, 2008) pour identifier les points d'entrée et de sortie des fonctions profilées. MEMTRACE (Hubert et Stabernack, 2009) réalise à l'aide d'un ISS un profilage non-intrusif d'un logiciel et de ses accès au bus et à la mémoire. Il commence par extraire du code assembleur du logiciel le nom et l'adresse de chaque fonction et de chaque variable globale. L'exécution de ce logiciel est ensuite simulée sur une plate-forme virtuelle comprenant un ISS ARM, un modèle de bus AMBA et un modèle de mémoire. Pendant la simulation, les accès de l'ISS aux caches d'instructions et de données, au bus et aux mémoires d'instructions et de données sont profilées. Cela permet de déterminer de manière non intrusive le temps d'exécution de chaque fonction, le nombre d'accès à chaque variable, le taux de

réussite des accès à la cache et la charge du bus.

2.3.2 Profilage d'une plate-forme logicielle/matérielle

Plusieurs outils commerciaux permettent de simuler et de profiler une plate-forme comprenant des processeurs, des bus, des mémoires et des accélérateurs matériels. Seamless (Mentor Graphics Corp., 2005) est un environnement de co-vérification et co-simulation logiciel/matériel précise au cycle près dans lequel un code logiciel est simulé sur un ISS et l'architecture matérielle, qui est spécifiée en un langage de description matériel (HDL : Hardware Description Language), est simulée au niveau RTL par un simulateur logique. Le profilage intégré à Seamless permet d'obtenir le diagramme de Gantt des appels aux fonctions logicielles (le temps du début et de la fin de l'exécution de chaque fonction) et d'obtenir des statistiques sur l'utilisation de la cache, de la mémoire et du bus. Le simulateur logique peut également produire une trace (*waveform*) des signaux RTL de l'architecture matérielle. Metrix (VaST Systems Technology Corp., 2008) et CoWare Platform Architect (CoWare Inc., 2010) profilent une co-simulation logiciel/matériel d'un ISS avec une architecture matérielle au niveau TLM afin d'obtenir le graphe d'appel des fonctions logicielles, de visualiser les transferts sur le bus et de déterminer l'utilisation de la cache, de la mémoire et du bus. CoWare Platform Architect (CoWare Inc., 2010) permet également d'obtenir le diagramme de Gantt des appels aux fonctions logicielles et de produire une trace des ports TLM, des signaux RTL ou d'une variable membre d'un module SystemC de l'architecture matérielle.

Bien que ces outils permettent de profiler de manière non-intrusive un logiciel et l'architecture matérielle dans laquelle il s'exécute, ce profilage demeure à relativement bas niveau. Par exemple, ils voient le code logiciel profilé comme un ensemble de fonctions sans en interpréter la sémantique. Si ce logiciel contient un RTOS, ils ne peuvent donc pas détecter les changements de contexte, ni distinguer les différentes tâches logicielles, ni profiler les communications entre ces tâches logicielles. Notre profilage présenté au chapitre 6 est au contraire capable de profiler les opérations du RTOS et des tâches logicielles tout en demeurant non-intrusif. De plus, ces outils supposent que le partitionnement logiciel/matériel du système a déjà été choisi : les métriques fournies pour les parties logicielle et matérielle sont si différentes qu'il n'est pas possible de comparer une implémentation logicielle et une implémentation matérielle d'une même tâche. Notre profilage est quant à lui capable d'extraire des métriques comparables quant au temps d'exécution et aux communications des tâches qui se trouvent en logiciel ou en matériel. En particulier, notre profilage est capable de détecter et d'extraire des métriques comparables pour des communications entre deux modules matériels, un module logiciel et un module matériel ou entre deux modules logiciels qui s'exécutent sur le même processeur ou sur deux processeurs différents.

2.3.3 Profilage des changements de contexte

Deux méthodes de profilage d'un code logiciel sur FPGA tentent de détecter les changements de contexte. Dans (Hough *et al.*, 2007), l'ordonnanceur d'un système d'exploitation Linux embarqué est instrumenté de manière à écrire l'identificateur du processus courant sur le bus APB (AMBA Peripheral Bus) d'un processeur LEON2 après un changement de contexte. Cela implique donc que ce profilage est intrusif. DAProf (Shankar et Lysecky, 2009) suppose que les tâches d'un logiciel embarqué sont représentées par un ensemble de plages de code disjointes. Lors du profilage sur FPGA, DAProf considère qu'une tâche s'exécute tant que le PC est compris entre l'adresse de début et l'adresse de fin de la plage de code spécifiée par l'utilisateur. Un changement de contexte est détecté lorsque le PC change de plage de code. Cependant, si deux tâches différentes appellent une fonction partagée, notamment une des fonctions offertes par un RTOS, alors le code correspondant à ces deux tâches ne forme pas des plages disjointes. DAProf détectera donc un changement de contexte à chaque fois que cette fonction partagée est appelée même si aucun changement de contexte ne se produit en réalité. De plus, ces travaux portent seulement sur le profilage d'un logiciel. Notre profilage détecte les changements de contexte de manière non-intrusive et exacte tout en s'appliquant à une plate-forme logicielle/matérielle complète.

2.3.4 Profilage des arguments et des valeurs de retour

Contrairement à notre profilage, les travaux présentés jusqu'à maintenant ne profilent ni les valeurs des arguments passés en paramètres aux fonctions du code logiciel, ni les valeurs de retour de ces fonctions. À notre connaissance, le seul autre outil qui extrait de manière non intrusive ces informations lors d'une simulation sur ISS est ReSP qui a récemment été présenté dans (Beltrame *et al.*, 2009) et qui le réalise en accédant au PC et aux registres d'un ISS ARM. Cependant, ces informations sont utilisées à des fins fort différentes. ReSP les extrait non pas pour réaliser un profilage de performance, mais plutôt afin d'intercepter des appels à certaines fonctions dans le code cible et de les émuler nativement sur le processeur hôte pour accélérer la simulation. Notre profilage utilise ces informations pour extraire les dépendances de communications entre les tâches et pour réaliser un profilage de performance des fonctions du RTOS et des communications entre tâches selon leurs paramètres et leur résultat. En effet, si on suppose sans perte de généralité qu'une fonction f a un paramètre $n \geq 0$ et exécute un algorithme de complexité $O(n^a)$ tel que $a > 0$, alors le temps d'exécution de cette fonction dépend de la valeur de n. Notre profilage permet de mettre en lumière cette relation en profilant pour chaque appel de f son temps d'exécution et la valeur de n.

2.3.5 Portabilité du profilage

Alors que (Hubert et Stabernack, 2009) présente une méthodologie pour porter le profilage de PC et d'accès mémoire de MEMTRACE vers d'autres processeurs, le chapitre 6 présente une méthodologie pour porter vers d'autres processeurs non seulement le profilage du temps d'exécution des fonctions, mais également le profilage de leurs arguments et de leurs valeurs de retour. De plus, nos travaux sont à notre connaissance les seuls à présenter un profilage non-intrusif générique utilisant GDB (Stallman *et al.*, 2002) et à présenter une méthodologie pour porter vers d'autres RTOS le profilage non-intrusif des changements de contexte et des communications entre tâches. En effet, bien que (Benini *et al.*, 2003) et (Fummi *et al.*, 2009) utilisent GDB pour réaliser une co-simulation logiciel/matériel entre un ISS et un simulateur SystemC, ils n'implémentent pas de profilage non-intrusif.

2.4 Estimation du temps d'exécution

Diverses méthodes permettent d'estimer le temps d'exécution d'une implémentation d'un logiciel ou d'un système embarqué sans avoir à simuler chacune de ses opérations précisément au cycle près. Ces méthodes se classent en trois catégories principales : l'analyse statique, l'instrumentation d'une simulation TLM avec des annotations temporelles et l'évaluation d'une trace générée dynamiquement.

2.4.1 Analyse statique

Les méthodes d'analyse statique pour l'estimation du temps d'exécution consistent à réaliser un ordonnancement statique des tâches qui ont seulement des dépendances de données ou à estimer le temps d'exécution de la pire séquence de blocs de base qui peut être exécutée selon leurs dépendances de contrôle.

2.4.1.1 Estimation par ordonnancement statique

La performance d'un système embarqué peut être caractérisée à l'aide d'un modèle de graphe de tâches. Un graphe de précédences de tâches (TPG : Task Precedence Graph) est un graphe acyclique orienté (DAG : Directed Acyclic Graph) $G(V, A)$ tel que les noeuds V sont les tâches et les arcs A sont les contraintes de précédence entre les tâches (Kwok et Ahmad, 1999). Une tâche ne peut pas démarrer son exécution tant que toutes les tâches qui la précèdent n'ont pas terminé leur exécution. Ces contraintes de précédence représentent typiquement des dépendances de données entre les tâches. Ainsi, une tâche reste bloquée tant que les données sur chacun de ses arcs entrants ne sont pas arrivées, puis elle s'exécute

pendant un certain temps et écrit ses données sur chacun de ses arcs sortants à la fin de son exécution. Cela correspond donc essentiellement au modèle de calcul des SDF homogènes et acycliques (Knerr *et al.*, 2008). Il est donc possible de réaliser un ordonnancement statique d'un TPG, par exemple sur un nombre fixe de processeurs (Kwok et Ahmad, 1999) ou sur une architecture composée d'un processeur et d'un co-processeur matériel (Wiangtong *et al.*, 2002).

Si on associe un temps d'exécution $e(v)$ à chaque noeud $v \in V$ et un temps de communication $c(a)$ à chaque arc $a \in A$, la performance d'un TPG peut donc être estimée en effectuant un ordonnancement statique du TPG pour trouver son temps d'exécution total. Dans (Vallerio et Jha, 2003), une méthode est présentée pour extraire, à partir d'une analyse statique et d'un profilage intrusif d'un code C séquentiel sur le processeur hôte, un TPG avec ses valeurs de $e(v)$ et $c(a)$. Cependant, les valeurs recueillies ne s'appliquent pas nécessairement au processeur cible. Étant donné que l'ordonnancement statique d'un TPG générique est un problème NP-complet (Kwok et Ahmad, 1999), des heuristiques sont généralement utilisées. Une telle estimation suppose donc que l'estimation et la synthèse du système utilisent le même ordonnancement statique. La précision de cette estimation dépend également des caractéristiques de l'architecture qui sont prises en compte par l'ordonnancement statique. Il a notamment été démontré que l'estimation par ordonnancement statique d'un TPG sur une architecture à bus partagés a une très grande erreur si on ne tient pas compte de la contention sur le bus (Sinnen et Sousa, 2004). Il est donc nécessaire d'ordonnancer les arcs du TPG sur les bus en plus d'ordonnancer les noeuds sur les processeurs et les accélérateurs matériels. Il a aussi été démontré que, si un processeur doit être impliqué dans les communications réalisées par les tâches qui lui sont assignées, alors l'ordonnancement statique doit tenir compte du fait que le processeur est réservé par une communication pour un certain temps avant, pendant ou après celle-ci (Sinnen *et al.*, 2006).

Le principal inconvénient de l'estimation par ordonnancement statique des TPG est qu'ils ne représentent adéquatement qu'une quantité restreinte d'applications. Ainsi, ils ne peuvent exprimer ni les tâches qui ont des dépendances de contrôle telles que des lectures ou écritures conditionnelles, ni les tâches qui effectuent des lectures ou écritures au milieu de leur exécution, tels les processus d'un KPN. Ces inconvénients demeurent même si l'estimation s'effectue par un ordonnancement statique des SDF (Knerr *et al.*, 2008; Lee *et al.*, 2010).

Un modèle de tâches à plus gros grains est le graphe d'interaction de tâches (TIG : Task Interaction Graph) qui est un graphe non orienté $G(V, E)$ dans lequel les noeuds V représentent des tâches et les arêtes E des flots de communications bidirectionnels entre les tâches (Stone, 1977). Les tâches sont supposées devoir toutes s'exécuter séquentiellement, et ce même si elles sont assignées à plusieurs processeurs. Le temps d'exécution total du TIG est

donc égal à la somme des temps d'exécution de chaque noeud et des temps de communication de chaque arête, qui peuvent varier selon l'architecture ciblée. Cela est donc équivalent à une estimation par ordonnancement statique d'un TPG dégénéré qui serait un arbre unaire, chaque noeud (sauf un noeud source) ayant exactement un prédécesseur et chaque noeud (sauf un noeud puits) ayant exactement un successeur. Le modèle TIG est notamment utilisé pour estimer le temps d'exécution dans le partitionnement logiciel/matériel sur une architecture composée d'un processeur et d'un co-processeur (Arato *et al.*, 2005).

2.4.1.2 Estimation statique du WCET

Un programme séquentiel peut être représenté par un graphe de flot de contrôle (CFG : Control Flow Graph). Un CFG est un graphe orienté $G = (V, A)$ tel que les noeuds V sont des blocs de base et les arcs A sont des branchements (Allen, 1970). Un bloc de base est une séquence linéaire d'instructions sans branchements : si la première instruction d'un bloc de base s'exécute, alors toutes les autres instructions s'exécuteront nécessairement ensuite. Une exécution donnée d'un programme séquentiel correspond donc à un chemin donné dans son CFG et à l'exécution d'une séquence de blocs de base, dans laquelle un bloc de base donné peut être exécuté plus d'une fois. Une méthode pour estimer le temps d'exécution d'un programme séquentiel consiste donc à associer un temps d'exécution à chaque bloc de base de son CFG et à additionner les temps de chaque exécution de chaque bloc de base selon le chemin parcouru dans le CFG. Il reste alors à déterminer quel chemin utiliser pour déterminer le temps d'exécution du programme. Les deux principales approches sont l'énumération de tous les chemins pour trouver le chemin avec le pire temps d'exécution (WCET : Worst-Case Execution Time) et la simulation du programme avec un stimulus donné pour obtenir un cas donné qu'on suppose représentatif. En effet, étant donné qu'un CFG contient des dépendances de contrôle, il n'est pas possible de réaliser un ordonnancement statique des blocs de base contrairement aux tâches d'un TPG.

Cinderella (Li et Malik, 1997) commence par analyser le WCET de chaque bloc de base d'un CFG compilé pour un processeur Intel i960KB (Intel Corp., 2002), puis réalise une énumération implicite des chemins du CFG afin de trouver le pire chemin qui respecte un ensemble de contraintes structurelles et fonctionnelles. Les contraintes structurelles sont automatiquement extraites du CFG et sont un ensemble d'équations à variables entières. Chaque équation exprime le fait que le nombre de fois où un chemin donné entre dans un bloc de base donné est égal au nombre de fois où il en sort, ce qui est égal au nombre de fois où le bloc de base est exécuté. Les contraintes fonctionnelles sont fournies par l'utilisateur et sont des équations qui spécifient notamment le nombre maximal d'itérations qui peuvent être effectuées par une boucle. Il est en effet nécessaire de borner le nombre d'itérations de

chaque boucle pour éviter que le pire chemin consiste trivialement à effectuer une boucle infinie pendant un temps infini. Cinderella trouve le WCET par une programmation linéaire en nombre entiers (ILP : Integer Linear Programming) qui maximise le temps d'exécution du CFG tout en respectant les contraintes structurelles et fonctionnelles. Cinderella permet de trouver une borne supérieure au WCET d'un code logiciel séquentiel. En effet, il s'agit d'une borne et non d'une valeur exacte étant donné que le pire chemin trouvé par Cinderella peut en réalité être impossible et que les blocs de base sur un tel chemin ne s'exécuteront pas nécessairement tous avec leur WCET. Dans (Yoo *et al.*, 2006), cette approche est étendue pour trouver le WCET d'un accélérateur matériel correspondant à un code séquentiel. Cependant, cette approche est davantage pessimiste que Cinderella, car elle suppose que les blocs de base s'exécutent séquentiellement comme en logiciel alors qu'un accélérateur matériel pourrait plutôt les exécuter en parallèle, par exemple grâce à un déroulage de boucle.

L'autre problème auquel fait face cette approche est qu'elle ne s'applique pas à un CFG qui contient un bloc de base effectuant une opération bloquante sur un canal externe et dont le temps d'exécution est donc non-déterministe du point de vue du CFG. Ainsi, elle s'appliquerait difficilement au code séquentiel d'un processus KPN qui peut effectuer des lectures bloquantes à des endroits arbitraires dans le code. Cette estimation du WCET peut par contre être utilisée pour un processus SDF ou pour les modes d'exécution d'un processus CSDF. En effet, comme toutes les lectures sont effectuées au début d'un tel processus, il est possible d'estimer le WCET du code subséquent à ces lectures. Ainsi, SoCDAL (Ahn *et al.*, 2008) évalue, à l'aide des méthodes de (Li et Malik, 1997) et (Yoo *et al.*, 2006) décrites au paragraphe précédent, le WCET des implémentations logicielles et matérielles des modes d'exécution des processus d'un CSDF. Il réalise ensuite un ordonnancement statique de ce CSDF sur une architecture comprenant un nombre variable de processeurs et d'accélérateurs matériels afin d'en évaluer le temps d'exécution.

SymTA/S (Henia *et al.*, 2005) évalue le WCET d'un ensemble de tâches ordonnancées dynamiquement sur un nombre variable de processeurs ayant chacun une politique d'ordonnancement. Les tâches considérées par SymTA/S sont semblables à un processus SDF dans le sens où chaque tâche est activée par l'arrivée d'une donnée sur un FIFO d'entrée et s'exécute, avec un WCET qui doit être fourni, jusqu'à la production d'une donnée sur un FIFO de sortie. Les flots de données en entrée et en sortie des tâches y sont caractérisés par leur période, leur gigue (*jitter*) et la distance minimale entre deux évènements. Ainsi, pour un ensemble donné de caractéristiques des flots de données à l'entrée du système, SymTA/S détermine, à l'aide des WCET des taches et des politiques d'ordonnancement des processeurs, les caractéristique des flots de données à l'interne et à la sortie du système afin de trouver une borne supérieure au WCET du système.

Ces méthodes se basant sur une estimation du WCET sont bien adaptées à la conception des systèmes embarqués ayant des contraintes temps-réel dures. En effet, dans un tel système, la priorité est d'éviter toute violation d'une contrainte temps-réel et il est donc nécessaire de déterminer une borne supérieure au WCET, même si cette borne est pessimiste. Par contre, dans un système embarqué ayant des contraintes temps-réel molles, la performance moyenne est plus importante et il est suffisant que les contraintes temps-réel soient respectées la plupart du temps : la violation occasionnelle d'une telle contrainte n'entraîne aucune conséquence majeure. Dans ce cas, d'assurer que le WCET du système respecte ces contraintes risquerait de demander une trop grande quantité de processeurs et d'accélérateurs matériels, ou une trop grande fréquence et consommation de puissance, pour les besoins réels du système, particulièrement si le WCET est estimé de manière pessimiste. De plus, la détermination du WCET demande généralement une grande implication de l'utilisateur pour spécifier des contraintes fonctionnelles sur les chemins possibles et cela limite le potentiel d'automatisation d'une telle méthodologie.

Une méthode mieux adaptée aux systèmes embarqués ayant des contraintes temps-réel molles est de simuler l'exécution du système tel qu'exercé par un banc d'essai représentatif. Cela suppose qu'un banc d'essai soit fourni avec l'application, mais cette exigence n'impose pas un effort supplémentaire de développement étant donné qu'un tel banc d'essai a généralement déjà été développé pour valider la fonctionnalité de l'application. De plus, cela permet de cibler des types d'applications pour lesquelles les méthodes d'analyse statique du WCET ne s'appliquent pas. C'est donc une approche par banc d'essai qui est retenue au chapitre 7 de cette thèse.

2.4.2 Annotation temporelle d'une simulation TLM

La performance d'un système embarqué peut être évaluée en co-simulant son logiciel sur un ISS (ou un ensemble d'ISS si il contient plusieurs processeurs) avec son architecture matérielle au niveau RTL. Cependant, une telle co-simulation demande un temps important. Cette simulation peut être accélérée en remplaçant les modèles RTL des composants génériques tel que les bus, les mémoires et les périphériques par des modèles TLM précis au cycle près (Cai et Gajski, 2003). Cependant, un tel système peut contenir un code logiciel et des accélérateurs matériels spécifiques à l'application et pour lesquels il n'existe pas de modèles TLM précisément temporisés. Le problème qui se pose est donc de trouver comment générer automatiquement de tels modèles TLM afin d'obtenir une co-simulation au niveau TLM qui soit grandement accélérée et qui permette d'estimer adéquatement le temps d'exécution qu'on obtiendrait suite à une co-simulation précise au cycle près. L'avantage de cette estimation par simulation est que, contrairement à l'analyse statique, elle peut s'appliquer à des tâches qui

effectuent des communications à des endroits arbitraires dans le CFG et selon des dépendances de contrôle arbitraires entre les tâches. L'annotation temporelle d'un modèle TLM peut se faire à l'aide d'une analyse statique ou d'une analyse dynamique du code de l'application modélisée.

2.4.2.1 Annotation temporelle par analyse statique

Différents travaux déterminent par une analyse statique le temps d'exécution de chaque opération ou de chaque bloc de base du CFG d'un logiciel embarqué. Le code original, ou un code équivalent à celui-ci, est ensuite instrumenté statiquement avec des annotations temporelles pour produire un modèle TLM temporisé du logiciel. Cette instrumentation accumule le temps d'exécution correspondant à chaque bloc de base exécuté lors de la simulation. Le modèle TLM du logiciel se synchronise avec le reste de la plate-forme matérielle, selon le temps d'exécution accumulé depuis la dernière synchronisation, seulement lorsqu'il doit communiquer avec celle-ci. Cela permet de limiter le nombre de synchronisations et d'accélérer la simulation. En effet, un bout de code se trouvant entre deux communications, qui est appelé un segment de programme (Wolf et Ernst, 2001; Posadas *et al.*, 2004), peut contenir plusieurs blocs de base.

Ainsi, dans (Posadas *et al.*, 2004), l'instrumentation statique est implémentée par une surcharge des opérateurs C/C++ (addition, multiplication, etc.) À chaque fois qu'un tel opérateur est appelé dans le code source d'un module TLM SystemC, le temps d'exécution du module est incrémenté du temps d'exécution associé à l'opérateur. La caractérisation d'un processeur cible donné consiste donc à associer à chaque opérateur C/C++ un temps d'exécution sur le processeur cible. L'estimation réalisée par (Hwang *et al.*, 2008) suit des principes similaires. Un CFG est d'abord extrait d'un code C à l'aide de l'infrastructure de compilation LLVM (Lattner et Adve, 2004). Le temps d'exécution de chaque bloc de base est ensuite estimé statiquement, pour le processeur MicroBlaze préalablement caractérisé, selon les opérateurs qu'il contient et le pipeline du processeur. Des taux fixes de défaut (*miss*) de cache et d'erreur de prédiction de branchement sont utilisés pour estimer les pénalités associées à ces évènements. Le code C est ensuite annoté avec les délais associés aux blocs de base, puis encapsulé dans un module SystemC. Le problème avec une telle approche est que le temps d'exécution sur un processeur cible est connu pour les instructions assembleur et non pour les opérateurs d'un langage à haut niveau tel C/C++. Rien ne garantit que les compilateurs pour un processeur cible donné génèreront toujours la même série d'instructions assembleur pour un même opérateur. De plus, les appels à certains opérateurs présents dans le code source pourraient être éliminés par les optimisations du compilateur.

(Schnerr *et al.*, 2008) commence par réaliser une compilation croisée d'un code C vers un

processeur cible Infineon TriCore (Infineon Technologies AG, 2000), qui a préalablement été caractérisé. Le temps d'exécution de chaque bloc de base est déterminé statiquement selon les instructions assembleur correspondant au bloc de base et en tenant compte du pipeline du processeur. Les temps d'exécution extraits sont alors annotés pour chaque bloc de base dans le code C original, qui est ensuite encapsulé en un module SystemC. Cette méthode suppose donc que le CFG du code source C est identique au CFG obtenu suite à la compilation croisée, ce qui n'est pas nécessairement le cas en raison des optimisations du compilateur. Le code du module SystemC est également annoté pour faire appel à un modèle de cache d'instructions et un modèle de prédiction dynamique de branchement. Cela permet d'ajouter dynamiquement les pénalités de défauts (*miss*) de cache et d'erreur de prédiction de branchement au temps d'exécution du logiciel.

Dans (Gerin *et al.*, 2009), un code C/C++ est d'abord converti en une représentation intermédiaire (IR : Intermediate Representation) lors d'une compilation croisée avec LLVM. L'IR obtenue tient donc compte des optimisations de compilation, que celles-ci dépendent du processeur cible ou non, étant donné que c'est l'IR qui est directement utilisée pour la génération du code assembleur cible. Le temps d'exécution de chaque bloc de base de cette IR est analysé statiquement, pour un processeur cible ARM9 préalablement caractérisé, en tenant compte du pipeline. Des appels à une fonction d'instrumentation accumulant ces temps d'exécution sont ajoutés aux blocs de base de cette IR, puis elle est utilisée pour générer directement un code assembleur pour le processeur hôte. Cela assure que l'assembleur hôte instrumenté et l'assembleur cible ont le même CFG. Cet assembleur hôte est exécuté nativement sur un modèle TLM de processeur, qui implémente une couche d'abstraction matérielle (HAL : Hardware Abstraction Layer) permettant au logiciel simulé de communiquer avec la plate-forme matérielle. Ce modèle utilise également un taux fixe de défauts de cache d'instructions pour en estimer les pénalités.

Il est possible que plusieurs tâches logicielles soient assignées à un même processeur et ordonnancées par un RTOS. Pour éviter des erreurs importantes d'estimation, il est alors nécessaire de tenir compte non seulement de la politique d'ordonnancement du RTOS, qui sérialise l'exécution des tâches logicielles, mais également du temps d'exécution du RTOS lui-même (Hwang *et al.*, 2009). Les travaux présentés ci-dessus le font à divers degrés. Ainsi, (Schnerr *et al.*, 2008) supporte une modélisation TLM de l'ordonnancement et des changements de contexte, mais ne tient pas compte de leur temps d'exécution. Quant à (Posadas *et al.*, 2004) et (Hwang *et al.*, 2009) (qui étend (Hwang *et al.*, 2008)), ils intègrent une modélisation TLM temporisée du RTOS, mais seul (Hwang *et al.*, 2009) présente une manière de caractériser le temps d'exécution du RTOS. Cette caractérisation consiste en un profilage intrusif d'une application qui exerce les différentes primitives du RTOS dans un ordre

déterminé. Cela permet d'extraire le temps d'exécution des primitives du RTOS, tel qu'un changement de contexte ou l'utilisation d'un sémaphore. Dans (Gerin *et al.*, 2009), une méthode différente est utilisée pour un RTOS dont le code source est disponible : celui-ci est instrumenté statiquement comme les tâches logicielles et le tout est exécuté sur la HAL fourni par un modèle TLM du processeur. Par contre, le temps d'exécution de la HAL elle-même, qui comprend notamment les changements de contexte, n'est pas pris en compte.

Notre méthode d'estimation tient compte des temps d'exécution des tâches logicielles, du RTOS et de la HAL, qui sont automatiquement caractérisés. La caractérisation du RTOS est similaire à celle de (Hwang *et al.*, 2009) mais l'améliore en utilisant un profilage non-intrusif et en tenant compte de l'état du RTOS lorsqu'une de ses primitives est exercée (par exemple, libérer un sémaphore prend un temps supplémentaire si une tâche attend après celui-ci même si cela ne produit pas de changement de contexte). Notre méthode automatise également la caractérisation du temps d'exécution des accélérateurs matériels générés à l'aide de la synthèse comportementale alors que les travaux présentés ci-dessus permettent de générer des modèles TLM temporisés seulement pour les tâches logicielles. Cependant, la différence plus fondamentale entre ces travaux et notre méthode d'estimation est que celle-ci effectue plutôt une analyse des traces extraites d'un ensemble de simulations initiales d'une même application donnée. Cela permet d'évaluer rapidement une grande quantité d'implémentations possibles de cette application sans avoir à simuler sa fonctionnalité à chaque fois. Les méthodes d'instrumentation statique présentées ci-dessus pourraient être utilisées pour accélérer les simulations initiales nécessaires à la production des traces et sont donc complémentaires à notre méthode d'estimation.

2.4.2.2 Annotation temporelle par analyse dynamique

Il est possible d'insérer des annotations temporelles dans les modèles TLM des tâches de l'application non pas au niveau de leurs opérateurs C/C++ ou de leurs blocs de base, mais plutôt au niveau de leurs segments de programme, soit entre deux communications externes aux tâches. Cependant, un segment de programme peut contenir plusieurs blocs de base et constitue un CFG qui est un sous-graphe du CFG de la tâche. Le temps d'exécution d'un segment de programme peut donc varier selon le chemin parcouru dans son CFG. Il serait possible d'effectuer une analyse statique du WCET d'un segment de programme à l'aide des méthodes présentées à la section 2.4.1.2 et d'annoter le modèle TLM de la tâche en conséquence, mais à notre connaissance cette approche n'a pas été proposée dans la littérature et elle risquerait d'être pessimiste et de demander une intervention de l'utilisateur. L'autre méthode est de profiler le temps d'exécution de chaque segment de programme sur le processeur cible alors que l'application est exercée par un banc d'essai. Une telle approche se justifie

dans le cadre d'une méthodologie d'exploration architecturale, où la caractérisation initiale d'une application permettra d'accélérer les simulations d'une grande quantité d'architectures implémentant celle-ci.

Ainsi, les simulateurs VPC (Streubuhr *et al.*, 2009) et Sesame (Pimentel *et al.*, 2006) sont respectivement utilisés par les méthodologies d'exploration architecturale SystemCoDesigner (Keinert *et al.*, 2009) et Daedalus (Nikolov *et al.*, 2008c) pour estimer la performance des solutions possibles. Ces deux simulateurs exécutent un modèle non temporisé de l'application qui génère des évènements de communications et de calculs sur un modèle TLM temporisé d'une architecture qui répond à ces évènements. Une couche d'assignation relie entre eux les modèles de l'application et de l'architecture et associe notamment à chaque segment de programme un temps d'exécution donné. VPC modélise l'application comme un ensemble d'acteurs à états finis SysteMoC (Falk *et al.*, 2006) et simule l'ensemble des couches à l'aide de SystemC. Sesame effectue plutôt une co-simulation : un modèle d'application sous la forme d'un KPN est exécuté nativement avec la librairie C/C++ pthreads alors que la couche d'assignation et les modèles d'architecture sont simulés avec le simulateur à évènements discrets Pearl (Muller, 1993).

(Keinert *et al.*, 2009) obtient le temps d'exécution sur un processeur MicroBlaze des actions d'un acteur SysteMoC (essentiellement des segments de programme) à l'aide d'un profilage du PC sur FPGA semblable aux méthodes présentées à la section 2.3.1.1. Ainsi, l'utilisateur doit extraire l'adresse du point d'entrée et du point de sortie de chaque segment à profiler, configurer le profileur FPGA avec ces adresses, exécuter le profilage puis configurer la couche d'assignation de VPC selon les résultats recueillis lors du profilage. Dans (Pimentel *et al.*, 2008), le temps d'exécution logiciel des segments de programme d'un processus KPN modélisé avec Sesame est extrait à l'aide d'une co-simulation du modèle d'application KPN avec un ISS SimpleScalar qui exécute le code du processus. Ainsi, l'utilisateur doit d'abord annoter le code C/C++ du processus à caractériser pour identifier les segments de programme. Une compilation croisée du processus vers le processeur cible est ensuite réalisée, puis l'utilisateur doit instrumenter le code assembleur produit pour que le logiciel mesure le nombre de cycles écoulés pendant l'exécution d'un segment de programme. Cette méthode est donc intrusive. Un profilage du processus sur l'ISS est ensuite réalisé et, pour chaque segment de programme profilé, la valeur moyenne du temps d'exécution est extraite. Ces valeurs peuvent être utilisées par l'utilisateur pour configurer la couche d'assignation.

Alors que ces méthodes de caractérisation logicielle des segments de programme par profilage sont semi-manuelles, notre méthode par profilage sur ISS automatise pleinement toutes les étapes de la caractérisation logicielle. Notre méthode automatise également la caractérisation du temps d'exécution matériel des segments de programme par le profilage d'une

simulation SystemC mixte de l'application au niveau TLM avec le module RTL à caractériser. Elle automatise aussi la caractérisation du temps d'exécution des opérations du RTOS afin d'en tenir compte lors de l'estimation. Cette automatisation de la caractérisation permet d'adapter facilement l'estimation aux changements dans le code de l'application ou aux mises à jour du RTOS. Bien que VPC et Sesame tiennent compte de l'ordonnancement des tâches, ils ne tiennent pas compte du temps d'exécution de l'ordonnancement dynamique, ce qui peut causer d'importantes erreurs d'estimation (Streubuhr *et al.*, 2009; Pimentel *et al.*, 2008). Finalement, notre estimation par trace se base sur le temps de chaque exécution de chaque segment de programme au lieu de se baser seulement sur leur temps moyen, ce qui permet de mieux estimer la performance de ceux-ci.

2.4.3 Évaluation d'une trace dynamique

Les méthodes d'estimation du temps d'exécution par évaluation d'une trace commencent par réaliser une simulation fonctionnelle de l'application. Une trace d'évènements est ensuite extraite de cette simulation. Cette trace est ensuite analysée pour en estimer la performance sur diverses architectures sans avoir à simuler l'application sur chacune de ces architectures.

2.4.3.1 Évaluation par analyse statique

Dans (Cai *et al.*, 2004), les processus d'une simulation SpecC sont instrumentés afin d'extraire le nombre d'exécutions de chacun de leurs blocs de base et le nombre d'accès à chacun de leurs ports d'entrée et de sortie. Une analyse statique détermine, pour chaque bloc de base, le nombre et le type d'opérations SpecC correspondant à une exécution et, pour chaque port, le nombre d'octets transférés correspondant à un accès. En combinant cette analyse statique à la trace fonctionnelle, on obtient pour chaque processus le nombre et le type d'opérations SpecC exécutées ainsi que le nombre d'octets transférés. Les processus SpecC peuvent ensuite être assignés à une architecture composée de processeurs et de bus préalablement caractérisés. Cette caractérisation associe un temps d'exécution à chaque type d'opération SpecC sur le processeur de même qu'un temps de communication selon le nombre d'octets transférés. Cela permet d'obtenir des métriques de temps d'exécution et de communication pour chaque processus du système. Cependant, cette estimation suppose que le compilateur génèrera toujours la même série d'instructions assembleur pour une même opération SpecC et que le CFG de chaque module ne sera pas modifié par les optimisations du compilateur.

(Jaddoe *et al.*, 2009) suit une approche similaire en simulant un réseau KPN pour obtenir, pour chaque processus, une trace d'évènements de calcul et de communication. Chaque processus est compilé pour le processeur cible puis analysé pour extraire le nombre et le type

d'instructions assembleur exécutées en moyenne pour un évènement de calcul d'un processus. Un temps d'exécution est associé à chaque type d'instruction. Selon le nombre d'évènements de calcul et de communication produits, un temps d'exécution est ensuite associé à chaque processus en additionnant les temps de calcul et de communication. Le temps d'exécution d'un processeur est considéré comme la somme des temps d'exécution des processus qui lui sont assignés et le temps d'exécution du système est celui du processeur avec le plus long temps d'exécution. Ces méthodes d'estimation purement additives ne tiennent pas compte de l'ordonnancement des processus ou de leurs dépendances de données.

2.4.3.2 Évaluation par analyse dynamique

Une manière plus précise d'estimer la performance d'une application embarquée sur une architecture donnée est de représenter la trace sous la forme d'un graphe $G(V, E)$ dont les noeuds V sont les évènements de calcul ou de communication de la trace et dont les arcs E sont les relations de précédence entre ces évènements, telles que des dépendances de données ou de séquence. Selon l'assignation des éléments de l'application à l'architecture, un temps d'exécution ou de communication est associé à chaque évènement de calcul ou de communication de ce graphe, qui est ensuite ordonnancé pour en estimer le temps d'exécution. Une telle méthode a quelques similarités avec l'ordonnancement statique d'un TPG. Cependant, alors qu'un TPG représente l'application elle-même, une trace contient les évènements correspondant à une exécution donnée de l'application, qui ne peut pas nécessairement être ordonnancée statiquement. Ainsi, l'ordonnancement statique d'un TPG vise à décider dans quel ordre les tâches du TPG s'exécutent alors que l'ordonnancement d'une trace vise à simuler comment les évènements de la trace auraient été ordonnancés dynamiquement sur les bus et les processeurs d'une architecture donnée. Le gain de vitesse par rapport à une simulation complète vient du très haut niveau d'abstraction de la trace : un segment de programme y est simulé comme un seul évènement de calcul au lieu de simuler séparément chacune de ses instructions.

(Lahiri *et al.*, 2001) utilise une méthode d'ordonnancement de trace pour évaluer la performance de différentes architectures de communication pour une application une fois que sont fixés le partitionnement logiciel/matériel et l'assignation des tâches aux processeurs. La trace est extraite en instrumentant une co-simulation logiciel/matériel de l'application réalisée avec Ptolemy (Buck *et al.*, 1994). Cette co-simulation fixe le temps d'exécution de chaque évènement de calcul de la trace, mais les temps des évènements de communication peuvent varier selon l'architecture de communication qui reste à déterminer. La performance d'une architecture de communication donnée est estimée en simulant l'ordonnancement de la trace sur cette architecture. Ainsi, différents bus partagés ou liens point-à-point peuvent

être caractérisés par leur largeur de bits, leur latence et leur fréquence. Les bus partagés sont également caractérisés par leur politique d'arbitrage, ce qui permet de tenir compte de la contention sur le bus et de l'arbitrage lors de l'estimation du temps d'exécution du système. Cependant, cette méthode ne permet pas d'explorer le partitionnement logiciel/matériel ou de modifier l'assignation des tâches aux processeurs.

Dans (Ueda *et al.*, 2005), le temps d'exécution des évènements de calcul peut varier de même que le temps des évènements de communication. Ainsi, les tâches de l'application peuvent être assignées à des blocs IP, qui peuvent eux-mêmes être assignés à un ensemble de bus partagés. Chaque bloc IP est caractérisé par l'ensemble des tâches qu'il peut exécuter et le temps d'exécution qu'il prend pour chacune des tâches. Chaque bus est caractérisé par sa largeur de bits et sa fréquence. La trace est extraite en instrumentant une simulation SystemC fonctionnelle de l'application et le temps d'exécution pour une architecture donnée est estimé en simulant l'ordonnancement de la trace pour cette architecture. Cette estimation tient compte de l'arbitrage des bus et de l'ordonnancement des tâches sur chaque bloc matériel, en supposant que ceux-ci utilisent un ordonnancement à priorités statiques. Contrairement à notre méthode, cette méthode d'estimation ne tient pas compte du temps pris par l'arbitrage et l'ordonnancement eux-mêmes et suppose que toutes les opérations de calcul d'une même tâche ont exactement le même temps d'exécution. De plus, elle n'automatise pas la caractérisation du temps d'exécution des tâches sur les blocs IP alors que notre méthode automatise la caractérisation du temps d'exécution logiciel et matériel de chaque tâche.

Dans (Isshiki *et al.*, 2009), les structures de contrôle (`if`, `for`, `while`) d'un code source C séquentiel sont instrumentées afin d'extraire, à chaque évaluation de celles-ci, la valeur `false` (0) ou `true` (1) de la condition de la structure de contrôle. L'exécution native de ce code C avec un stimulus donné permet d'extraire une trace sous la forme d'un train de bits de branchement, qui correspondent aux valeurs des conditions dans la série de structures de contrôle visitées. À partir du CFG du code C, il est donc possible de reconstituer la série de blocs de base visités en utilisant chaque valeur du train de bits pour choisir le chemin `false` ou `true` à chaque branchement dans le CFG. Cela permet également de déterminer le nombre d'exécutions de chaque bloc de base. Le temps d'exécution de chaque bloc de base est extrait par une analyse statique du code assembleur obtenu suite à une compilation croisée vers le processeur cible. Cette méthode suppose donc que la compilation croisée ne modifie pas le CFG du code. Le programme peut être divisé en plusieurs tâches. Dans ce cas, le CFG est partitionné en plusieurs CFG, chacun correspondant à une tâche, et des noeuds de synchronisation sont ajoutés aux CFG pour représenter les dépendances de données. Le train de bits est également partitionné entre les tâches. Une réduction remplace, dans chaque CFG, chaque sous-graphe correspondant à un segment de programme par un nouveau noeud

dont le temps d'exécution est égal au temps d'exécution moyen du segment. En déroulant séquentiellement chaque CFG selon le train de bits réduit qui lui est associé, on obtient donc une trace d'exécution composée d'évènements de calcul et de communication pour plusieurs tâches. Les tâches peuvent être assignées à un ensemble de processeurs homogènes eux-mêmes connectés par un ensemble de bus et le temps d'exécution de cette architecture est estimé en simulant l'ordonnancement de ces tâches pour l'architecture donnée. Cette simulation suppose un ordonnancement dynamique non préemptif des tâches sur les processeurs. Cette estimation ne supporte pas les accélérateurs matériels et ne tient pas compte de l'implication du processeur dans les communications des tâches.

Notre méthode extrait une trace d'évènements en réalisant un profilage au niveau système d'une simulation SystemC de l'application avec SPACE (Bois *et al.*, 2010). Pour chaque module de l'application, le temps d'exécution logiciel de chaque évènement de calcul, qui correspond à une exécution d'un segment de programme, est caractérisé par une exécution du code assembleur cible sur un ISS. Cela permet de tenir compte des effets de la compilation croisée sur le CFG et des branchements effectués à l'intérieur de chaque exécution de chaque segment. Notre méthode automatise la caractérisation des temps d'exécution matériel des différents évènements de calcul et simule l'ordonnancement de ces évènements en tenant notamment compte de l'arbitrage, de l'ordonnancement dynamique préemptif des tâches sur les processeurs par le RTOS ainsi que de l'implication des processeurs dans les communications.

Les travaux antérieurs utilisent la sémantique opérationnelle des KPN pour les évènements de communication capturés par la trace : il est supposé que toutes les lectures sont bloquantes. Cela permet d'assurer que, pour un stimulus donné, toutes les exécutions de l'application produiront exactement la même trace fonctionnelle d'évènements peu importe l'ordonnancement des tâches. Cela permet notamment d'assurer que la trace extraite de la simulation fonctionnelle et la trace obtenue pour une implémentation donnée contiennent les mêmes évènements de calcul et de communication et que seuls leurs temps diffèrent. Cela assure la validité de l'estimation du temps d'exécution par analyse de trace. Notre méthode étend ces travaux en permettant aux tâches d'utiliser un modèle de calcul plus général que les KPN et une sémantique opérationnelle qui inclut des lectures non-bloquantes. Cela fait en sorte que la trace produite peut dépendre de l'ordonnancement des tâches, mais la validité de l'estimation par trace est préservée en vérifiant si l'ordonnancement simulé de la trace pour une implémentation donnée correspond à un ordonnancement fonctionnellement équivalent de la trace initialement extraite.

2.5 Algorithmes d'exploration architecturale

Un algorithme d'exploration architecturale vise à trouver, parmi un ensemble d'ensemble d'architectures qui forment un espace de recherche, une architecture optimale pour une application embarquée selon une fonction objective qui évalue les solutions avec différentes métriques. Les différentes méthodes d'exploration architecturale se distinguent non seulement par leur algorithme, mais également par leur espace de recherche, leur fonction objective et les applications auxquelles elles s'appliquent.

L'exploration architecturale dont il est question ici concerne la macro-architecture et non la micro-architecture. Ainsi, l'espace de recherche de ces algorithmes peut considérer, pour un ensemble de tâches, les possibilités de partitionnement logiciel/matériel, d'allocation de processeurs et d'accélérateurs matériels, d'assignation des tâches aux processeurs et aux accélérateurs matériels, d'allocation de bus et de liens point-à-point et d'assignation des composants matériels à l'architecture de communication.

La fonction objective considère généralement des métriques de temps d'exécution et de quantité de ressources matérielles et peut également considérer d'autres métriques, par exemple la consommation de puissance. Une métrique est à optimiser alors que les autres métriques doivent respecter un ensemble de contraintes. La valeur de la fonction objective est typiquement égale à la valeur de la métrique à optimiser pour les solutions qui respectent toutes les contraintes. La fonction objective impose une pénalité, qui peut être finie ou infinie, aux solutions qui ne respectent pas toutes les contraintes (Lopez-Vallejo *et al.*, 2000). Ainsi, on peut chercher à minimiser le temps d'exécution tout en respectant des contraintes de ressources matérielles, ou à minimiser celles-ci tout en respectant une contrainte de temps d'exécution. Les métriques de quantité de ressources matérielles sont généralement évaluées par des méthodes simplement additives : la quantité de ressources matérielles d'une architecture est la somme des quantités de ressources matérielles de ses composants. Sauf dans les cas les plus simples, le temps d'exécution ne peut pas être évalué par une méthode additive et il faut avoir recours aux méthodes d'estimation présentées à la section 2.4, qui vont de l'ordonnancement statique d'un TPG à la simulation au niveau TLM.

Les problèmes d'exploration architecturale considérant plusieurs métriques sont généralement fortement NP-difficiles même dans les cas les plus simples (Arato *et al.*, 2005). Des algorithmes exacts, tels que la résolution par ILP ou un algorithme de séparation et évaluation (*branch and bound*) (Nemhauser et Wolsey, 1988), peuvent être utilisés, mais leur temps d'exécution croît de manière exponentielle avec la taille du problème, qui se compte typiquement en nombre de tâches. On préfère généralement utiliser des algorithmes heuristiques qui donnent des solutions non exactes mais quasi-optimales dans un temps raisonnable.

Une heuristique peut être créée sur mesure pour un problème d'exploration architecturale ou une méta-heuristique générique peut être spécialisée pour s'appliquer à ce problème. Une méta-heuristique de recherche locale raffine itérativement une solution. Différentes stratégies permettent d'éviter que la solution reste bloquée dans un optimum local qui n'est pas un optimum global. Ainsi, le recuit simulé (van Laarhoven et Aarts, 1987) accepte de manière stochastique des mouvements qui dégradent la solution alors que la recherche tabou (Glover et Laguna, 1997) utilise une mémoire à court terme et une mémoire à long terme pour interdire ou encourager certains mouvements. D'autres méta-heuristiques explorent l'espace de recherche à l'aide d'une population de solutions. Ces méta-heuristiques sont inspirées par l'évolution naturelle, tels que les algorithmes génétiques (Goldberg, 1989), ou par le comportement social des animaux, tels que l'optimisation par colonie de fourmis (Dorigo *et al.*, 1996) ou par essaims particulaires (Kennedy et Eberhart, 1995).

Certains problèmes d'exploration architecturale tentent d'optimiser plusieurs métriques à la fois et sont donc des problèmes multi-objectifs. La solution d'un tel problème n'est pas une seule solution optimale, mais plutôt un ensemble de solutions qui sont optimales au sens de Pareto. Une solution est Pareto-optimale si aucune autre solution n'est meilleure qu'elle pour chacune des métriques (Zitzler *et al.*, 2003).

2.5.1 Partitionnement logiciel/matériel avec un processeur

Un algorithme de partitionnement logiciel/matériel détermine, parmi un ensemble de tâches, lesquelles implémenter en logiciel et lesquelles implémenter en matériel. Une formulation de ce problème cible une architecture composée d'un seul processeur et d'un co-processeur s'exécutant à tour de rôle. L'application est spécifiée sous la forme d'un TIG en associant à chaque noeud un temps d'exécution logiciel et une quantité de ressources matérielles et en associant à chaque arc un temps de communication. Le temps d'exécution d'un partitionnement logiciel/matériel donné est évalué en additionnant le temps d'exécution des tâches assignées au logiciel et le temps de communication des arcs qui relient une tâche logicielle et une tâche matérielle. La quantité de ressources matérielles est également évaluée par une sommation sur les tâches assignées en matériel (Arato *et al.*, 2005).

Une formulation ILP (Arato *et al.*, 2003) ou un algorithme par séparation et évaluation (Mann *et al.*, 2007a; Jigang *et al.*, 2009) permettent d'obtenir une solution exacte à ce problème. Des heuristiques basées sur les algorithmes génétiques (Arato *et al.*, 2003) et sur l'algorithme de partitionnement de graphe de Kernighan-Lin (Mann *et al.*, 2007b) ont également été proposées. Dans (Arato *et al.*, 2005), ce problème est montré équivalent à un ensemble de problèmes de flot maximum paramétrés par deux nombres réels. L'algorithme de (Arato *et al.*, 2005) consiste donc à échantillonner l'espace 2D formé par les valeurs de

ces deux paramètres et à résoudre le problème de flot maximum correspondant à chaque point échantillonné. De manière similaire, (Jigang *et al.*, 2010) démontre que ce problème est équivalent à un ensemble de problèmes du sac à dos paramétré par un nombre réel et un algorithme est défini pour échantillonner l'espace 1D formé par les valeurs de ce paramètre.

Une autre formulation du problème de partitionnement logiciel/matériel cible une architecture composée d'un seul processeur et d'un ou plusieurs co-processeurs matériels tout en permettant une exécution parallèle de ces composants, qui sont connectés par un bus commun. L'application est alors exprimée sous la forme d'un TPG et un temps d'exécution logiciel ou matériel est associé à chaque du noeud du TPG selon le partitionnement à évaluer. Un temps de communication est également associé à chaque arc selon ce partitionnement. La fonction objective intègre une heuristique d'ordonnancement statique pour déterminer le temps d'exécution d'un partitionnement donné selon les relations de précédence entre les tâches et selon l'accès aux ressources partagées que sont le bus et le processeur. Les ressources matérielles d'une implémentation sont estimées par sommation, en associant un coût matériel à chaque tâche assignée en matériel (Wiangtong *et al.*, 2002). Un coût matériel en termes de mémoire requise peut également être associé aux tâches assignées en logiciel.

Plusieurs heuristiques ont été appliquées à ce problème, soit une heuristique gloutonne (Jigang *et al.*, 2008), le recuit simulé (Lopez-Vallejo *et al.*, 2000; Wiangtong *et al.*, 2002), les algorithmes génétiques (Wiangtong *et al.*, 2002; Resano *et al.*, 2003), la recherche tabou (Wiangtong *et al.*, 2002) et l'optimisation par colonie de fourmis (Wang *et al.*, 2003). Des extensions à ce problème ont été proposées pour choisir parmi plus d'une implémentation logicielle ou matérielle d'une tâche donnée (Kalavade et Lee, 1997), intégrer un ordonnancement pipeliné de plusieurs itérations du TPG (Chatha et Vemuri, 2002) ou intégrer l'ordonnancement statique d'un graphe SDF (Knerr *et al.*, 2008). Les principales limites de ces algorithmes sont que leur architecture cible n'a qu'un seul processeur et un seul bus et qu'ils supposent que l'application peut être ordonnancée statiquement.

2.5.2 Exploration architecturale avec plusieurs processeurs

Les algorithmes présentés dans cette section assignent les tâches de l'application à une architecture qui comprend plusieurs processeurs et possiblement des accélérateurs matériels. On distingue trois catégories d'algorithmes. Dans la première, le nombre de processeurs et la topologie de communication sont fixes. La deuxième catégorie décide du nombre de processeurs à allouer, mais n'explore pas la topologie de communication. La troisième catégorie explore à la fois l'allocation des processeurs et la topologie de communication. On désigne les algorithmes des deux dernières catégories comme des algorithmes de synthèse d'architecture, étant donné qu'ils n'assignent pas seulement des tâches sur une architecture donnée, mais

qu'ils décident également de cette architecture. Notre méthode présentée au chapitre 8 se classe dans la troisième catégorie.

2.5.2.1 Nombre fixe de processeurs

Plusieurs algorithmes ont été proposés pour minimiser le temps d'exécution d'un ordonnancement statique d'un TPG sur un nombre fixe de processeurs (Kwok et Ahmad, 1999). Dans le contexte des systèmes embarqués, on s'intéresse également à l'impact de l'assignation et de l'ordonnancement des tâches sur les ressources matérielles requises. Ainsi, (Orsila *et al.*, 2007) présente un algorithme de recuit simulé et une adaptation de l'heuristique de Kernighan-Lin (Kernighan et Lin, 1970) pour optimiser le temps d'exécution et la mémoire requise lors de l'assignation des tâches d'un TPG à une architecture composée d'un nombre fixe de processeurs homogènes et d'un bus partagé. Ces algorithmes, qui sont utilisés dans le cadre de la méthodologie de conception de systèmes embarqués Koski (Kangas *et al.*, 2006), évaluent les solutions proposées par un ordonnancement statique du TPG sur les processeurs et le bus partagé. Il n'est cependant pas possible d'assigner une tâche en matériel.

D'autres algorithmes réalisent également un partitionnement logiciel/matériel : chaque tâche est assignée soit à un des processeurs de l'architecture, soit en tant qu'accélérateur matériel. Ainsi, (Wang *et al.*, 2006) présente un algorithme hybride pour l'ordonnancement statique et l'assignation des tâches d'un TPG sur une architecture composée d'un processeur PowerPC (IBM Microelectronics Division, 1998), d'un processeur DSP TMS320C25 (Texas Instruments Inc., 2010) et d'un FPGA. Cet algorithme commence par produire un ensemble de solutions via une optimisation par colonie de fourmis, puis raffine chacune d'entre elles par un recuit simulé, selon des critères de temps d'exécution et de coût matériel. Dans (Niemann et Marwedel, 1997), une formulation ILP est présentée pour le problème de l'ordonnancement statique et de l'assignation des tâches d'un TPG sur un ensemble fixe de processeurs, d'ASIC et de bus. Cet algorithme permet de trouver une solution optimale, mais sa complexité exponentielle rend difficile son application à une architecture autre que celle utilisée pour un partitionnement logiciel/matériel à un processeur. SoCDAL (Ahn *et al.*, 2008) utilise un algorithme évolutionniste d'inspiration quantique (Han et Kim, 2002) pour l'assignation des processus d'un graphe CSDF à une architecture composée d'un nombre fixe de processeurs ARM7, de DSP CEVA Teak (CEVA, Inc., 2004), d'accélérateurs matériels et d'un bus partagé. Dans (Wild *et al.*, 2003), un recuit simulé ou une recherche tabou est utilisée pour assigner un TPG sur une architecture similaire. Ces algorithmes ont en commun qu'ils supposent qu'un ordonnancement statique de l'application est possible et qu'ils négligent l'implication du processeur dans les communications (Sinnen *et al.*, 2006).

2.5.2.2 Exploration de l'allocation des processeurs

SOS (Prakash et Parker, 1992) est une formulation ILP d'un problème de synthèse d'architecture multi-processeurs hétérogène pour une application spécifiée sous la forme d'un TPG. Un gabarit d'architecture spécifie le nombre maximal de chaque type de processeurs et l'algorithme alloue un nombre variable de ces processeurs ayant chacun une mémoire locale et un co-processeur pour la gestion des communications. Les tâches du TPG sont assignées et ordonnancées statiquement sur l'architecture allouée. Un temps d'exécution est assigné à chaque tâche du TPG selon le processeur sur laquelle elle est assignée et un temps de communication est également associé aux communications entre processeurs. Un coût matériel est associé à chaque processeur alloué de même qu'aux liens de communication alloués entre les processeurs.

Bien que cet algorithme alloue également des liens de communications entre les processeurs, il ne réalise pas une exploration de la topologie de communication. En effet, pour une allocation de processeurs donnée et pour une assignation donnée des tâches aux processeurs, l'algorithme considère qu'il existe une et une seule topologie de communication : celle obtenue en allouant un lien de communication point à point entre une paire de processeurs si et seulement si une tâche assignée au premier processeur communique avec une tâche assignée au deuxième processeur.

Plusieurs heuristiques ont été proposées pour accélérer la résolution de ce problème, soit les algorithmes génétiques (Dhodhi *et al.*, 1995), les algorithmes à évolution différentielle (Rae et Parameswaran, 1998) et une heuristique de descente (Wolf, 1997). Dans (Prakash et Parker, 1994), une formulation ILP est proposée pour une variante de ce problème dont la topologie de communication est un bus partagé et qui tient compte de la quantité de mémoire utilisée par le code des processeurs alloués.

Dans (Erbas *et al.*, 2006), des algorithmes génétiques multi-objectifs sont utilisés pour optimiser le temps d'exécution, la puissance et le coût matériel d'une architecture multi-processeurs implémentant un KPN. Les processus du KPN sont caractérisés par leur charge de calcul alors que les canaux sont caractérisés par leur charge de communication. Les processus sont assignés aux processeurs alors que les canaux qui réalisent des communications inter-processeurs sont assignés à des mémoires partagées. Le temps d'exécution est évalué en additionnant les charges imposées aux processeurs par les différents éléments du KPN sans tenir compte de l'ordonnancement dynamique des processus (dépendances de données, changements de contexte, etc.) L'algorithme prend en entrée un graphe d'architecture qui indique quels processeurs peuvent être alloués et quels processeurs peuvent communiquer entre eux. Le coût matériel est calculé selon les processeurs alloués à l'intérieur de ce gabarit. Un autre algorithme génétique multi-objectifs basé sur un gabarit d'architecture est proposé

dans (Schlichter *et al.*, 2006), qui assigne les tâches d'un TPG à un graphe d'architecture et évalue leur temps d'exécution par un ordonnancement statique. Une variante de cet algorithme est utilisé par SystemCoDesigner (Keinert *et al.*, 2009), alors que le temps d'exécution de l'application, qui est constituée d'un ensemble d'acteurs SysteMoC (Falk *et al.*, 2006), est plutôt évalué par une simulation TLM avec VPC (Streubuhr *et al.*, 2009).

Ces algorithmes limitent le nombre de processeurs qui peuvent être alloués à moins que le gabarit d'architecture contienne au moins autant de processeurs que de tâches. Cela est problématiques pour les formulations ILP étant donné que le nombre de variables et d'équations y est proportionnel à la fois au nombre de tâches et processeurs (Prakash et Parker, 1992, 1994). Aussi, ces algorithmes ne supportent pas directement l'assignation de modules en matériel. Pour ce faire, il faut plutôt procéder indirectement, en définissant pour chaque module un « processeur » spécial représentant l'implémentation matérielle du module et en spécifiant que seul ce module peut être assigné à ce processeur. Cela est nécessaire pour s'assurer que ces modules matériels puissent être ordonnancés en parallèle. Cependant, cela augmente grandement le nombre de processeurs dans le gabarit d'architecture et le risque qu'une assignation donnée des tâches soit invalide. Cela est particulièrement problématique pour les algorithmes génétiques présentés. En effet, les individus (qui représentent chacun une allocation et assignation données) y sont représentés sous la forme d'un vecteur de bits sur lesquels s'appliquent des opérateurs de croisement et de mutation. Ces opérateurs sont aveugles (ils ne tiennent pas compte des spécificités du problème) et leur application peut donc produire une assignation invalide (Erbas *et al.*, 2006; Schlichter *et al.*, 2006). Il est alors nécessaire d'appliquer une procédure de réparation sur un tel individu, et ce possiblement sur plusieurs individus et sur plusieurs générations (ou itérations) de l'algorithme. Cette réparation peut être complexe : dans (Schlichter *et al.*, 2006; Keinert *et al.*, 2009), elle implique de résoudre un problème NP-complet de satisfiabilité booléenne (Garey et Johnson, 1979) dont le nombre de clauses est proportionnel au nombre de processeurs.

À l'inverse, notre formulation du problème n'impose aucune borne explicite au nombre de processeurs et supporte directement l'assignation des modules au matériel. Les opérateurs (mouvements) définis pour la recherche locale tiennent compte de la spécificité du problème et assurent que toutes les assignations produites sont correctes par construction. Il est ensuite aisé d'ajouter une borne explicite, si nécessaire, au nombre de processeurs : il suffit d'interdire les mouvements qui feraient dépasser cette borne. Nous avons implémenté une telle borne, qui est optionnelle, pour tous nos algorithmes. Les mêmes principes s'appliquent directement à l'exploration des topologies de communication sur plusieurs bus.

2.5.2.3 Allocation des processeurs et topologie de communications

Divers algorithmes combinent l'exploration de l'allocation des processeurs et de la topologie de communication. Ainsi, (Le Beux *et al.*, 2009) présente un algorithme génétique multi-objectifs pour assigner les tâches d'un TPG à un nombre variable de processeurs homogènes reliés entre eux par un réseau sur puce (NoC : Network on Chip). Cette exploration permet de choisir parmi trois topologies régulières pour le NoC, soit une topologie crossbar, maillée (*mesh*) 2D ou en anneau. Notre approche explore plutôt des topologies de bus hiérarchiques qui peuvent être irrégulières et relier des accélérateurs matériels en plus des processeurs.

(Blickle *et al.*, 1998) assigne les noeuds et les arcs d'un TPG à un graphe d'architecture dont les noeuds représentent des processeurs ou des liens de communication et dont les arcs indiquent quels processeurs peuvent utiliser quels liens de communication. Le gabarit d'architecture peut contenir des liens point-à-point et des bus partagés, mais ne contient pas de ponts entre les bus. Cette méthode ne cible donc pas les topologies de bus hiérarchiques sur puce. Un algorithme génétique multi-objectifs est utilisé pour explorer les différentes solutions possibles, dont le temps d'exécution est évalué par un ordonnancement statique. Cette approche partage les limites des algorithmes génétiques présentés à la section précédente. En particulier, l'application des opérateurs génétiques peut rendre un individu invalide et sa réparation est un problème NP-complet.

L'algorithme génétique MOGAC (Dick et Jha, 1998) s'attaque à ce problème en définissant des grappes de solutions qui ont exactement la même allocation de processeurs et la même topologie de liens de communication. Les opérateurs génétiques sont principalement appliqués à l'intérieur d'une même grappe pour éviter de modifier l'allocation ou la topologie et de créer des individus structurellement incohérents. L'allocation et la topologie sont explorées lors de l'application occasionnelle des opérateurs génétique aux grappes elles-mêmes. Chaque individu voit alors ses informations d'assignation aléatoirement réinitialisées pour sa nouvelle allocation et topologie. L'exploration de l'allocation et de l'assignation se font ainsi en deux phases itératives alors qu'elle se fait en une phase combinée dans notre méthode. Dans (Deniziak et Gorski, 2008), la programmation génétique est appliquée à ce problème. Dans cet algorithme, un individu ne représente pas une allocation et assignation particulière, mais plutôt un algorithme glouton, composé à partir de règles élémentaires, qui permet de construire une telle allocation et assignation en une phase combinée. Les opérateurs génétiques modifient les règles utilisées par ces algorithmes gloutons, mais rien ne garantit que l'ensemble de ces règles permet de générer toutes les allocations, topologies et assignations qui font partie de l'espace de recherche. Ces méthodes considèrent également les topologies composées de liens point-à-point et de bus partagés, mais non les topologies de bus hiérar-

chiques.

(Madsen *et al.*, 2006) présente un algorithme génétique multi-objectifs pour l'exploration de l'allocation des processeurs et des topologies de bus hiérarchiques pour un TPG. Ce modèle d'architecture considère donc les ponts. De plus, les opérateurs génétiques utilisés ne sont pas aveugles et tiennent compte des spécificités du problème. Ainsi, l'opérateur de croisement s'assure de conserver la cohérence entre l'allocation et l'assignation. Les opérateurs de mutation permettent de modifier l'assignation d'une tâche, de même que d'ajouter ou de retirer un processeur tout en modifiant l'assignation des tâches en conséquence. Ces opérateurs de mutation sont semblables aux mouvements de recherche locale de notre méthode, mais celle-ci inclut également des opérateurs pour ajouter ou retirer des bus de même que pour déplacer des processeurs d'un bus à un autre. L'exploration de la topologie dans (Madsen *et al.*, 2006) doit donc se faire à l'intérieur d'un gabarit. Cette méthode modélise les accélérateurs matériels simplement comme un processeur avec une fréquence plus élevée alors que la nôtre considère comme distinctes les implémentations matérielles des différentes tâches. Notre méthode tient ainsi compte du fait que toutes les tâches ne subissent pas une accélération uniforme en passant du logiciel au matériel.

2.5.3 Algorithmes de recherche locale

Les principaux travaux portant sur l'exploration combinée de l'allocation des processeurs et de la topologie de communications utilisent des algorithmes évolutionnistes plutôt que des algorithmes de recherche locale tels que le recuit simulé (van Laarhoven et Aarts, 1987) ou la recherche tabou (Glover et Laguna, 1997), qui raffinent itérativement une solution tout en pouvant sortir d'un optimum local. Les résultats de travaux antérieurs portant sur un sous-ensemble de ce problème indiquent que la recherche tabou se compare avantageusement aux algorithmes génétiques pour l'exploration architecturale. Ainsi, la recherche tabou s'exécute plus rapidement et obtient de meilleures solutions qu'un algorithme génétique pour le partitionnement logiciel/matériel d'un TPG avec ordonnancement statique (Wiangtong *et al.*, 2002; Jigang *et al.*, 2008) ainsi que pour l'exploration de l'allocation des processeurs et de l'assignation des tâches à ceux-ci (Axelsson, 1997). Ces résultats nous ont incité à proposer ce qui est, à notre connaissance, la première heuristique de recherche tabou pour un problème combinant l'exploration de l'allocation des processeurs et d'une topologie de bus hiérarchiques. Le recuit simulé est concurrentiel avec les algorithmes génétiques pour l'exploration architecturale (Axelsson, 1997; Wiangtong *et al.*, 2002; Jigang *et al.*, 2008) et nous avons également implémenté cet algorithme qui, à notre connaissance, n'a pas non plus été proposé pour le problème combiné d'exploration architecturale. Les heuristiques proposées se distinguent également des algorithmes génétiques multi-objectifs présentés à la section 2.5.2.3

en ce qu'elles optimisent plutôt un critère donné tout en soumettant les autres critères à des contraintes.

Les heuristiques de recherche locale raffinent une solution en y appliquant itérativement des opérateurs qu'on appelle mouvements. Le voisinage d'une solution donnée est l'ensemble des solutions qu'on peut obtenir en appliquant un mouvement à cette solution. Le recuit simulé sélectionne aléatoirement un mouvement dans un voisinage et applique ce mouvement à la solution si il améliore la solution, ou sinon avec une probabilité qui dépend de la température courante. La valeur de la température subit généralement une décroissance géométrique lors du recuit simulé ($T(i+1) = a * T(i)$ avec $a < 1$) et le recuit simulé cesse lorsque la température devient trop basse pour échapper à un optimum local. Le recuit simulé a été appliqué aux problèmes du partitionnement logiciel/matériel (Eles *et al.*, 1997; Wiangtong *et al.*, 2002; Ahmed et Khan, 2004; Banerjee et Dutt, 2004), de l'assignation des tâches aux processeurs (Wild *et al.*, 2003; Orsila *et al.*, 2007) et de l'allocation des processeurs (Axelsson, 1997). Le temps d'exécution et la qualité des solutions trouvées par le recuit simulé dépendent de la température initiale ainsi que du taux de décroissance. De plus, les valeurs optimales de ces paramètres dépendent de l'instance du problème à résoudre. La valeur optimale du taux de décroissance peut également varier pendant un recuit simulé donné. Cependant, la plupart des travaux utilisent des valeurs fixes pour ces paramètres (Axelsson, 1997; Eles *et al.*, 1997; Wild *et al.*, 2003; Ahmed et Khan, 2004; Banerjee et Dutt, 2004). Une méthode qui équivaut à adapter la température initiale aux caractéristiques de l'instance du problème est présentée dans (Orsila *et al.*, 2007). Dans (Wiangtong *et al.*, 2002), la température est légèrement augmentée lorsqu'un mouvement est rejeté, ce qui permet un certain ajustement du taux de décroissance selon l'instance du problème. Notre recuit simulé combine ces deux améliorations tout en ajustant continuellement le taux de décroissance. Il s'agit à notre connaissance de la première application du recuit simulé adaptatif (Ortner, 2004; Perrin *et al.*, 2005) à l'exploration architecturale.

La recherche tabou est une heuristique locale qui, à chaque itération, applique le meilleur mouvement non tabou dans un voisinage. Le caractère tabou ou non d'un mouvement est déterminé selon les informations contenues dans une mémoire à court terme appelée liste tabou. Après un mouvement, les informations associées à ce mouvement sont ajoutées à la liste tabou, puis elle sont généralement retirées après un nombre d'itérations qui est égal à longueur de la liste tabou. La recherche tabou a été appliqué aux problèmes du partitionnement logiciel/matériel (Eles *et al.*, 1997; Wiangtong *et al.*, 2002; Ahmed et Khan, 2004), de l'assignation des tâches aux processeurs (Wild *et al.*, 2003) et de l'allocation des processeurs (Axelsson, 1997; Slomka *et al.*, 2004). L'efficacité de la recherche tabou dépend donc de la structure de la liste tabou et de sa longueur. En particulier, la longueur optimale de la liste

tabou dépend de l'instance du problème à résoudre et peut varier au cours d'une recherche tabou donnée. Par contre, ces travaux utilisent une liste tabou de longueur fixe (Axelsson, 1997; Eles *et al.*, 1997; Wiangtong *et al.*, 2002; Wild *et al.*, 2003; Slomka *et al.*, 2004). La seule exception est (Ahmed et Khan, 2004), qui fait subir une décroissance géométrique à la longueur de la liste tabou jusqu'à ce qu'elle atteigne une longueur fixe minimale. En comparaison, notre recherche tabou peut augmenter ou diminuer la longueur de la liste tabou selon l'évolution de la recherche. Ainsi, il s'agit à notre connaissance de la première application à un problème d'exploration architecturale de la recherche tabou réactive (Battiti et Tecchiolli, 1994, 1995), qui inclut également un mécanisme de diversification basé sur une mémoire à long terme qui détecte les répétitions. De plus, les autres travaux utilisent une liste tabou qui contient soit des solutions récemment visitées (Wiangtong *et al.*, 2002; Wild *et al.*, 2003; Ahmed et Khan, 2004; Slomka *et al.*, 2004) ou l'inverse des mouvements récemment effectués (Axelsson, 1997; Eles *et al.*, 1997). Notre liste tabou est plus puissante en ce qu'elle contient des attributs qui s'appliquent à des classes des mouvements et qu'elle traite séparément l'ajout et le retrait d'attributs par les mouvements.

2.6 Méthodologies intégrées

Le chapitre 3 de cette thèse présente une méthodologie intégrée de conception, d'exploration architecturale et de synthèse des systèmes embarqués. Alors que les sections précédentes ont comparé aux travaux antérieurs nos contributions aux différentes étapes de cette méthodologie, cette section compare notre méthodologie dans son ensemble à d'autres méthodologies intégrées de conception, d'exploration architecturale et de synthèse. Ces méthodologies sont SoCDAL (Ahn *et al.*, 2008), PeaCE (Lee *et al.*, 2010), Koski (Kangas *et al.*, 2006), Daedalus (Nikolov *et al.*, 2008c) et SystemCoDesigner (Keinert *et al.*, 2009).

2.6.1 SoCDAL

Dans SoCDAL (Ahn *et al.*, 2008), une application embarquée est spécifiée sous la forme d'un graphe CSDF et la fonctionnalité de chaque processus CSDF est spécifiée en SystemC. L'architecture cible comprend un nombre fixe de processeurs, des accélérateurs matériels et un bus partagé. SoCDAL assigne et ordonnance statiquement les processus CSDF sur cette architecture. Le temps d'exécution d'une assignation et ordonnancement donnés est donc évalué par analyse statique. Ainsi, pour chaque processus CSDF, un ensemble d'acteurs SDF correspondant aux modes d'exécution de ce processus est extrait. Le code C associé à chaque acteur SDF est également extrait, de même que son CFG. Le temps d'exécution de chaque bloc de base est estimé statiquement, puis le WCET logiciel d'un acteur SDF est estimé par

la résolution d'un problème ILP formulé selon les contraintes structurelles du CFG, tel que décrit à la section 2.4.1.2. Le WCET matériel d'un acteur SDF est estimé de manière analogue (Yoo *et al.*, 2006). Dans les deux cas, le processus d'estimation statique peut demander une intervention de l'utilisateur pour déterminer le nombre maximal d'itérations d'une boucle. Des coûts en ressources matérielles ou en mémoire logicielle sont aussi associés à chaque processus CSDF, mais il n'est pas clair si cela est réalisé automatiquement ou manuellement.

SoCDAL procède ensuite à l'exploration architecturale à l'aide d'un algorithme évolutionniste d'inspiration quantique, qui vise à optimiser l'assignation des processus CSDF aux processeurs et accélérateurs matériels de l'architecture. Le temps d'exécution d'une assignation donnée est évalué par un ordonnancement statique non-préemptif des processus du CSDF, qui tient également compte de la contention sur le bus partagé. Le coût d'une assignation est évalué selon la mémoire logicielle utilisée par les processus CSDF assignés en logiciel et les ressources matérielles utilisées par ceux assignés en matériel. SoCDAL peut ensuite générer, pour l'assignation et l'ordonnancement statique retenus par l'algorithme, un modèle BCA du système. Ainsi, un ISS ARM est associé à chaque processeur et le logiciel embarqué exécuté par cet ISS est construit à partir du code C des processus qui lui sont assignés. Ce logiciel embarqué inclut également un ordonnanceur statique non-préemptif. Un modèle TLM est généré pour chaque processus assigné en matériel et une simulation BCA du système peut être réalisée avec le simulateur MaxSim (ARM Ltd., 2010). Ainsi, cette méthode ne réalise pas une synthèse matérielle du système étant donné qu'un code RTL n'est pas généré pour les modules assignés en matériel.

Notre méthodologie se distingue de SoCDAL en ce qu'elle se base sur un modèle de calcul plus général, qui ne peut pas nécessairement être ordonnancé statiquement. En conséquence, l'estimation du temps d'exécution dans notre méthodologie se base plutôt sur des méthodes dynamiques, qui incluent une caractérisation par profilage non-intrusif. Notre exploration architecturale permet d'explorer non seulement l'assignation des tâches logicielles, mais également l'allocation des processeurs et la topologie de communication. De plus, nous utilisons des algorithmes de recherche locale plutôt que des algorithmes évolutionnistes. Finalement, notre méthode automatise la synthèse matérielle des modules assignés en matériels vers une implémentation RTL, ce qui permet d'implémenter l'architecture retenue sur une cible FPGA.

2.6.2 PeaCE

Une application embarquée est modélisée dans PeaCE comme un ensemble de tâches périodiques ayant chacune une échéance (*deadline*) à respecter et une priorité statique (Lee *et al.*, 2010). Chaque tâche est un graphe SDF et la fonctionnalité de chaque processus SDF est spécifiée en C. L'architecture cible comprend un nombre variable de processeurs et

d'accélérateurs matériels de même qu'une topologie de bus hiérarchiques. L'exploration architecturale se fait en deux phases. La première phase alloue les processeurs et les accélérateurs matériels et assigne à ceux-ci les processus SDF de manière à minimiser le coût matériel tout en s'assurant que chaque tâche respecte son échéance. La seconde phase choisit une topologie de communication de manière à minimiser le temps d'exécution des tâches.

PeaCE associe un temps d'exécution logiciel à chaque processus SDF et sa caractérisation s'effectue à l'aide d'une instrumentation intrusive du code source du processus, puis d'un profilage sur l'ISS cible (Kwon *et al.*, 2004). S'il existe un accélérateur matériel pour un processus SDF donné, alors un temps d'exécution matériel peut lui être associé, mais cette caractérisation est effectuée manuellement. Le coût matériel d'une allocation donnée de processeurs et d'accélérateurs matériels est évalué en additionnant le coût matériel de chacun d'entre eux. Pour une assignation donnée des processus SDF à une telle allocation, PeaCE évalue le respect des échéances des tâches selon un ordonnancement statique de tous les processus SDF pour une hyperpériode des tâches. Étant donné que la topologie de communication n'est pas encore fixée, les temps de communication y sont modélisés de manière abstraite selon un coefficient qui représente un temps de communication unitaire. Une heuristique gloutonne alloue des processeurs ou des accélérateurs matériels jusqu'à ce que les échéances de chaque tâche soient respectées (Oh et Ha, 2002). Pour une allocation donnée, une heuristique d'ordonnancement par liste assigne les processus SDF de chaque tâche aux composants alloués. Cette assignation se fait par ordre de priorité des tâches et l'heuristique d'ordonnancement ajuste les temps d'exécution des processus selon le taux d'utilisation des processeurs par les tâches de priorités plus élevées.

Une fois que l'allocation est fixée de même que l'assignation des processus, PeaCE peut générer, à partir du code C des processus, un code logiciel embarqué pour chaque processeur alloué. Par contre, il n'y a pas de synthèse comportementale des processus assignés en matériel et un module VHDL doit être fourni pour chacun d'entre eux. PeaCE génère alors un contrôleur pour l'intégration de ces modules VHDL (Jung *et al.*, 2008). Ensuite, une cosimulation logiciel/matériel avec des ISS et un simulateur HDL est réalisée pour extraire une trace des accès aux mémoires. Cette trace inclut les communications entre processeurs, qui s'effectuent par mémoire partagée, et une topologie de communication donnée peut être évaluée en ordonnançant sur celle-ci la trace. Une heuristique de descente explore la topologie de communication (Kim et Ha, 2006). Cette heuristique débute par une topologie à un seul bus partagé, puis tente d'ajouter des bus ou de modifier l'assignation des composants aux bus jusqu'à ce qu'aucun mouvement ne permette de diminuer le temps d'exécution du système. Il est cependant possible que les tâches ne respectent plus leurs échéances une fois que la topologie de communication est prise en compte. Dans ce cas, la phase d'allocation des

processeurs et d'assignation des tâches est lancée à nouveau avec un coefficient de temps de communication unitaire plus élevé. Une nouvelle topologie est alors choisie et ainsi de suite jusqu'à ce qu'on converge vers une solution. Dans l'exemple fourni dans (Lee *et al.*, 2010), il a été nécessaire d'itérer 27 fois entre ces deux phases avant de converger, ce qui inclut 27 co-simulations logiciel/matériel.

Notre méthodologie se distingue de PeaCE par l'utilisation d'un modèle de calcul moins restrictif que les SDF, par l'automatisation de la synthèse comportementale des modules assignés en matériel, par l'utilisation d'un profilage non-intrusif, par l'automatisation de la caractérisation des temps d'exécution des modules matériels et du RTOS ainsi que par l'utilisation d'heuristiques, telles que le recuit simulé ou la recherche tabou, qui peuvent sortir d'un optimum local. De plus, notre formulation du problème d'exploration architecturale combine l'exploration de l'allocation des processeurs et de la topologie de communication en une seule et même phase, plutôt que d'itérer entre deux phases distinctes.

2.6.3 Koski

Une application embarquée est spécifiée dans Koski (Kangas *et al.*, 2006) en UML suivant le modèle de calcul des KPN. L'architecture cible comprend un nombre variables de processeurs homogènes NIOS II et un bus partagé. L'exploration architecturale consiste donc à choisir le nombre de processeurs à allouer et à assigner les tâches aux processeurs. Pour une allocation et une assignation données, Koski peut générer soit un modèle TLM de l'architecture, soit une implémentation RTL de l'architecture incluant un code C embarqué pour chaque processeur alloué. Koski caractérise le temps d'exécution logiciel moyen de chaque processus par un profilage intrusif sur le processeur cible.

Le réseau KPN de l'application est ensuite statiquement converti en un TPG aux fins de l'exploration architecturale, mais les détails de cette conversion ne sont pas fournis. Le temps d'exécution d'une allocation et assignation données sont évalué par un ordonnancement statique du TPG. L'exploration architecturale alloue successivement de à 1 à N processeurs et applique pour chaque allocation une heuristique d'assignation hybride qui combine le recuit simulé et l'heuristique de Kernighan-Lin (Orsila *et al.*, 2005). Le recuit simulé est d'abord appliqué, puis l'heuristique de Kernighan-Lin est appliquée au résultat du recuit simulé avec des mouvements qui consistent à déplacer une tâche d'un processeur à un autre. Le recuit simulé et Kernighan-Lin sont ensuite appliqués de nouveau en diminuant de moitié la température initiale du recuit et ainsi de suite jusqu'à la terminaison de l'heuristique. Cette heuristique optimise le temps d'exécution et la taille des tampons de mémoire (Orsila *et al.*, 2007).

Un désavantage de l'utilisation d'un TPG pour l'exploration architecturale est que l'es-

timation du temps d'exécution du système ne tient pas compte du temps d'exécution du RTOS et des changements de contexte, alors qu'un RTOS avec ordonnancement dynamique sera utilisé pour l'implémentation du KPN. Une deuxième phase d'exploration architecturale raffine donc la solution obtenue et l'évaluation du temps d'exécution se fait alors en simulant l'application sur l'architecture cible. Cette deuxième phase est réalisée par une heuristique de descente, qui tente de déplacer une tâche à la fois d'un processeur à un autre jusqu'à ce qu'aucun mouvement ne puisse améliorer la solution. Les paramètres du bus, tel que sa largeur de bits et les priorités des maîtres, peuvent ensuite être configurés par une heuristique inspirée du recuit simulé (Riihimaki *et al.*, 2002).

Notre méthodologie se distingue de Koski par l'utilisation d'un modèle de calcul plus général, par l'automatisation de la synthèse comportementale des modules matériels et par l'intégration de ceux-ci à l'exploration architecturale, par l'utilisation d'un profilage non-intrusif pour la caractérisation automatique des modules logiciels et matériels ainsi que du RTOS, par l'utilisation d'une analyse dynamique par trace tout au long de l'exploration architecturale et par l'utilisation de la recherche tabou. De plus, notre formulation du problème d'exploration architecturale considère également la topologie de communication et ne sépare pas l'allocation et l'assignation en deux phases.

2.6.4 Daedalus

Une application embarquée est représentée dans Daedalus (Nikolov *et al.*, 2008c) comme un KPN où un code séquentiel C++ est associé à chaque processus. L'architecture cible comprend un nombre variable de processeurs et peut également contenir des accélérateurs matériels. L'exploration architecturale choisit les processeurs et les accélérateurs matériels alloués et assigne les tâches à ceux-ci. Pour une allocation et une assignation donnée, il est possible de générer un modèle TLM temporisé de l'architecture et de simuler son exécution de l'application via Sesame (Pimentel *et al.*, 2006), tel que décrit à la section 2.4.2.2. Une implémentation RTL de l'architecture peut aussi être générée avec ESPAM (Nikolov *et al.*, 2008b), qui génère le logiciel embarqué associé à chaque processeur ainsi qu'une plate-forme RTL à partir des modèles RTL des processeurs, bus et mémoires. Un accélérateur matériel pour un module de l'application peut être utilisé seulement s'il existe déjà un IP RTL pour celui-ci. En effet, Daedalus n'automatise pas la synthèse comportementale des accélérateurs matériels, mais permet l'intégration d'un IP RTL par la génération d'un contrôleur (Nikolov *et al.*, 2008a).

L'exploration architecturale s'effectue principalement à l'aide d'un algorithme génétique multi-objectifs qui vise à optimiser le temps d'exécution et le coût matériel du système (Erbas *et al.*, 2006). Un gabarit d'architecture (tel qu'un ensemble de processeurs connectés par un

bus partagé ou un crossbar) doit être fourni à l'algorithme, qui cherche à trouver un ensemble Pareto-optimal d'allocations de processeurs et d'assignation des tâches. Le coût matériel est évalué selon l'ensemble des composants alloués alors que le temps d'exécution est évalué à partir d'un paramètre de charge de calcul associé à chaque processus du KPN et d'un paramètre de charge de communication associé à chaque canal du KPN. Ces paramètres peuvent être caractérisés par une analyse statique d'une trace de simulation du KPN tel que présenté à la section 2.4.3.1, mais il n'est pas clair à quel point cette caractérisation doit être faite manuellement ou automatiquement.

L'estimation du temps d'exécution lors de cette exploration architecturale est très grossière, car elle ne tient pas compte de l'ordonnancement des processus. Ainsi, Daedalus réalise une deuxième phase d'exploration architecturale en effectuant avec Sesame une simulation TLM temporisée des solutions prometteuses trouvées à l'étape précédente, ce qui permet de tenir compte de l'ordonnancement pour ces solutions. Les solutions qui demeurent Pareto-optimales après ces simulations seront considérées pour une implémentation au niveau RTL. Avant d'effectuer ces simulations temporisées, il est nécessaire de caractériser le temps d'exécution logiciel des calculs effectués par les processus du KPN. Cette caractérisation implique une annotation manuelle des segments de programme dans le code source et assembleur ainsi qu'un profilage intrusif de chaque processus sur un ISS afin d'extraire le temps d'exécution moyen de chaque segment de programme (Pimentel *et al.*, 2008). La caractérisation du temps d'exécution des accélérateurs matériels est par contre réalisée manuellement.

Une restriction des implémentations RTL générées dans Daedalus est qu'il doit être possible d'ordonnancer statiquement l'ensemble des tâches assignées à un processeur donné. En effet, ESPAM ne supporte pas un RTOS lors de la génération du logiciel embarqué d'un processeur et ordonnance plutôt au moment de la compilation les tâches qui y ont été assignées (Nikolov *et al.*, 2008b). Or il est généralement impossible d'ordonnancer statiquement les processus d'un KPN (Buck, 1993). L'exploration architecturale ne semble ni vérifier si un tel ordonnancement statique sera possible sur chaque processeur, ni tenir compte du temps d'exécution d'un éventuel ordonnancement dynamique. Bref, cette méthodologie semble mieux appropriée à un sous-ensemble des KPN pouvant être ordonnancés statiquement, tel que les CSDF.

Notre méthodologie se distingue de Daedalus par l'utilisation d'un modèle de calcul plus général, par l'automatisation de la synthèse comportementale des modules matériels, par l'utilisation d'un profilage non-intrusif pour la caractérisation automatique des modules logiciels et matériels, par le support d'un RTOS lors de l'estimation et de la synthèse logicielle, par l'utilisation d'une analyse dynamique par trace tout au long de l'exploration architecturale et par l'utilisation des algorithmes de recherche locale. De plus, notre estimation du

temps d'exécution tient compte des variations dans le temps d'exécution de chaque segment de programme plutôt que de considérer qu'il est toujours égal à un temps moyen. Finalement, notre formulation du problème d'exploration architecturale explore également la topologie de communication.

2.6.5 SystemCoDesigner

Une application embarquée est spécifiée dans SystemCoDesigner (Keinert *et al.*, 2009) comme un ensemble d'acteurs SysteMoC (Falk *et al.*, 2006), qui est une librairie SystemC basée sur le modèle de calcul FunState (Strehl *et al.*, 2001). Chaque acteur est spécifié avec SysteMoC comme une machine à états finis qui exécute des actions (calculs ou communications) lors des transitions d'état. Un tel acteur correspond à un processus KPN s'il effectue seulement des lectures bloquantes, mais il est également possible qu'un acteur effectue des lectures non-bloquantes. L'architecture cible est composée d'un nombre variable de processeurs MicroBlaze et d'accélérateurs matériels reliés entre eux par un ensemble de FIFO. SystemCoDesigner permet de générer une implémentation RTL de chaque acteur SysteMoC par une synthèse comportementale avec Cynthesizer (Meredith, 2008). Il semble que la synthèse d'interface ne couvre pas la sérialisation des structures de données et que les acteurs SysteMoC ne puissent communiquer qu'en utilisant des types de données primitifs C/C++.

Pour une allocation et une assignation donnée, SystemCoDesigner peut générer une modèle TLM temporisé ou une implémentation RTL. La simulation TLM temporisée s'effectue avec le simulateur VPC (Streubuhr *et al.*, 2009), tel que décrit à la section 2.4.2.2. Des temps d'exécution logiciel et matériel moyens sont associés aux calculs de chaque action, qui correspond approximativement à un segment de programme. Cette simulation TLM ordonnance les acteurs sur des modèles de processeur, mais ne tient pas compte du temps d'exécution de l'ordonnancement, ce qui mène à d'importantes erreurs d'estimation (Keinert *et al.*, 2009; Streubuhr *et al.*, 2009) par rapport au temps d'exécution de l'implémentation RTL. Une implémentation est générée en instanciant les blocs RTL correspondant à l'ensemble des processeurs et accélérateurs matériels alloués de même que les FIFO RTL les reliant entre eux. Le code logiciel pour chaque processeur est généré en convertissant chaque acteur en une classe C++ et en générant un ordonnanceur dynamique. Lors de l'exécution, celui-ci interroge successivement chaque acteur afin de voir si une de ses transitions d'état est activée et l'exécute si c'est le cas.

La caractérisation de ces temps d'exécution logiciel pour la simulation avec VPC se fait à l'aide d'un profilage du PC sur FPGA, tel que décrit à la section 2.3.1.1. Cette méthode demande une implication de l'utilisateur pour extraire la plage d'adresse associée à chaque action, configurer le profileur FPGA avec ces adresses, exécuter le profilage puis en extraire

les résultats. La caractérisation des temps d'exécution matériel pour VPC est supposée être faite par l'utilisateur, de même que la caractérisation du coût matériel des composants matériels. Après cette caractérisation, l'exploration architecturale utilise une simulation avec VPC pour évaluer le temps d'exécution des solutions et procède par sommation pour évaluer leurs quantités de ressources matérielles. L'allocation des processeurs et des accélérateurs matériels ainsi que l'assignation des acteurs à ceux-ci sont explorées par un algorithme génétique multi-objectifs basé sur un gabarit d'architecture (Schlichter *et al.*, 2006). Il est à noter que le fait que le modèle de calcul permette des lectures non-bloquantes implique que deux simulations VPC avec deux allocations et assignations différentes n'exécuteront pas nécessairement la même fonctionnalité et que leur temps d'exécution ne sont donc pas nécessairement comparables. L'exploration architecturale de SystemCoDesigner les considère néanmoins comparables et ne vérifie pas si la même fonctionnalité a été exécutée dans les deux cas.

Notre méthodologie se distingue de SystemCoDesigner par la spécification de chaque module sous la forme d'un code séquentiel, par la sérialisation des structures de données lors de la synthèse comportementale, par l'automatisation de la caractérisation des modules logiciels et matériels ainsi que du RTOS à l'aide d'un profilage sur ISS, par l'utilisation d'une estimation par trace qui tient compte du temps d'exécution de l'ordonnancement et par l'utilisation des algorithmes de recherche locale. Aussi, notre modèle de calcul possède une sémantique dénotationnelle qui permet de vérifier lors de l'estimation si deux implémentations différentes exécutent la même fonctionnalité et si leurs temps d'exécution sont donc comparables. De plus, notre estimation du temps d'exécution tient compte des variations dans le temps d'exécution de chaque segment de programme plutôt que de considérer qu'il est toujours égal à un temps moyen. Finalement, notre formulation du problème d'exploration architecturale explore également la topologie de communication.

CHAPITRE 3

PRÉSENTATION DE LA MÉTHODOLOGIE

Ce chapitre contient une vue d'ensemble de la méthodologie de conception, d'exploration architecturale et de synthèse des systèmes embarqués présentée dans cette thèse, alors que les chapitres suivants (4 à 8) détaillent des aspects de cette méthodologie. Ce chapitre présente également la plate-forme virtuelle SPACE (Filion *et al.*, 2007; Bois *et al.*, 2010) qui a été utilisée pour implémenter cette méthodologie.

3.1 Aperçu général

Un aperçu général de la méthodologie proposée est présenté à la figure 3.1 alors que la figure 3.2 illustre les changements que subit une application donnée à différentes étapes de la méthodologie. Le point de départ est un modèle de calcul et une spécification exécutable parallèle d'un système embarqué sous la forme d'un ensemble de modules SystemC (IEEE, 2005) communiquant ensemble et avec des périphériques via des canaux de communications abstraits, tel qu'illustré à la figure 3.2(a) (où le *crossbar* est fonctionnellement équivalent à un ensemble de canaux FIFO). Ce système embarqué sera implémenté sur une technologie cible, pour laquelle la plate-forme virtuelle SPACE (Filion *et al.*, 2007; Bois *et al.*, 2010) fournit un ensemble de modèles transactionnels de bus, de processeurs, de mémoires et de périphériques.

Pour implémenter la spécification sur la plate-forme virtuelle, il est nécessaire de prendre des décisions quant à l'allocation des processeurs, bus et autres composants de la plate-forme et à l'assignation des modules et des périphériques de la spécification sur ces composants alloués. On désigne cette prise de décision comme la synthèse d'architecture et son résultat est l'architecture du système embarqué. Une telle architecture n'est donc pas directement exécutable, mais représente un devis pour l'implémentation du système embarqué, tel qu'illustré à la figure 3.2(b).

La synthèse du système représente la mise en oeuvre de ce devis afin d'obtenir, pour le système embarqué, une implémentation SystemC à bas niveau qui est exécutable et est approximativement précise au cycle près, tel qu'illustré à la figure 3.2(c). À partir du devis de l'architecture, la synthèse du système alloue donc un ensemble de modèles transactionnels de composants de la plate-forme virtuelle puis assigne les modules et périphériques de la spécification à ces composants. Plus précisément, si le devis indique qu'un module doit être

assigné à un processeur, alors la synthèse du système assigne à ce processeur l'implémentation logicielle de ce module ciblant ce type de processeur. De la même manière, pour les modules qui doivent être assignés directement sur un bus, la synthèse du système y assigne l'implémentation matérielle RTL de ces modules pour la technologie cible. La synthèse préalable des modules permet de construire cette librairie d'implémentations logicielles et matérielles des modules qui sont utilisées par la synthèse du système. Cette synthèse du système inclut aussi un raffinement des canaux de communications entre les modules vers des protocoles et des mécanismes de communication concrets. Elle génère également, pour chaque processeur, un logiciel embarqué comprenant un RTOS et les modules logiciels assignés au processeur. Le résultat de cette synthèse est ainsi une implémentation en SystemC, qui est composée de blocs RTL, de modèles transactionnels de bus et de périphériques ainsi que de logiciels embarqués exécutés presqu'au cycle près (*cycle-approximate*) par des simulateurs de jeu d'instructions (ISS).

Par la suite, la synthèse de plate-forme remplace les modèles transactionnels des composants de la plate-forme virtuelle par les blocs RTL correspondant à ces composants, tel qu'illustré à la figure 3.2(d). L'implémentation finale obtenue est alors complètement au niveau RTL et, à la suite d'une synthèse logique dans un flot de conception RTL bien établi, le système embarqué est réalisé sur sa cible finale, tel qu'un FPGA ou un ASIC.

La partie du flot présentée jusqu'à maintenant est linéaire : la spécification d'une application est progressivement raffinée et synthétisée jusqu'à ce qu'une implémentation RTL en soit obtenue. Cependant, un flot strictement linéaire ne répond pas aux questions suivantes : « Comment prendre les décisions inhérentes à la synthèse d'architecture ? » et « Comment vérifier que les bonnes décisions ont été prises ? » Pour ce faire, on introduit le profilage, la caractérisation, l'estimation et l'exploration architecturale.

Une première réponse à ces questions est d'effectuer une simulation avec profilage logiciel/ matériel et une synthèse logique de l'implémentation SystemC obtenue suite à la synthèse du système et d'ainsi recueillir des métriques sur la validité fonctionnelle, la performance et le coût matériel de cette implémentation. Un processus d'exploration architecturale compare les métriques mesurées aux objectifs de performance et de coût de l'application et, si l'implémentation évaluée est satisfaisante, l'exploration architecturale peut se terminer et cette implémentation devient alors l'implémentation finale. Dans le cas contraire, l'exploration architecturale modifie les décisions qui avaient été prises lors de la synthèse d'architecture, génère une nouvelle implémentation via une synthèse du système, puis l'évalue et ainsi de suite. L'exploration architecturale automatise donc un processus d'essais et erreurs pour la synthèse d'architecture et l'implémentation d'un système embarqué.

Un tel processus d'exploration architecturale fonctionne correctement, mais peut être trop

lent étant donné que le profilage et la synthèse logique d'une implémentation sont des opérations dont le temps se mesure en minutes, voire en heures, et que l'exploration architecturale peut avoir à évaluer des milliers d'implémentations. Pour raccourcir la boucle d'exploration architecturale et l'accélérer, on a recours à l'estimation, qui permet d'estimer les métriques de validité fonctionnelle, de performance et de coût matériel d'une architecture sans avoir à effectuer un profilage et une synthèse logique de chaque implémentation à évaluer. Cette méthode d'estimation se base sur la simulation et la synthèse logique totale ou partielle d'un nombre restreint d'implémentations initiales qui permettent de caractériser les paramètres de performance et de coût matériel de l'application et de la plate-forme. Après la caractérisation initiale, la méthode d'estimation accélère grandement l'évaluation des différentes architectures de l'application par l'exploration architecturale.

La vérification de l'application, de sa spécification exécutable ou de l'implémentation finale ne fait pas partie des présents travaux. On suppose ainsi que la spécification exécutable a déjà fait l'objet d'une vérification fonctionnelle, qui se trouve en amont de cette méthodologie, et qu'elle est correcte. Des méthodes existantes de vérification au niveau RTL peuvent également être appliquées sur l'implémentation finale en aval de cette méthodologie. On se sert néanmoins des propriétés de notre modèle de calcul pour vérifier l'équivalence fonctionnelle des ordonnancements réalisés par la spécification exécutable et par l'implémentation de l'application. Ce degré d'équivalence constitue une des métriques mesurées ou estimées dans la méthodologie.

3.2 Présentation des éléments de la méthodologie

Cette section présente plus en détails les éléments de la méthodologie introduite à la section 3.1 et indique quels éléments constituent, en plus de la méthodologie elle-même, les contributions de cette thèse, et lesquels sont des travaux antérieurs liés à la plate-forme virtuelle SPACE.

3.2.1 Modèle de calcul RTPN

Le modèle de calcul utilisé dans cette méthodologie sont les réseaux de processus temps-réel (RTPN), qui sont introduits pour la première fois au chapitre 4 de cette thèse. Les RTPN sont une extension des réseaux de processus Kahn (KPN) (Kahn, 1974). Ainsi, tel un KPN, un RTPN est un réseau de processus séquentiels et déterministes qui communiquent entre eux par un ensemble de canaux FIFO. Cependant, à la différence des KPN, les RTPN permettent les lectures non-bloquantes sur les canaux ainsi que les écritures non-bloquantes sur des canaux de taille finie. Cela permet de modéliser des aspects des systèmes embarqués,

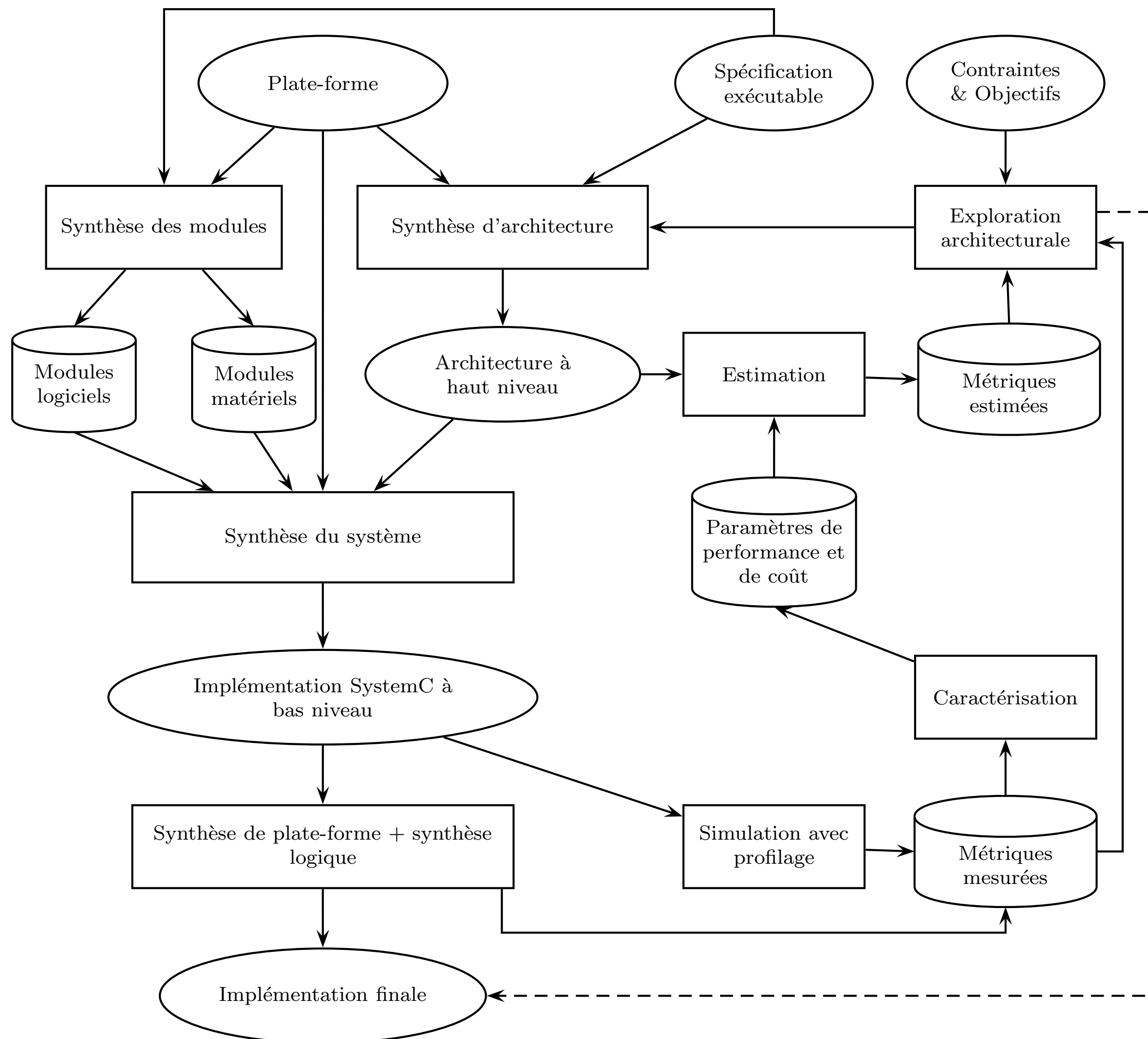

Figure 3.1 Aperçu général de la méthodologie proposée

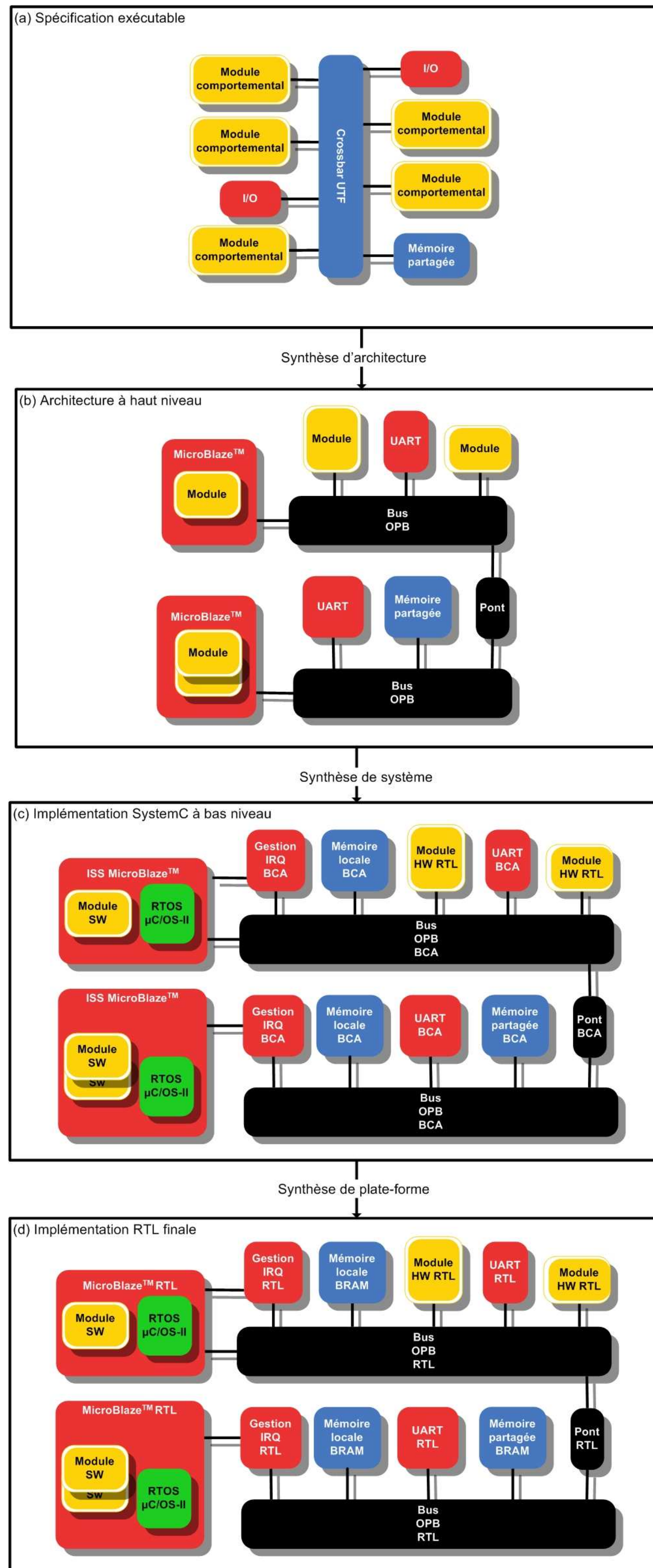

Figure 3.2 Évolution d'une application à différentes étapes de la méthodologie

tels que les senseurs échantillonnés, les périphériques d'entrée/sortie et les contraintes temps-réel, que les KPN ne modélisent pas adéquatement. Sous certaines conditions, les RTPN sont un modèle de calcul déterministe dans un méta-modèle de calcul temporisé. Cette propriété est utile à la simulation et à la caractérisation d'une application embarquée représentée par un RTPN.

3.2.2 Spécification exécutable parallèle

Un système embarqué ciblant la plate-forme virtuelle SPACE est représenté sous la forme d'une spécification exécutable composée d'un ensemble de modules et de périphériques codés en SystemC (IEEE, 2005). Chaque module est un processus séquentiel et déterministe possédant un et un seul fil d'exécution (sous la forme d'un `SC_THREAD` ou `SC_CTHREAD` SystemC). Bien que chaque module soit séquentiel, ils s'exécutent en parallèle entre eux et cette spécification exécutable est donc parallèle. Le niveau d'abstraction de ces modules est similaire au niveau comportemental de SystemC (Grotker *et al.*, 2002) ou au niveau TLM (Cai et Gajski, 2003). En effet, le code de ces modules spécifie le comportement et les opérations d'entrée/sortie des modules sans spécifier le nombre d'instructions ou le nombre de cycles d'horloge que doivent prendre chaque opération. De plus, les modules ne manipulent pas directement les signaux SystemC (`sc_in`, `sc_out` et `sc_signal`) pour communiquer entre eux, mais ils utilisent plutôt des fonctions transactionnelles de lecture et d'écriture dont les paramètres sont présentés au tableau 3.1. Cela permet d'accélérer l'intégration dans les modules SystemC d'un code C/C++ déjà existant. Au niveau de la spécification exécutable, ces fonctions transactionnelles font transiter les communications entre modules via des canaux FIFO sans modéliser précisément les temps de communication. Les modules peuvent ainsi communiquer entre eux selon une sémantique de passage de messages qui est soit asynchrone (auquel cas le FIFO sert de tampon), soit par rendez-vous (auquel cas le module qui écrit bloque jusqu'à la réception d'un acquittement du module lecteur). De plus, les adresses utilisées par ces fonctions de communication sont indépendantes de l'implémentation logicielle ou matérielle des modules : cela permet d'avoir une seule et même version de la spécification exécutable pour un ensemble d'architectures hétéroclites.

Le fil d'exécution de chaque module peut contenir un code C/C++ ou SystemC arbitraire et, tant que ce code compile et s'exécute correctement, la spécification pourra s'exécuter correctement. Cependant, pour que ces modules puissent être synthétisés en un logiciel embarqué ou en un composant matériel par les étapes suivantes de la méthodologie, il est nécessaire de définir un sous-ensemble synthétisable de C/C++ et SystemC et d'imposer des contraintes à leurs fils d'exécution afin que leur code soit dans ce sous-ensemble. Ces contraintes s'appliquent également à toutes les fonctions appelées par les modules, à l'exception des fonctions

Tableau 3.1 Paramètres des fonctions de communication de SPACE

Paramètre	**Utilisation en lecture**	**Utilisation en écriture**
Adresse distante	Adresse où lire les données.	Adresse où envoyer les données.
Taille	Nombre d'octets à lire.	Nombre d'octets à écrire.
Tampon de données	Tampon local où sont écrites les données lues.	Tampon local d'où sont lues les données à écrire.
Bloquant ?	Indique si la lecture bloque en l'absence de données.	Indique si l'écriture bloque jusqu'à l'acquittement de la lecture.

transactionnelles de communication qui sont traitées séparément dans la méthodologie. Les règles à suivre sont les suivantes :

- Les seuls types de données primitifs utilisés sont les types primitifs booléens (`bool`) et entiers (`char`, `short`, `int`, `long` et `long long`, signés ou non) de C/C++ ainsi que les types de données primitifs entiers de SystemC (`sc_uint` et `sc_int`). L'utilisation des types de données primitifs à virgule flottante de C/C++ (`float` et `double`) est autorisée uniquement si le module sera nécessairement implémenté en logiciel.
- Les seuls pointeurs, tableaux, structures et classes utilisés font uniquement référence (soit directement, soit indirectement sans récursivité) à des types de données primitifs autorisés ou à des types énumérés de C/C++ (`enum`).
- Les seuls types de données transmis via les fonctions transactionnelles de communication sont les types primitifs booléens et entiers de C/C++, les types énumérés de C/C++, ou des tableaux, structures ou classes composés uniquement de ces types.
- Aucune allocation dynamique de mémoire (`new`, `alloc`, `malloc`) n'est faite.
- Aucun typage dynamique (tel que `dynamic_cast`) n'est effectué.

La spécification exécutable parallèle peut également contenir des périphériques en plus des modules. La plate-forme virtuelle SPACE fournit des modèles purement fonctionnels de périphériques, tels que des blocs de mémoire et des périphériques d'entrée/sortie, qui peuvent être insérés dans la spécification exécutable. L'ajout de telles mémoires permet aux modules de communiquer indirectement entre eux selon une sémantique de mémoire partagée. La principale différence entre les modules et les périphériques sont que les modules sont des maîtres dotés de leur propre fil d'exécution et capables de lancer des requêtes de communication alors que les périphériques sont des esclaves qui ne font que répondre aux requêtes qui leur sont envoyées. De plus, les modules sont généralement spécifiques à l'application et fournis par l'utilisateur de la méthodologie alors que les périphériques sont des composants génériques. Les communications entre les modules et les périphériques se font également au moyen des fonctions transactionnelles de lecture et d'écriture.

Étant donné que la spécification exécutable est composée de modules, périphériques et

canaux FIFO qui sont tous des composants SystemC comportementaux, celle-ci peut être exécutée par un simulateur SystemC à un niveau purement fonctionnel, sans se préoccuper de la performance d'une éventuelle implémentation. La figure 3.2(a) est une représentation schématique d'une spécification exécutable d'un système embarqué dans SPACE composé de cinq modules comportementaux, deux périphériques d'entrée/sortie et une mémoire partagée. Cette spécification se trouve au niveau Elix selon la terminologie de SPACE (Filion *et al.*, 2007; Bois *et al.*, 2010).

SpaceStudio (Filion *et al.*, 2007) est un environnement de développement intégré basé sur Eclipse (Clayberg et Rubel, 2006) qui facilite notamment le développement, la simulation, le débogage et la validation des spécifications exécutables compatibles avec la plate-forme virtuelle SPACE. Dans le cadre de la méthodologie présentée dans cette thèse, on suppose que la spécification exécutable est fonctionnellement correcte et qu'elle a donc déjà été déboguée et validée. On suppose également qu'un banc d'essai, qui permet d'exercer la spécification exécutable, est fourni avec celle-ci et que sont spécifiées les contraintes temporelles que doivent respecter les entrées et les sorties d'une implémentation du système embarqué.

3.2.3 Plate-forme virtuelle SPACE

La plate-forme virtuelle SPACE (Filion *et al.*, 2007; Bois *et al.*, 2010) offre les fonctions transactionnelles de communications décrites à la section 3.2.2 ainsi que des modèles SystemC de bus, processeurs, mémoires et périphériques à différents niveaux d'abstraction. Cette plate-forme virtuelle peut ainsi être utilisée pour développer, raffiner, simuler, déboguer et optimiser les systèmes embarqués.

Les fonctions transactionnelles de communications et les modèles comportementaux de périphériques fournis par SPACE sont utilisés lors de la spécification exécutable parallèle du système. Les modèles transactionnels de SPACE sont à un niveau d'abstraction plus bas que ces modèles comportementaux, mais plus élevé que le niveau RTL. Ainsi, les bus, processeurs, mémoires et périphériques y sont modélisés au niveau des transferts de paquets, des instructions logicielles, des accès à la mémoire et des requêtes et réponses plutôt que d'être modélisés au niveau RTL, où on tenterait de déterminer à chaque cycle la valeur exacte de chaque broche et de chaque signal interne de chaque composant. Par exemple, les bus modélisent les délais de communication en tenant compte de l'arbitrage, de la transmission, du pipelinage et de l'acquittement des transferts. Les modèles de processeur sont des simulateurs de jeu d'instructions (ISS) qui exécutent le code logiciel cible cycle par cycle en tenant compte des accès au bus et de la hiérarchie de mémoire. Les modèles de périphériques simulent leur temps de réponse aux requêtes. On modélise ainsi le comportement de ces composants et les délais associés à leurs fonctionnalités sans avoir à modéliser tous leurs détails. Ces composants

peuvent ainsi être utilisés pour une simulation approximativement précise au cycle près d'une implémentation SystemC du système embarqué qui est bien plus rapide qu'une simulation RTL complète.

Les composants suivants sont intégrés dans la plate-forme SPACE :

- Les bus OPB (*On-chip Peripheral Bus*), PLB (*Processor Local Bus*) et DCR (*Device Control Register*) du standard CoreConnect (IBM Corp., 1999) ;
- Les liens de communication FIFO point-à-point, avec support du protocole FSL (*Fast Simplex Link*) (Rosinger, 2004) ;
- Les processeurs MicroBlaze (Xilinx Inc., 2005) et PowerPC 405 (IBM Microelectronics Division, 1998) ;
- Les mémoires SRAM et SDRAM ;
- Un contrôleur programmable d'interruption ;
- Une minuterie programmable ;
- Un UART (EIA, 1969) ;

Les bus OPB, les processeurs MicroBlaze, les mémoires SRAM et SDRAM ainsi que les périphériques les suivant sont supportés par la méthodologie présentée dans cette thèse.

La plate-forme virtuelle définit quels types de composants peuvent être alloués dans une architecture et comment ces types de composants peuvent être connectés entre eux. Cependant, la plate-forme virtuelle n'impose ni une allocation particulière de composants, ni une topologie particulière de communications.

3.2.4 Synthèse d'architecture

La synthèse d'architecture consiste à préparer un devis pour l'implémentation sur la plate-forme virtuelle SPACE du système embarqué décrit par la spécification exécutable. Ainsi, la synthèse d'architecture consiste à choisir une allocation des composants de la plate-forme et une assignation des modules et périphériques de la spécification à ces composants.

Dans la présente méthodologie, la synthèse d'architecture se subdivise en trois sous-problèmes tel que décrit à la section 8.1 :

(a) Le partitionnement logiciel/matériel, qui consiste à déterminer quels modules de l'application seront implémentés en tant que tâches logicielles et lesquels seront implémentés en tant que composants matériels ;

(b) L'allocation des processeurs et l'assignation des tâches aux processeurs, qui consiste à déterminer le nombre de processeurs à allouer puis à assigner chaque tâche logicielle à un des processeurs alloués ;

(c) Le choix d'une topologie de communications, qui consiste à déterminer le nombre de bus à allouer puis à assigner chacun des composants matériels du système à ces bus.

Par exemple, la synthèse d'architecture de la spécification exécutable de la figure 3.2(a) résulte en l'architecture illustrée à la figure 3.2(b), qui représente le devis suivant pour l'implémentation du système embarqué :

(a) Les trois premiers modules seront implémentés en logiciel et les deux autres le seront en matériel.
(b) Deux processeurs MicroBlaze seront alloués, le premier module logiciel sera assigné au premier processeur et les deux autres modules logiciels seront assignés au deuxième processeur.
(c) Deux bus OPB seront alloués. Le premier processeur, les deux modules matériels et le premier UART seront assignés au premier bus. Le deuxième processeur, le deuxième UART et la mémoire partagée seront assignés au deuxième bus. De plus, un pont reliera entre eux les bus.

L'environnement de développement SpaceStudio (Filion *et al.*, 2007) permet de créer une nouvelle architecture ou de modifier une architecture existante. Une particularité de SPACE est que les modules du système embarqué qui respectent les contraintes présentées à la section 3.2.2 peuvent être implémentés autant en logiciel qu'en matériel. Il est donc possible de modifier rapidement le partitionnement logiciel/matériel d'une application dans SpaceStudio par un glisser-déposer (*drag and drop*) d'un module d'un processeur vers un bus (ou vice versa). La méthodologie présentée ici tire profit de cette flexibilité pour explorer une large gamme d'architectures. Cependant, pour automatiser la boucle d'exploration architecturale, la synthèse d'architecture ne s'effectue pas au moyen de l'interface graphique de SpaceStudio, mais plutôt de manière programmative.

3.2.5 Synthèse des modules

Les modules de la spécification exécutable sont à un niveau SystemC purement comportemental. Ils s'exécutent donc directement et nativement dans le simulateur SystemC, et ce sans modéliser la performance de leurs opérations. Bien que cela soit parfaitement approprié pour une spécification exécutable, cela pose deux problèmes dans le raffinement vers une implémentation embarquée : l'implémentation finale (FPGA ou ASIC) n'exécutera pas de simulateur SystemC et les informations sur la performance des modules sont nécessaires à l'exploration architecturale. La synthèse des modules résout ces deux problèmes en générant pour chaque module une implémentation en tant que tâche logicielle pour le processeur et le RTOS cible ou en tant que composant matériel RTL pour la technologie cible. Cette implémentation logicielle ou matérielle des modules peut alors être utilisée pour le raffinement vers l'implémentation finale de l'application ou pour une simulation approximativement précise au cycle près.

3.2.5.1 Synthèse logicielle

La synthèse des modules vers une implémentation logicielle embarquée a été présentée en détail dans (Chevalier *et al.*, 2006) et on en résume ici les grandes lignes. La figure 3.3 illustre la structure d'un logiciel embarqué synthétisé pour un processeur dans SPACE. Ainsi, un tel processeur embarqué exécute un système d'exploitation temps-réel (RTOS) avec une couche d'abstraction matérielle (HAL). Le RTOS et la HAL se chargent notamment de l'ordonnancement dynamique des tâches logicielles, des changements de contexte, du traitement des interruptions et des accès au bus. Le défi est donc de faire en sorte qu'un module de la spécification exécutable puisse s'exécuter sur un tel processeur cible comprenant un RTOS plutôt que de s'exécuter nativement dans un simulateur SystemC.

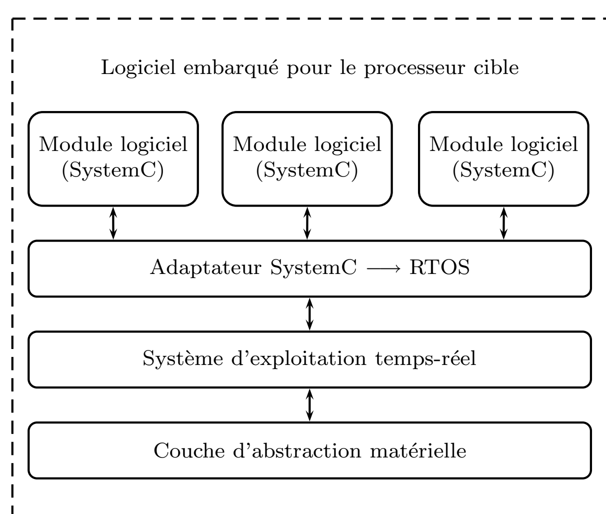

Figure 3.3 Structure d'un logiciel embarqué synthétisé pour un processeur dans SPACE

Tel qu'illustré à la figure 3.3, la solution retenue consiste à insérer entre les modules et le RTOS une interface de programmation (API) qui convertit les appels aux fonctions SystemC en des appels aux fonctions du RTOS. Cette API a reçu le nom de Tor (pour *Translator*) (Bois *et al.*, 2010). On trouve dans (Chevalier *et al.*, 2006) une liste de conversions entre les appels SystemC et les appels du RTOS μC/OS-II (Labrosse, 2002). Par exemple, la déclaration d'un fil d'exécution SystemC par `SC_THREAD` est converti à un appel à la fonction `taskCreate` de cette API qui, pour μC/OS-II, est elle-même réalisée par la fonction `OSTaskCreate` du RTOS. Les types de données primitifs entiers de SystemC (`sc_uint` et `sc_int`) sont convertis selon leur nombre de bits en des types de données entiers équivalents de C/C++ en suivant les règles présentées au tableau 3.2. Le Tor comprend également un gestionnaire de communications qui implémente les fonctions transactionnelles de communications à l'aide des fonctions du RTOS et du HAL.

Grâce au Tor, le fil d'exécution C/C++/SystemC du module de la spécification exécutable est synthétisé en un fil d'exécution C/C++ qui peut s'exécuter en tant que tâche logicielle sur le RTOS cible. De plus, cette synthèse logicielle s'effectue au moyen d'une compilation croisée ciblant le jeu d'instructions du processeur embarqué : l'implémentation logicielle du module pourra alors être exécutée instruction par instruction par un ISS de la plate-forme virtuelle ou par un processeur embarqué de l'implémentation finale.

Tableau 3.2 Conversion des types de données SystemC pour un logiciel embarqué

Type SystemC	Nombre de bits	Type C/C++ équivalent
`sc_uint`	1 à 8	`unsigned char`
`sc_uint`	9 à 16	`unsigned short`
`sc_uint`	17 à 32	`unsigned long`
`sc_uint`	33 à 64	`unsigned long long`
`sc_int`	1 à 8	`char`
`sc_int`	9 à 16	`short`
`sc_int`	17 à 32	`long`
`sc_int`	33 à 64	`long long`

3.2.5.2 Synthèse des modules matériels

La synthèse des modules vers une implémentation matérielle au niveau RTL a été présentée pour la première fois dans (Moss *et al.*, 2008) et est détaillée au chapitre 5. Cette synthèse matérielle s'effectue en deux étapes principales. D'abord, les communications effectuées à un niveau transactionnel par les modules de la spécification exécutable doivent être raffinées vers un protocole RTL précis au cycle et à la broche près. Tel qu'il sera décrit à la section 5.2, la synthèse matérielle automatise la génération de transacteurs TLM-RTL pour le raffinement des communications des modules vers le protocole RTL défini pour SPACE dans (Faiz, 2007). Ensuite, chaque module, conjointement avec son transacteur TLM-RTL, est synthétisé vers une implémentation matérielle par un outil de synthèse comportementale. On présente à la section 5.3 comment l'outil de synthèse comportementale Cynthesizer (Meredith, 2008) est intégré dans cette synthèse matérielle automatisée.

3.2.6 Synthèse du système

La synthèse du système met en oeuvre le devis produit par la synthèse d'architecture et génère une implémentation SystemC de l'application composée des implémentations logicielles et matérielles des modules ainsi que des modèles transactionnels de bus, processeurs et

périphériques de la plate-forme virtuelle. Par exemple, la synthèse du système effectue les opérations suivantes pour générer l'implémentation de la figure 3.2(c) à partir de l'architecture de la figure 3.2(b) :

1. Deux modèles transactionnels de bus OPB sont alloués. Un modèle transactionnel de pont OPB-OPB est également alloué pour relier entre eux ces bus OPB. Ces modèles sont au niveau transactionnel BCA (Bus Cycle-Accurate), car les communications sont précises au cycle près sur le bus.

2. Pour chaque périphérique, la synthèse du système alloue un modèle transactionnel ayant une interface compatible avec celle du bus sur lequel il est assigné. Ainsi, un modèle transactionnel de l'UART ayant une interface transactionnelle OPB est connecté au premier bus OPB. Un deuxième modèle transactionnel de l'UART ainsi qu'un modèle transactionnel de mémoire partagée avec une interface OPB sont connectés au deuxième bus OPB.

3. Pour chaque module assigné en matériel, son implémentation RTL est connectée au bus auquel il est assigné. Un transacteur RTL-BCA et un modèle transactionnel d'adaptateur de bus sont insérés entre le module et le bus. Ce transacteur permet au module RTL de communiquer avec le reste du système, qui est au niveau transactionnel, alors que l'adaptateur de bus fait en sorte que ces communications s'effectuent selon le protocole du bus. (Pour des fins de simplification, les transacteurs et les adaptateurs de bus ne sont pas illustrés aux figures 3.2(c) et 3.2(d).)

4. Deux ISS de processeurs MicroBlaze avec interface OPB sont alloués et sont connectés à chacun des bus OPB. Pour chaque processeur, deux modèles transactionnels de périphériques dédiés au processeur sont également alloués : un gestionnaire programmable d'interruption et une mémoire locale. Ces périphériques sont connectés au même bus auquel le processeur est connecté et ont donc une interface compatible avec ce bus (OPB dans ce cas-ci).

5. Le logiciel embarqué de chaque processeur est généré par une compilation croisée de l'implémentation logicielle de chaque module qui lui est assigné, de l'API logicielle (Tor), du RTOS et du HAL, tel qu'illustré à la figure 3.3. Avant cette compilation croisée, le gestionnaire de communications de l'API logicielle est également configuré selon l'architecture du système embarqué. Le gestionnaire de communications permet ainsi à chaque module logiciel de communiquer avec les autres modules logiciels (sur le même processeur ou sur les autres processeurs) ainsi qu'avec les modules matériels et les périphériques (Chevalier *et al.*, 2006). Dans l'exemple de la figure 3.2(c), le premier ISS exécutera le code généré avec le premier module logiciel alors que le deuxième ISS exécutera le code généré avec les deux autres modules logiciels.

Le résultat d'une telle synthèse du système est une implémentation approximativement précise au cycle près du système embarqué qui peut être simulée en SystemC afin de vérifier

à la fois sa fonctionnalité et sa performance. Le fait que les bus, processeurs et périphériques soient des modèles transactionnels permet d'accélérer la simulation par rapport à une simulation 100% RTL.

3.2.7 Synthèse de plate-forme et synthèse logique

Tel qu'il sera décrit à la section 5.3.2, la synthèse de plate-forme remplace les modèles transactionnels des bus, processeurs et périphériques par des blocs RTL pour produire une implémentation RTL. Une telle implémentation RTL, illustrée à la figure 3.2(d), peut ensuite être synthétisée sur la technologie cible via une synthèse logique « traditionnelle ». L'outil GenX (Filion *et al.*, 2007; Bois *et al.*, 2010) permet de cibler les familles Virtex de FPGA Xilinx (Xilinx Inc., 2002). Une telle synthèse de plate-forme pourrait éventuellement cibler d'autres types de FPGA ou des ASIC.

3.2.8 Profilage au niveau système

Le profilage logiciel/matériel au niveau système a été présenté pour la première fois dans (Moss *et al.*, 2007) et sera présenté en détail au chapitre 6. Ce profilage se base sur une instrumentation non intrusive des modèles transactionnels de la plate-forme virtuelle pour extraire des métriques sur les performances des modules, des processeurs, des bus et des mémoires. Le caractère non intrusif de ce profilage fait en sorte qu'il mesure ces métriques sans les perturber.

3.2.9 Caractérisation et estimation

Le chapitre 7 présentera en détails une méthode d'estimation de la validité fonctionnelle, de la performance et du coût matériel d'une implémentation d'une application embarquée. Cette méthode d'estimation se base sur la caractérisation de l'application, de la plate-forme ainsi que des implémentations logicielles et matérielles des modules de l'application. La caractérisation du coût matériel des modules matériels et des composants de plate-forme s'effectue au moyen d'une synthèse logique initiale de ceux-ci. La caractérisation de la performance s'effectue à l'aide des métriques extraites par le profilage au niveau système lors de simulations initiales. La méthode d'estimation intègre ensuite le résultat de cette caractérisation dans un modèle de performance de l'application et de la plate-forme virtuelle afin d'estimer rapidement les métriques associées à un ensemble d'implémentations de l'application.

3.2.10 Exploration architecturale

L'exploration architecturale modifie itérativement l'architecture de l'application de manière à optimiser la performance et le coût matériel de son implémentation tout en respectant les contraintes de validité fonctionnelle, de performance et de coût matériel spécifiées pour l'application. L'exploration architecture se base soit sur les métriques mesurées par la synthèse logique et le profilage, soit sur les métriques estimées, pour évaluer et comparer entre elles les différentes architectures de l'application. Le chapitre 8 analyse en détails les problèmes d'optimisation que vise à résoudre l'exploration architecturale ainsi que plusieurs algorithmes qui permettent de résoudre ces problèmes de manière exacte ou heuristique.

CHAPITRE 4

RÉSEAUX DE PROCESSUS TEMPS-RÉEL

Les réseaux de processus Kahn (KPN) (Kahn, 1974) sont un modèle de calcul parallèle tel que le comportement d'un KPN est défini uniquement par la fonctionnalité des processus qui le composent. En particulier, elle ne dépend pas de l'ordre dans lequel les processus sont ordonnancés et exécutés. Différentes implémentations d'un système spécifié par un KPN peuvent ainsi faire varier cet ordonnancement, par exemple selon le temps d'exécution des processus, sans affecter la fonctionnalité du système. Cette propriété est intéressante pour une méthodologie d'exploration architecturale, comme celle présentée au chapitre 8, qui doit comparer différentes implémentations d'un même système. Cependant, les KPN ne peuvent pas exprimer des concepts importants du traitement temps-réel tels que les timeouts, les interruptions, la scrutation (polling) ou les contraintes temps-réel étant donné l'absence d'une opération de lecture non-bloquante. Ce chapitre présente les réseaux de processus temps-réel (RTPN), un modèle de calcul qui étend les KPN en permettant également les lectures non-bloquantes sur un canal, de même que des écritures non-bloquantes sur un canal de taille finie. On étudie dans quelle mesure le comportement d'un RTPN dépend de l'ordonnancement des processus et on indique comment ce modèle de calcul peut être appliqué à la modélisation des systèmes embarqués, incluant les spécifications exécutables créées avec SPACE. Ce modèle de calcul est également utilisé au chapitre 7 pour vérifier qu'une implémentation d'une application embarquée a une fonctionnalité équivalente à sa spécification exécutable.

4.1 Définition des réseaux de processus temps-réel

Cette section présente une définition des réseaux de processus temps-réel. On commence d'abord par définir plus généralement les réseaux de processus (PN). Un PN est un graphe orienté $G = (P, C)$ tel que les noeuds P sont l'ensemble des processus du réseau et les arcs C sont l'ensemble des canaux reliant les processus entre eux. De plus, tous les processus $p \in P$ sont déterministes et séquentiels et tous les canaux $c \in C$ dont la taille est supérieure à 1 sont des FIFO. Si un arc c relie un sommet x à un sommet y dans un graphe, alors on a $c = (x, y)$, $\text{tail}(c) = x$ et $\text{head}(c) = y$. On peut supposer, sans perte de généralité, qu'il existe un processus $\omega \in P$ qui correspond à l'environnement du système, alors que les autres processus $P \setminus \{\omega\}$ correspondent au système. L'ensemble des canaux $I \subseteq C$ d'entrée du système sont alors les canaux $i \in I$ tel que $tail(i) = \omega$ alors que l'ensemble des canaux de sortie $O \subseteq C$ sont les

canaux $o \in O$ tel que $head(o) = \omega$. La figure 4.1 illustre un exemple de reséau de processus (le processus ω est omis pour des fins de simplification).

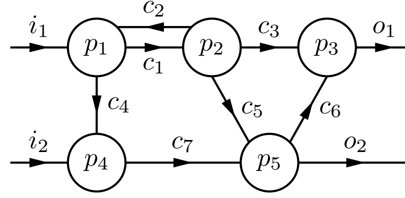

Figure 4.1 Exemple de réseau de processus avec $P = \{\omega, p_1, \ldots, p_5\}$, $C = \{c_1, \ldots, c_7, i_1, i_2, o_1, o_2\}$, $I = \{i_1, i_2\}$, $O = \{o_1, o_2\}$

4.1.1 Évènements et relation de précédence

Différents évènements peuvent se produire dans un PN, par exemple lorsqu'un processus effectue un calcul interne, tente de lire dans un canal ou d'y écrire. On définit E comme l'ensemble des évènements qui se produisent dans un PN de telle sorte qu'un événement $e \in E$ est soit un événement interne à un processus $p \in P$, soit une lecture ou une écriture dans un canal $c \in C$. L'écriture dans un canal d'entrée $i \in I$ et la lecture dans un canal de sortie $o \in O$ correspondent respectivement à la production d'une entrée ou à la consommation d'une sortie par l'environnement du système.

On définit la relation de précédence $<$ comme une relation binaire sur E tel que, pour toute paire d'évènements $x, y \in E$, $x < y$ signifie que x doit nécessairement se produire avant y ou, de manière équivalente, que y ne peut pas se produire tant que x ne s'est pas produit. Cette relation de précédence est un ordre partiel strict sur E et a donc les propriétés suivantes pour tout $x, y, z \in E$:

1. $x \nless x$ (irréflexivité).
2. Si $x < y$, alors $y \nless x$ (asymétrie).
3. Si $x < y$ et $y < z$, alors $x < z$ (transitivité).

Étant donné que les processus d'un PN sont séquentiels et déterministes, il existera généralement une relation de précédence entre les évènements correspondant aux opérations d'un même processus. La relation de précédence qui existe entre les différents processus dépend des communications effectuées par ceux-ci et des politiques des canaux quant aux lectures dans un canal vide et aux écritures dans un canal plein. Les ensembles d'évènements correspondant aux requêtes de communication ainsi qu'aux communications elles-mêmes sont définis comme des fonctions $w_b, w_e, r_b, r_e, a : C \times \mathbb{N} \to E$ de telle sorte que :

1. $w_b(c, i)$ est le début de la requête d'écriture du i^{eme} jeton dans le canal c.
2. $w_e(c, i)$ est la fin de la requête d'écriture du i^{eme} jeton dans le canal c.

3. $r_b(c,i)$ est le début de la requête de lecture du i^{eme} jeton dans le canal c.
4. $r_e(c,i)$ est la fin de la requête de lecture du i^{eme} jeton dans le canal c.
5. $a(c,i)$ est le moment où le i^{eme} jeton est disponible pour lecture dans c.

On suppose, pour des raisons de simplification et sans perte de généralité, que tous les canaux sont initialement vides. Étant donné qu'une requête de communication doit débuter avant de se terminer, on a par définition $w_b(c,i) < w_e(c,i)$ pour tout c,i et $r_b(c,i) < r_e(c,i)$ pour tout c,i. Les évènements $w_e(c,i)$ et $a(c,i)$ correspondent respectivement à l'envoi et à la réception d'un jeton et, comme un jeton doit être envoyé avant d'être reçu, on a $w_e(c,i) < a(c,i)$ pour tout c,i. Les évènements $w_b(c,i)$ et $w_e(c,i)$ font partie du processus qui envoie le jeton alors que les évènements $r_b(c,i)$, $r_e(c,i)$ et $a(c,i)$ font partie du processus qui reçoit le jeton : cela permet d'établir une relation de précédence entre des évènements de deux processus distincts.

Finalement, on suppose qu'un processus ne peut effectuer aucune autre opération tant qu'une requête de communication qu'il a initiée n'a pas été terminée. Ainsi, pour les évènements d'un même processus $p \in P$, on a que, pour tout évènement x qui n'est pas une réception de jeton ($x \notin a(C \times \mathbb{N})$), on a :

1. Si $w_b(c,i) < x$, alors $w_e(c,i) < x$.
2. Si $x < w_e(c,i)$, alors $x < w_b(c,i)$.
3. Si $r_b(c,i) < x$, alors $r_e(c,i) < x$.
4. Si $x < r_e(c,i)$, alors $x < r_b(c,i)$.

La relation de précédence $<$ présentée dans cette section a quelques similarités avec la relation de précédence $\prec$ présentée dans (Lamport, 1978). Cependant, la différence fondamentale est que la relation $\prec$ répond à la question « Est-ce que, pour une exécution donnée, l'évènement x est arrivé avant l'évènement y ? » alors que la relation $<$ répond à la question « Est-ce que, pour toute exécution, x doit nécessairement arriver avant y ? » On a donc que $x < y$ implique $x \prec y$, mais que l'inverse n'est pas nécessairement vrai.

4.1.2 Types de canaux et de réseaux de processus

Un PN peut contenir différents types de canaux qui se distinguent par leur taille et leur politique quant aux lectures dans un canal vide et aux écritures dans un canal plein. Ainsi, la lecture peut être bloquante ou non-bloquante. Si la lecture dans un canal c est bloquante, alors on a $a(c,i) < r_e(c,i)$ pour tout i, étant donné que le jeton doit être reçu avant que la requête de lecture de celui-ci se termine. Par contre, si la lecture dans un canal c est non-bloquante, alors il n'y a pas de relation de précédence entre ces évènements, car la requête de lecture peut se terminer même si le jeton n'a pas été reçu. Le consommateur obtient alors un jeton avec une valeur spéciale qu'on nomme valeur absente.

Un canal peut être de taille infinie ou finie. Si un canal est de taille infinie, alors l'écriture dans ce canal est non-bloquante et non-destructive. En effet, un canal infini ne peut jamais être plein et ne peut donc jamais déborder lors d'une écriture non-bloquante. La contrepartie est qu'une écriture non-bloquante dans un canal de taille finie peut être destructive parce que, si le canal est plein, un jeton précédemment écrit mais non lu sera écrasé par le nouveau jeton. Si un canal de taille finie peut contenir au maximum N jetons et que l'écriture dans le canal bloque lorsque celui-ci est plein, alors on a $r_e(c,i) < w_e(c,i+N)$ pour tout i. En effet, si les N jetons $(i, i+1, \ldots, i+N-1)$ se trouvent dans le canal, alors l'écriture du jeton $i+N$ ne pourra pas se terminer tant que le jeton i n'aura pas été lu.

Tableau 4.1 Définition des canaux Kahn, POLL, N-BREG et N-REG

Nom du canal	Taille	Écriture	Lecture
Kahn	Infinie	Non-bloquante et non-destructive	Bloquante
POLL	Infinie	Non-bloquante et non-destructive	Non-bloquante
N-BREG	$N \in \mathbb{N}$	Non-bloquante et destructive	Bloquante
N-REG	$N \in \mathbb{N}$	Non-bloquante et destructive	Non-bloquante

Les canaux Kahn, POLL, N-BREG et N-REG sont définis comme des canaux FIFO ayant les propriétés indiquées au tableau 4.1. Ainsi, un canal Kahn est un canal FIFO de taille infinie avec lecture bloquante, tel les canaux d'un KPN. Le canal POLL est une modification du canal Kahn permettant une lecture non-bloquante sur un canal vide. Un canal REG, analogue à un registre pouvant contenir un seul jeton, a été présenté dans (van Dijk *et al.*, 2003) et permet une écriture non-bloquante et destructive ainsi qu'une lecture non-bloquante. On généralise ici ce canal par un canal FIFO N-REG pouvant contenir N jetons. Le canal N-BREG est une modification du canal N-REG avec lecture bloquante. Il serait possible de définir d'autres types de canaux qui imposent une écriture bloquante mais, pour des raisons de simplification, on se limite ici à ces 4 types de canaux. Tel qu'il sera présenté à la section 4.3.5, ceux-ci sont suffisants pour modéliser une écriture bloquante avec acquittement comme celle utilisée dans SPACE. Un réseau de processus Kahn (Kahn, 1974) (KPN) est donc un réseau de processus tel que chaque canal $c \in C$ est un canal Kahn. Un réseau de processus temps-réel (RTPN) est défini comme un réseau de processus tel que chaque canal $c \in C$ est un canal Kahn, POLL, N-REG ou N-BREG. Un KPN est donc un cas particulier d'un RTPN.

L'ajout de canaux N-(B)REG permet aux RTPN de modéliser l'utilisation des mémoires partagées et des mécanismes d'entrée/sortie, tel qu'il sera décrit à la section 4.3. L'ajout des canaux POLL permet également de modéliser des mécanismes couramment utilisés dans les systèmes temps-réel, tels que la scrutation, les interruptions et les délais d'attente (*timeouts*)

(Panangaden et Stark, 1988). Cela permet donc aux RTPN de modéliser les aspects dépendant du temps que ne peuvent pas modéliser les KPN.

4.2 Sémantique dénotationnelle des RTPN

Soit S l'ensemble des séquences de jetons. Un processus RTPN (ou KPN) avec i canaux d'entrée et o canaux de sorties est une fonction $f : S^i \to S^o$. La sémantique dénotationnelle des KPN permet d'exprimer le comportement d'un KPN à partir du comportement de chacun de ses processus et des connexions entre ceux-ci. On obtient un système d'équations en appliquant les fonctions des processus à leurs entrées et sorties et le comportement du KPN est le plus petit point fixe de ce système d'équations, tel que décrit à la section 2.1.4.1.

Cette sémantique dénotationnelle des KPN suppose que les séquences de jetons ne sont pas modifiées lors de leur transfert sur un canal. Un canal Kahn garantit cette propriété grâce à sa lecture bloquante et à sa taille infinie. Cependant, cette propriété n'est garantie ni par les canaux POLL (des jetons de valeur absente peuvent être ajoutés), ni par les canaux N-REG ou N-BREG (des jetons peuvent être écrasés). De plus, ces canaux ne sont pas des fonctions $f : S \to S$ étant donné que les modifications apportées aux séquences de jeton dépendent de l'ordre dans lequel les processus producteur et consommateur accèdent au canal. Ainsi, les fonctions sur les séquences de jetons ne suffisent pas à définir la sémantique dénotationnelle des RTPN et il faut étudier l'ordonnancement des processus.

Un ordonnancement d'un réseau de processus associe un temps à chaque événement du réseau de processus en respectant la relation de précédence entre ces évènements. Ces temps peuvent représenter un temps physique, un temps logique (Lamport, 1978) ou un nombre de cycles d'horloge. Un ordonnancement est donc une fonction $t : E \to \mathbb{R}$ strictement monotone, ce qui signifie que, pour tout $x, y \in E$, $x < y$ implique $t(x) < t(y)$. Par exemple, l'ensemble d'évènements dont la relation de précédence est illustrée à la figure 4.2 subit un ordonnancement t à la figure 4.3. Un tel ordonnancement peut notamment être induit par une implémentation du RTPN.

Un RTPN dont on a fixé l'ordonnancement t des évènements devient un réseau de processus à évènements discrets (DEPN). Il est alors possible de définir les processus et les canaux

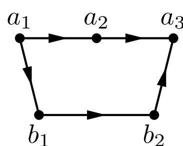

Figure 4.2 Évènements des processus a et b selon une relation de précédence.

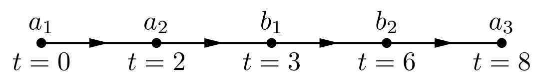

Figure 4.3 Évènements des processus a et b selon un ordonnancement t.

des RTPN comme des fonctions sur des séquences de jetons qui, contrairement aux KPN, sont temporisés. L'annexe A présente une modélisation des RTPN en tant que DEPN et démontre comment les RTPN acquièrent alors la sémantique dénotationnelle des DEPN en tant que réseau de processus temporisés. Ainsi, la simulation ou l'exécution d'un tel RTPN temporisé est déterministe.

Une telle sémantique dénotationnelle demande une temporisation au moins précise au cycle près et est donc appropriée une fois qu'une implémentation RTL du système embarqué a été synthétisée. Cependant, au niveau de la spécification et de l'exploration architecturale, la temporisation des différents processus et canaux n'a pas encore été fixée et il est donc nécessaire d'avoir une sémantique dénotationnelle plus flexible. On étudie ici sous quelles conditions deux ordonnancements distincts d'un même RTPN sont fonctionnellement équivalents, ce qui permet d'isoler l'effet de l'ordonnancement du RTPN sur sa fonctionnalité et de définir une sémantique dénotationnelle des RTPN comme un KPN paramétré.

4.2.1 Ordonnancements équivalents à un KPN classique

On commence par déterminer sous quelles conditions un ordonnancement d'un RTPN fait en sorte qu'il se comporte comme un KPN classique. Ainsi, si un canal $c \in C$ est avec lecture bloquante, alors on a, pour tout i, $a(c,i) < r_e(c,i)$ et donc $t(a(c,i)) < t(r_e(c,i))$ pour tout ordonnancement t. À l'opposé si, pour un ordonnancement t donné et un entier i donné, on observe $t(a(c,i)) \nless t(r_e(c,i))$, alors le canal c est avec lecture non-bloquante. En effet, cela signifie que le processus consommateur a tenté au moins une fois de lire un jeton dans le canal c lorsque celui-ci était vide et que le canal a alors retourné une valeur absente plutôt que d'attendre la réception du jeton. Par contre, si on observe, pour un ordonnancement t donné, $t(a(c,i)) < t(r_e(c,i))$ pour tout i, alors il n'est pas possible de dire à partir de ces seules observations si le canal c est avec lecture bloquante ou non-bloquante. On affirme donc qu'il se comporte, pour cet ordonnancement t, comme si il était avec lecture bloquante, dans le sens où son comportement ne peut pas être distingué de celui d'un canal avec lecture bloquante.

Si un canal $c \in C$ est de taille finie N et qu'on observe, pour un ordonnancement t donné et un entier i donné, que $t(r_e(c,i)) \nless t(w_e(c,i+N))$, alors cela signifie que le processus producteur a écrit au moins un jeton dans le canal alors que celui-ci était plein et qu'un jeton précédemment écrit mais non lu a donc été détruit lors de cette écriture non-bloquante et destructive. Par contre, si observe, pour un ordonnancement t donné, que $t(r_e(c,i)) < t(w_e(c,i+N))$ pour tout i, alors cela signifie qu'aucun jeton n'a été écrasé dans le canal c.

Un canal Kahn a comme propriétés que la lecture y est bloquante et que toutes les valeurs qui y sont écrites peuvent éventuellement être lues (étant donné que l'écriture est non-

destructive). Un canal se comporte donc comme un canal Kahn pour un ordonnancement t donné si et seulement si il satisfait les conditions suivantes : son comportement ne peut pas être distingué de celui d'un canal avec lecture bloquante et aucun jeton n'y est écrasé. En d'autres termes, un canal de taille infinie se comporte comme un canal Kahn pour un ordonnancement t donné si et seulement si $t(a(c,i)) < t(r_e(c,i))$ pour tout i. Cette condition est toujours respectée pour un canal Kahn, mais pas nécessairement pour un canal POLL. Un canal de taille N se comporte comme un canal Kahn pour un ordonnancement t donné si et seulement si $t(a(c,i)) < t(r_e(c,i))$ et $t(r_e(c,i)) < t(w_e(c,i+N))$ pour tout i. La première condition est toujours respectée pour un canal N-BREG, mais pas nécessairement pour un canal N-REG. Le tableau 4.2 résume ainsi quelles sont les conditions nécessaires et suffisantes pour que les types de canaux Kahn, POLL, N-BREG et N-REG se comportent comme un canal Kahn pour un ordonnancement t donné.

Tableau 4.2 Conditions à respecter pour qu'un canal se comporte comme un canal Kahn

Canal	Conditions à respecter
Kahn	Aucune (toujours vrai)
POLL	$t(a(c,i)) < t(r_e(c,i)), \forall i$
N-BREG	$t(r_e(c,i)) < t(w_e(c,i+N)), \forall i$
N-REG	$t(a(c,i)) < t(r_e(c,i)), \forall i$ et $t(r_e(c,i)) < t(w_e(c,i+N)), \forall i$

Pour un ordonnancement t donné, un RTPN se comporte comme un KPN si et seulement si chacun de ses canaux se comporte comme un canal Kahn. En d'autres termes, pour un ordonnancement t donné, un RTPN se comporte comme un KPN si et seulement si les accès à chacun de ses canaux POLL, N-BREG et N-REG respectent les contraintes temporelles présentées au tableau 4.2. En effet, on ne pourrait pas alors déterminer, par la seule observation de l'exécution de ce RTPN sous l'ordonnancement t, si il s'agit d'un KPN contenant uniquement des canaux Kahn ou bien d'un RTPN contenant également des canaux POLL, N-BREG et N-REG (qui se comportent comme des canaux Kahn pour t). De manière équivalente, le RTPN se comporterait de manière identique pour l'ordonnancement t si tous ses canaux étaient remplacés par des canaux Kahn. Si on dispose d'un ordonnancement t de ce RTPN (qui peut avoir été extrait par profilage de la simulation ou de l'exécution du RTPN), il est donc possible de vérifier si ce RTPN se comporte comme un KPN pour l'ordonnancement t. Pour un RTPN donné, on définit l'ensemble des ordonnancements équivalents à un KPN comme $T_{KPN} \subseteq [E \to \mathbb{R}]$ tel que ce RTPN se comporte comme un KPN pour tout $t \in T_{KPN}$.

4.2.2 Classes d'ordonnancements équivalents

Un KPN est tel que tous ses canaux appliquent la fonction identité $f(s) = s, \forall s$ aux séquences de jetons qui y transitent. Tel que décrit à la section précédente, un ordonnancement t d'un RTPN équivalent à un KPN classique est tel que tous les canaux du RTPN appliquent cette même fonction identité à leurs séquences de jetons. On peut généraliser cette définition. Deux ordonnancements t_1 et t_2 d'un RTPN sont fonctionnellement équivalents si et seulement si, pour chaque canal du RTPN, les fonctions $f_1 : S \to S$ et $f_2 : S \to S$ associées à ce canal selon respectivement t_1 et t_2 sont telles que $f_1(s) = f_2(s), \forall s$. En d'autres termes, ces deux ordonnancements sont équivalents si et seulement si, dans les deux cas, chaque canal ajoute des jetons de valeur absente (lecture non-bloquante sur un canal vide) et écrase des jetons (écriture non-bloquante dans un canal plein) exactement aux mêmes endroits dans sa séquence de jetons.

Pour représenter ces ajouts et ces retraits de manière concise, on définit une séquence de bits pour le port d'écriture et le port de lecture de chacun des canaux. Pour le port d'écriture, si le i^{eme} bit de la séquence de bits est à 1, alors le i^{eme} jeton écrit dans le canal est conservé, sinon il est effacé. Pour le port de lecture, un bit à 0 signifie l'ajout d'un jeton de valeur absente alors qu'un bit à 1 signifie le transfert d'un jeton de la séquence telle que modifiée par le port d'écriture. Deux ordonnancements d'un RTPN sont donc équivalents si et seulement si toutes leurs séquences de bits sont égales. Plus généralement, un ensemble $T \subseteq [E \to \mathbb{R}]$ d'ordonnancements forment une classe d'équivalence si et seulement si $t_1, t_2 \in T$ implique que toutes les séquences de bits t_1 et t_2 sont égales. En particulier, pour tout RTPN, les ordonnancements équivalents à un KPN classique forment la classe d'équivalence T_{KPN} pour laquelle toutes les séquences de bits sont exclusivement composées de 1 (aucun jeton n'est ajouté ou retiré).

Une procédure simple permet d'extraire ces séquences de bits pour un ordonnancement donné d'un RTPN. Pour chaque canal c, la séquence de lecture est construite en ajoutant 0 à la séquence de bits lorsqu'une lecture non-bloquante est effectuée dans c alors qu'il est vide et en ajoutant 1 pour toute autre lecture. La séquence d'écriture d'un canal de taille infinie est construite en ajoutant 1 à la séquence à chaque écriture. La séquence d'écriture d'un canal de taille N est initialisée avec une série de $N - 1$ bits à 1, puis on y ajoute 1 à chaque lecture d'un jeton (excluant les lectures non-bloquantes dans un canal vide) et on y ajoute 0 à chaque fois qu'une écriture non-bloquante est effectuée dans un canal plein. Cette modélisation des séquences d'écriture suppose que l'écriture non-bloquante dans un canal plein écrase le dernier jeton précédemment écrit. Ainsi, deux ordonnancements sont équivalents si et seulement si ils produisent les mêmes séquences de bits et pareillement pour les classes d'équivalence.

Cette définition crée des classes d'équivalences entre les ordonnancements qui permettent de comparer entre elles deux implémentations fonctionnellement équivalentes mais avec des performances différentes. Cela est particulièrement intéressant pour l'exploration architecturale, où on désire faire varier les caractéristiques de performance de l'architecture tout en s'assurant de préserver la fonctionnalité de l'application.

4.2.3 Représentation d'un RTPN comme un KPN paramétré

Il est possible d'utiliser les séquences de bits définies à la section précédente pour modéliser chaque canal d'un RTPN comme une fonction $f : S \times S_b \times S_b \to S$, où S_b est l'ensemble des séquences de bits. Cette fonction est réalisée par le processus KPN défini selon le langage de (Kahn, 1974) à la figure 4.4. La première boucle ajoute un jeton de valeur absente à la séquence de jetons à la sortie du canal pour chaque 0 dans la séquence de bits de lecture. La deuxième boucle efface (plus précisément, omet de transmettre à la sortie) un jeton de la séquence de jetons à l'entrée du canal pour chaque 0 dans la séquence de bits d'écriture. Un jeton de la séquence d'entrée est transmis tel quel à la séquence de sortie lorsqu'il y a un 1 dans la séquence de lecture et un 1 dans la séquence d'écriture. Une telle modélisation s'applique à la fois aux canaux Kahn, POLL, N-REG et N-BREG.

Étant donné que tout processus RTPN est également un processus KPN, il est possible de construire, pour tout RTPN, un KPN équivalent avec les processus du RTPN et avec les canaux du RTPN représentés en tant que processus KPN. Un tel KPN équivalent est alors paramétré par un ensemble de séquences de bits. Formellement, soit $G(P, C)$ un RTPN, alors son KPN paramétré équivalent est un graphe $G'(P', C')$ tel que :

1. $P' = P \cup C \cup \{b_w, b_r\}$, où b_w et b_r sont deux processus KPN paramétrés qui génèrent respectivement un ensemble de séquences de bits d'écriture et de lecture ;
2. C' est formé par l'union de :
   (a) $\{(p, c) | p \in P, c \in C, \mathrm{tail}(c) = p\}$ (séquences de jetons d'entrée des canaux) ;
   (b) $\{(c, p) | p \in P, c \in C, \mathrm{head}(c) = p\}$ (séquences de jetons de sortie des canaux) ;
   (c) $\{(b_w, c) | c \in C\}$ (séquences de bits d'écriture des canaux) ;
   (d) $\{(b_r, c) | c \in C\}$ (séquences de bits de lecture des canaux).

La figure 4.5 illustre le KPN paramétré équivalent au RTPN de la figure 4.1 (les noeuds ω, b_w et b_r et les arcs (b_w, c) et (b_r, c) sont omis de la figure pour la simplifier).

La sémantique dénotationnelle des KPN définit le comportement d'un KPN comme son plus petit point fixe. L'existence et l'unicité d'un tel plus petit point fixe est garantie si tous les processus du KPN sont continus au sens de Scott (Kahn, 1974). Pour une valeur fixe de leurs paramètres (donc un ensemble donné de séquences de bits), les processus b_w et b_r sont trivialement continus au sens de Scott car leur sortie est un ensemble de séquences constantes.

Les canaux C sont également continus au sens de Scott étant donné que leur modélisation en processus KPN utilise le langage de (Kahn, 1974), tel qu'illustré à la figure 4.4. On obtient donc que, pour un ensemble donné de séquences de bits, le comportement (plus petit point fixe) du KPN paramétré équivalent d'un RTPN existe et est unique si tous les processus P du RTPN sont continus au sens de Scott. Cela permet de définir la sémantique dénotationnelle des RTPN comme un ensemble de KPN qui diffèrent entre eux seulement par la valeur de leurs séquences de bits. Cela permet d'isoler les effets de l'ordonnancement et des aspects dépendant du temps, tout en tenant compte de ceux-ci contrairement à un KPN classique.

```
channel(in dataIn, in writeBitstream, in readBitstream, out dataOut) {
    while(1) {
        while(readBitstream.read() == 0) {
            dataOut.write(EMPTY_TOKEN); // add absent value token
        }

        while(writeBitstream.read() == 0) {
            dataIn.read(); // discard token
        }

        dataOut.write(dataIn.read()); // transfer token as is
    }
}
```

Figure 4.4 Représentation d'un canal RTPN comme un processus KPN paramétré par des séquences de bits de lecture et d'écriture

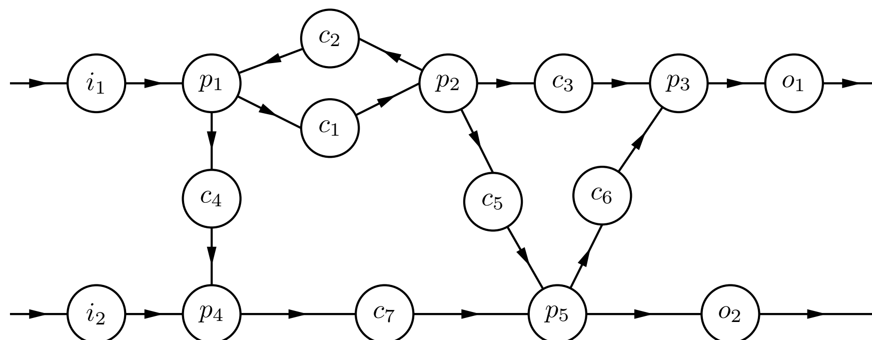

Figure 4.5 KPN paramétré équivalent au RTPN de la figure 4.1 avec $P' = \{\omega, p_1, \ldots, p_5, c_1, \ldots, c_7, i_1, i_2, o_1, o_2, b_w, b_r\}$.

4.3 Applications aux systèmes embarqués

Les RTPN, avec les canaux définis à la section 4.1.2, peuvent être utilisés pour modéliser plusieurs aspects des systèmes temps-réels, tels que les communications par mémoire partagée, les senseurs échantillonnés, les périphériques d'entrée/sortie et les contraintes temps-réel. On indique aussi comment ce modèle de calcul s'applique à la modélisation des applications embarquées réalisées avec SPACE.

4.3.1 Mémoire partagée

Un canal 1-REG peut modéliser une mémoire partagée pouvant contenir un jeton et utilisée par un producteur et un consommateur. Cette modélisation se généralise à une mémoire partagée pouvant contenir N jetons (tel un bloc de mémoire RAM) en plaçant entre le producteur et le consommateur N canaux 1-REG en parallèle.

Tel que discuté à la section 4.2.1, cette mémoire partagée se comporte alors comme un KPN classique pour un ordonnancement t si et seulement si $t\left(w_e\left(c,i\right)\right) < t\left(r_e\left(c,i\right)\right), \forall i$ et $t\left(r_e\left(c,i\right)\right) < t\left(w_e\left(c,i+1\right)\right), \forall i$. Supposons qu'on ait les relations de précédence $w_e\left(c,i\right) < r_e\left(c,i\right), \forall i$ et $r_e\left(c,i\right) < w_e\left(c,i+1\right), \forall i$ (ces relations de précédence ne seraient pas causées par l'utilisation de la mémoire partagée elle-même, mais plutôt par l'échange direct ou indirect de jetons de synchronisation entre le producteur et le consommateur via d'autres canaux). Étant donné que $x < y$ implique $t(x) < t(y)$, on obtiendrait alors que la mémoire partagée se comporte comme un canal Kahn pour tout ordonnancement t. Le comportement obtenu est alors que le producteur écrive une donnée dans le canal 1-REG, que le consommateur lise une donnée, puis que le producteur écrive une nouvelle donnée et ainsi de suite. Cela démontre que l'accès à une ressource partagée peut respecter la sémantique des KPN s'il est fait de manière disciplinée. C'est l'équivalent de l'utilisation d'une section critique pour protéger l'accès à une ressource partagée dans la programmation multi-tâches.

Les réseaux de processus Kahn sont fréquemment implémentés à l'aide de mémoire partagée. La modélisation présentée ici permet de réaliser le chemin inverse et de déterminer dans quelles circonstances une application utilisant de la mémoire partagée se comporte comme un réseau de processus Kahn. Pour simplifier la notation, on représentera une mémoire partagée par un processus spécial nommé *mem*, tel que présenté à la figure 4.6.

4.3.2 Senseur échantillonné

Les entrées d'un système temps-réel prennent souvent la forme d'un senseur échantillonné périodiquement. Lorsque le senseur est échantillonné, la valeur échantillonnée remplace la valeur précédente. Si le système n'avait pas eu le temps de lire la valeur précédente, alors

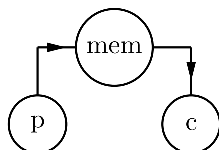

Figure 4.6 Producteur et consommateur communiquant via une mémoire partagée.

celle-ci est perdue et ne sera pas traitée. Cela correspond au comportement d'un canal 1-BREG.

Un système comprenant de telles entrées ne se comporte pas comme un KPN dans le cas général et ne peut donc pas être modélisé par un KPN. On peut cependant le modéliser à l'aide d'un RTPN qui se comporte comme un KPN classique pour un ordonnancement total t si et seulement si $t(r_e(c,i)) < t(w_e(c,i+1)), \forall i$. Cela indique qu'il se comporte comme un KPN si une donnée est lue avant que la donnée suivante ne soit écrite.

4.3.3 Périphériques d'entrée et de sortie

Les entrées d'un système temps-réel peuvent aussi comprendre des périphériques tel qu'un émetteur-récepteur asynchrone universel (UART). Un UART possède un FIFO interne de 16 octets. Si l'environnement écrit un 17e octet dans le FIFO avant que le 1er octet n'ait été lu par le système, le 16e octet sera écrasé. On peut modéliser cette entrée à l'aide d'un 16-BREG tel qu'un jeton corresponde à un octet. Ce système est alors un RTPN qui se comporte comme un KPN classique pour un ordonnancement total t si et seulement si $t(r_e(c,i)) < t(w_e(c,i+16)), \forall i$.

De la même manière, si l'UART est utilisé comme une sortie du système, alors il faut s'assurer que l'environnement ait le temps de consommer les valeurs produites par le système. On modélise de la même manière le canal de sortie comme un 16-BREG et on lui associe la même contrainte temporelle si on veut que le RTPN se comporte comme un KPN. Le même raisonnement peut s'appliquer à tout périphérique d'entrée/sortie qui utilise des FIFO d'entrée et de sortie de taille finie.

4.3.4 Contraintes temps-réel

Une caractéristique fondamentale des systèmes temps-réel est l'existence de contraintes sur le temps de production des sorties. On peut modéliser ce phénomène en utilisant un canal POLL pour la sortie et en faisant en sorte que l'environnement lise la sortie au moment où la contrainte temporelle vient à échéance. Si le système produit la sortie assez rapidement, alors le système se comporte comme un KPN classique. Sinon, l'environnement lit une valeur

absente et cela correspond à une violation de la contrainte temps-réel. Cela correspond à une contrainte $t(a(c,i)) < t(r_e(c,i)), \forall i$ sur les canaux de sortie. Les temps de consommation des sorties peuvent soit être définis de manière absolue (par exemple, une sortie est consommée à chaque seconde) ou de manière relative (par exemple, une sortie est consommée par l'environnement une seconde après que celui-ci ait produit une entrée donnée). On peut combiner les contraintes temps-réel sur le fonctionnement du système embarqué avec les contraintes temporelles sur le fonctionnement des périphériques d'entrée/sortie en utilisant un canal N-REG plutôt qu'un canal POLL pour le canal de sortie.

4.3.5 Modélisation d'une application SPACE par un RTPN

La figure 4.7 donne un exemple d'un système embarqué modélisé à l'aide d'un RTPN alors que la figure 4.8 représente la spécification exécutable associé à ce RTPN. Tel que présenté à la section 3.2.2, chaque module d'une application dans SPACE est un processus séquentiel et déterministe et on fait donc correspondre à chaque module un processus du RTPN (p_1 à p_7 sur la figure 4.7). On ajoute également un processus correspondant à l'environnement, selon le banc d'essai et les contraintes temporelles fournies avec la spécification exécutable (ce processus n'est pas illustré pour des raisons de simplification). Les sections précédentes indiquent comment modéliser la mémoire partagée (*mem*) de même que les périphériques d'entrée/sortie (canal d'entrée 16-REG) et les contraintes temps-réel (canal de sortie POLL). Afin d'éviter d'avoir à modéliser un grand nombre de canaux 1-REG, on peut supposer que les accès faits aux mémoires partagées par les processus de l'application SPACE sont faits d'une manière disciplinée qui respecte la sémantique des réseaux de processus Kahn, tel que décrit à la section 4.3.1.

Les communications entre modules dans SPACE se font au moyen d'une sémantique FIFO, comme dans un RTPN. Les canaux RTPN peuvent donc être utilisés pour modéliser les fonctions transactionnelles de communication de SPACE. Si un module x envoie des données au module y et que celui-ci les lit à l'aide d'une lecture bloquante, alors on ajoute au RTPN un canal Kahn de données qui va du processus RTPN x au processus y. Si y effectue plutôt une lecture non-bloquante, alors on ajoute un canal POLL de données qui va de x à y. Pour des raisons de simplification, on exige que les lectures effectuées par y sur les envois de données de x soient soit toujours bloquantes, soit toujours non-bloquantes. Par contre, le module x peut alterner de manière arbitraire les écritures bloquantes et les écritures non-bloquantes.

Une écriture bloquante dans SPACE ne signifie pas que l'écriture bloque seulement lorsque le canal est plein, mais plutôt qu'il s'agit d'une écriture par rendez-vous, qui bloque jusqu'à ce que la lecture ait été acquittée par le module consommateur. On modélise une telle écriture

à la figure 4.9 par l'ajout d'un canal d'acquittement de type Kahn qui va du processus y consommateur au processus x producteur. Le processus RTPN associé au module x annote les jetons écrits dans le canal de données avec un bit qui indique si l'écriture de ce jeton est bloquante ou non. Après l'écriture bloquante d'un jeton, le canal x effectue une lecture (bloquante) sur le canal d'acquittement. Lorsque le processus y reçoit et lit ce jeton, il envoie un jeton sur le canal d'acquittement si le bit annoté indique qu'il s'agit d'une écriture bloquante. Si le module y doit également envoyer des données à x, alors on ajoute un canal de données de y vers x et un canal d'acquittement de x vers y, et ainsi de suite pour chaque paire de modules qui communiquent entre eux.

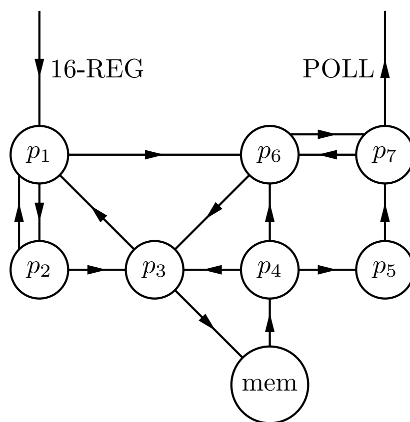

Figure 4.7 Modélisation d'un système embarqué par un RTPN.

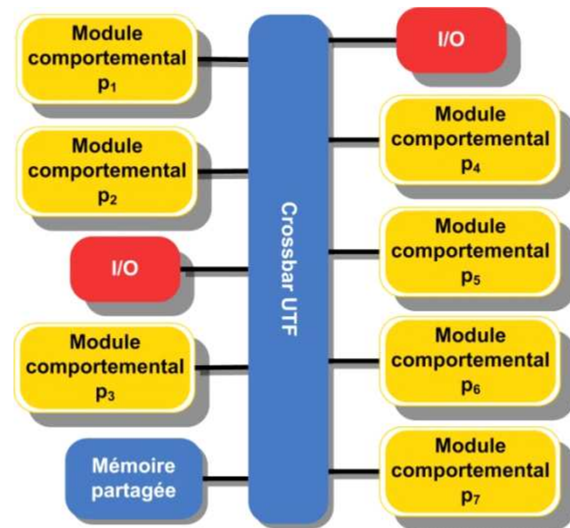

Figure 4.8 Spécification exécutable parallèle d'un système embarqué avec SPACE.

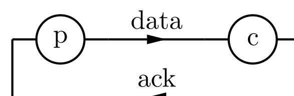

Figure 4.9 Modélisation d'une écriture rendez-vous avec un canal d'acquittement.

CHAPITRE 5

RAFFINEMENT DES COMMUNICATIONS ET SYNTHÈSE DU MATÉRIEL

Dans la méthodologie de niveau système présentée au chapitre 3, une application embarquée est une spécification exécutable composée d'un ensemble de modules SystemC (IEEE, 2005) au niveau comportemental qui s'exécutent en parallèle et qui communiquent au travers d'un ensemble de canaux abstraits de niveau transactionnel (TLM) (Cai et Gajski, 2003). Pour réaliser l'application sur une cible FPGA ou ASIC, sa spécification exécutable doit être raffinée vers une implémentation RTL et un tel raffinement est à la fois fastidieux et sujet aux erreurs lorsque fait manuellement. De plus, une telle duplication des efforts de programmation à la fois au niveau système et au niveau RTL est une barrière significative à l'adoption par l'industrie des méthodologies de conception au niveau système. Ce chapitre présente donc une méthode d'automatisation du raffinement à partir des spécifications au niveau système vers des implémentations concrètes. Comme la synthèse logicielle des modules et la génération d'un logiciel embarqué pour chaque processeur ont déjà été présentées à la section 3.2.5.1, ce chapitre se concentre sur la synthèse matérielle. Celle-ci comprend le raffinement des communications TLM vers des protocoles précis au cycle et à la broche près et la génération automatique d'un code RTL à partir de la spécification exécutable, pour une architecture donnée. Ce chapitre ne couvre pas le choix d'une architecture, qui est plutôt l'objet du chapitre 8.

La première étape du raffinement des communications est la sérialisation des types de données abstraits en une représentation précise au bit près pour leur transfert précis au cycle près. Le raffinement des communications se poursuit avec le raffinement du protocole et la synthèse de l'interface de chaque module assigné en matériel. Une implémentation RTL de chacun de ces modules est ensuite générée au moyen d'une synthèse comportementale. Ces implémentations RTL des modules sont utilisées au chapitre 7 pour caractériser le temps d'exécution et les quantités de ressources matérielles des modules assignés en matériel. Elles sont également utilisées lors de la synthèse de la plate-forme, qui permet de générer un code RTL pour une architecture donnée de l'application dans son ensemble.

Dans ce chapitre, on désigne les modèles transactionnels qui sont non temporisés ou qui sont approximativement temporisés comme des modèles TLM ; les modèles transactionnels temporisés dont les communications sont précises au cycle près sur le bus sont plutôt désignés comme des modèles BCA (Bus Cycle-Accurate).

5.1 Sérialisation des types de données

Les modèles TLM peuvent transférer en une seule transaction des structures de données de taille arbitraire entre les modules de l'application. Pour les transférer sur un bus, les communications raffinées à un niveau précis au cycle près devront diviser ces structures de données en une série de tranches de largeur égale à celle du bus. Pour les types de données primitifs et les structures de données sans pointeurs, cette section présente deux méthodes de sérialisation qui permettent l'automatisation du raffinement des communications.

5.1.1 Sérialisation triviale

Une méthode triviale de sérialisation est d'utiliser la représentation native des types des données des modules de l'application. Une seule paire de fonctions de sérialisation et de désérialisation est alors nécessaire pour l'ensemble des types de données transférés. La fonction de sérialisation convertit simplement le type de données en un tableau d'octets via une coercition (par exemple `bytes = (char*)&data ;`). Ce tableau d'octets est ensuite divisé en une série de tranches de largeur égale au bus. À l'opposé, la fonction de désérialisation convertit un tableau d'octets, qui contient une série de tranches lues sur le bus, vers le type de données initial via une coercition.

Une telle sérialisation triviale est appropriée quand tous les modules partagent une même représentation native pour chacun des types de données transférés. Cependant, il est possible que les représentations natives d'un même type de données pour deux modules différents n'aient pas la même largeur, le même alignement ou le même ordre des octets (*endianness*). Par exemple, certains modules pourraient être exécutés à un niveau comportemental sur un processeur hôte Intel qui est *little-endian* alors qu'il y aurait eu une compilation croisée des autres modules qui sont exécutés sur un ISS cible MicroBlaze (Xilinx Inc., 2005) *big-endian*. Des représentations incompatibles d'un même type de données causent alors des erreurs dans les transferts de données entre les modules. De plus, les outils de synthèse comportementale interdisent généralement les manipulations de pointeurs qui sont nécessaires à l'implémentation de la sérialisation triviale dans les modules.

5.1.2 Sérialisation standardisée

Une solution au problème de l'interopérabilité des données est de standardiser la représentation des types de données sur le réseau comme dans (Srinivasan, 1995). Tous les modules de l'application doivent alors sérialiser et désérialiser les types de données selon la représentation standardisée du réseau lorsqu'ils effectuent des opérations d'écriture ou de lecture. Cela

assure que les communications se font correctement entre les modules indépendamment de leur représentation native des types de données.

Le désavantage de la sérialisation standardisée est qu'il n'est plus possible d'utiliser une seule fonction générique de sérialisation pour gérer tous les types de données. Comme le montre le tableau 5.1, la conversion d'une valeur de 32 bits d'une représentation *big endian* à une représentation *little endian* produit des résultats différents selon le type de données de cette valeur de 32 bits. (Pour un type de données qui prend plusieurs octets, tel un `short` ou un `long`, le premier octet est l'octet de poids fort dans le cas d'une représentation *big endian* et l'octet de poids faible dans le cas d'une représentation *little endian*.)

Tableau 5.1 Conversion, pour différents types de données, de la valeur de 32 bits `0x0A0B0C0D` d'une représentation *big endian* vers une représentation *little endian*

Type de données	**Octet 0**	**Octet 1**	**Octet 2**	**Octet 3**
`long`	0x0D	0x0C	0x0B	0x0A
`short[2]`	0x0B	0x0A	0x0D	0x0C
`char[4]`	0x0A	0x0B	0x0C	0x0D

`struct {`
`short`
`char[2]`
`}`
	0x0B	0x0A	0x0C	0x0D

Il est donc nécessaire d'effectuer une introspection des types de données pour implémenter la sérialisation standardisée. L'introspection dynamique (pendant l'exécution) n'est pas utilisée, car elle n'est pas supportée par les outils de synthèse comportementale. Dans ce chapitre, l'introspection des types de données est plutôt réalisée statiquement au moment de la synthèse du matériel.

La figure 5.1 illustre le flot utilisé pour implémenter la sérialisation standardisée. D'abord, chaque module de l'application est analysé à l'aide de l'analyseur syntaxique SC2AST de la *Karlsruhe SystemC Parser Suite* (GreenSocs Ltd, 2008). Les arbres syntaxiques obtenus sont ensuite traités par un analyseur sémantique qui extrait l'ensemble des types de données transférés via les fonctions de communication de SPACE. Pour chaque type de données transféré, une paire de fonctions de sérialisation et de désérialisation standardisée sont générées. Les fonctions associées aux types de données primitifs sont générées selon la représentation du réseau. Les fonctions associées à une structure de données sont générées en appliquant à chacun de ses champs la fonction associée au type de données du champ. On suppose que les structures ne contiennent pas de pointeurs. Une structure de données peut contenir une autre structure de données, en autant que la relation d'inclusion des structures n'est ni cyclique, ni infinie. Cela assure que la structure de données peut être sérialisée en un tableau d'octets

de taille finie. Finalement, ces fonctions sont utilisées pour générer des transacteurs SystemC entre les modèles TLM et les modèles précis au cycle près. Ces transacteurs sont utilisés dans le raffinement des communications présenté à la section 5.2.

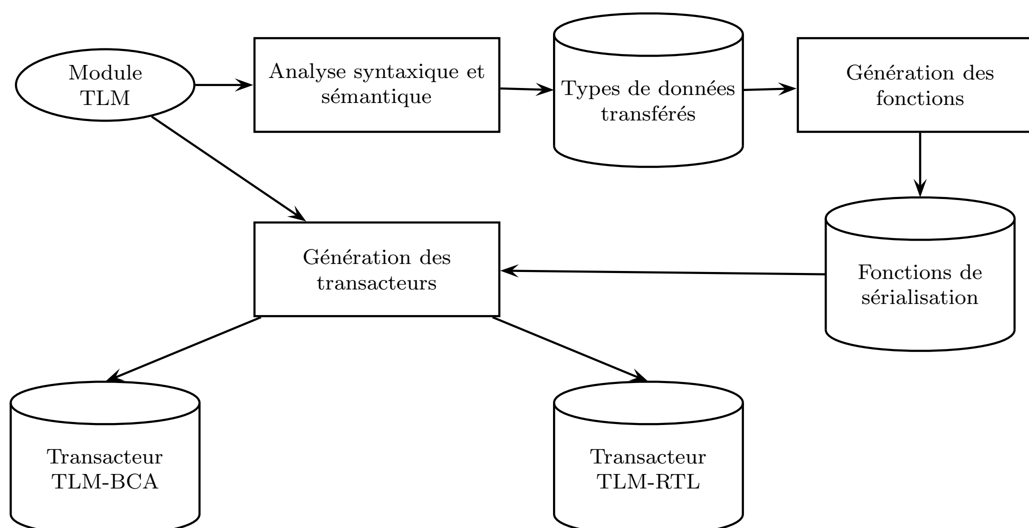

Figure 5.1 Flot de sérialisation standardisée

Sans perte de généralité, la représentation standardisée retenue pour notre sérialisation est la représentation native utilisée par le compilateur C/C++ ciblant le processeur MicroBlaze. Cela permet au logiciel embarqué sur le MicroBlaze d'utiliser une sérialisation triviale qui s'exécute rapidement tout en étant interopérable avec les modules matériels, qui effectuent une sérialisation standardisée via les transacteurs générés par notre flot.

5.2 Raffinement des communications

Cette section décrit le raffinement des fonctions de communication TLM utilisées dans notre méthodologie en des protocoles précis au cycle et à la broche près. La figure 5.2 présente une vue générale du raffinement des communications et de la synthèse du matériel décrits dans les sections 5.2 et 5.3.

5.2.1 Spécification exécutable au niveau TLM

Comme le montre la figure 5.2(a), le point de départ est une spécification exécutable au niveau TLM dans laquelle chaque module est un module comportemental qui communique avec les autres modules ou avec les périphériques seulement au travers de fonctions TLM `read()` et `write()`, tel que décrit à la section 3.2.2. Les types de données primitifs C/C++ ainsi que des structures de données sans pointeurs (définies par l'utilisateur) peuvent être

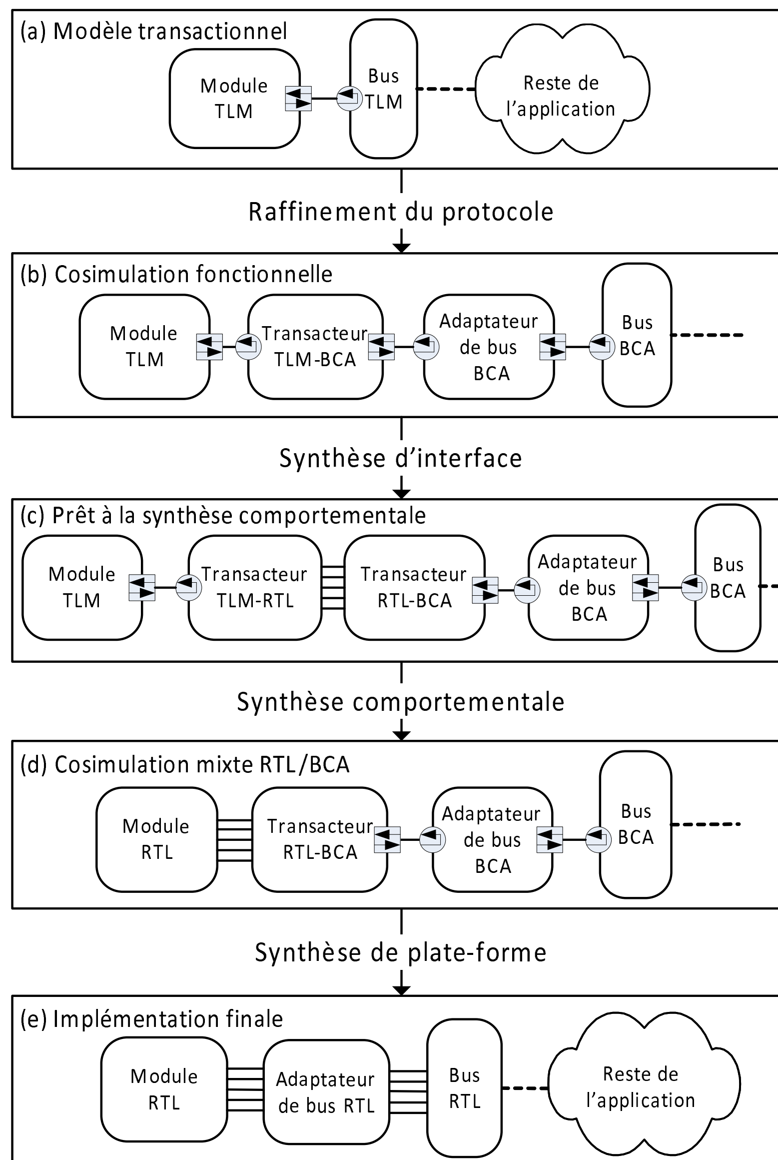

Figure 5.2 Flot automatisé pour le raffinement des communications et la synthèse du matériel

transférées avec ces fonctions de communication. Les communications entre les modules se font au travers de canaux FIFO abstraits et une sémantique bloquante ou non-bloquante peut être utilisée autant pour les écritures que pour les lectures, tel que décrit à la section 4.3.5. Les requêtes aux périphériques se font de manière synchrone. Tous les modules et les périphériques sont adressés par les fonctions de communication au moyen d'un identificateur unique. À plus bas niveau, ces identificateurs sont utilisés pour adresser à la fois les modules matériels et les modules logiciels. Un bus TLM, qui est un crossbar, route les communications entre les modules et les périphériques. À ce niveau, aucune sérialisation n'est nécessaire et les structures de données sont transférées par une copie directe de la mémoire avec `memcpy()`.

Dans la figure 5.3, on présente comme exemple le fil d'exécution d'un module comportemental simple. Ce fil d'exécution est une boucle infinie qui lit de deux autres modules deux structures de données, effectue un calcul sur celles-ci, puis écrit le résultat de son calcul vers un troisième module. Le calcul réalisé par la fonction `doProcessing()` peut être arbitrairement complexe.

```
struct s_in valueIn1, valueIn2;
struct s_out valueOut;

while(true) {
    read(SRC_ID_1, &valueIn1, sizeof(valueIn1), BLOCKING);
    read(SRC_ID_2, &valueIn2, sizeof(valueIn2), BLOCKING);
    valueOut = doProcessing(valueIn1, valueIn2);
    write(DEST_ID, &valueOut, sizeof(valueOut), NON_BLOCKING);
}
```

Figure 5.3 Exemple d'un fil d'exécution d'un module comportemental

5.2.2 Raffinement du protocole de communications

Dans cette étape, les communications TLM sont raffinées vers des communications précises au cycle près sur le bus (BCA), comme le montre la figure 5.2(b). Un modèle BCA est associé à chaque bus alloué dans l'architecture choisie. Pour chaque module de l'application, un transacteur TLM-BCA est synthétisé en utilisant la méthode présentée à la section 5.1.2. Ces transacteurs se chargent de la sérialisation et de la désérialisation des types de données et ils communiquent avec les adaptateurs de bus BCA selon un protocole générique précis au cycle près. Les adaptateurs de bus traduisent ce protocole en un protocole spécifique au bus. Les adaptateurs de bus communiquent donc avec le bus, traitent les acquittements du bus et associent une plage d'adresse sur le bus à chaque module et périphérique. Ce raffinement des communications peut ainsi être utilisé pour cibler différents protocoles de bus. Des modèles

BCA et des adaptateurs pour les bus OPB et PLB du standard CoreConnect (IBM Corp., 1999) ont été intégrés dans la plate-forme virtuelle SPACE.

Ce niveau de raffinement des communications permet également d'assigner des modules en tant que tâches logicielles s'exécutant sur un ISS connecté à un bus, ce qui permet une co-simulation fonctionnelle entre les modules qui ont été raffinés en logiciel et ceux qui sont restés à un niveau TLM (Chevalier *et al.*, 2006). Étant donné que les composants de la plate-forme virtuelle, notamment les adaptateurs de bus, gèrent les aspects des communications qui sont spécifiques à l'architecture, l'assignation d'un module donné en logiciel ou en matériel est transparent aux autres modules.

5.2.3 Synthèse d'interface

Le but de cette étape est de faire en sorte que les modules de l'application soient prêts pour la synthèse comportementale. Le protocole de communication générique entre les transacteurs TLM-BCA et les adaptateurs de bus est donc raffiné en un protocole RTL précis au cycle et à la broche près, comme le montre la figure 5.2(c). Un tel protocole RTL a été défini pour SPACE dans (Faiz, 2007). Un transacteur TLM-RTL ciblant ce protocole est généré, selon la méthode présentée à la section 5.1.2 pour chaque module assigné en matériel. La figure 5.4 définit une structure de données utilisée comme exemple. Dans la figure 5.5 et la figure 5.6, un transacteur TLM-RTL sérialise cette structure de données en une représentation précise au bit près qui est écrite cycle par cycle en tranches de 32 bits. Il est également possible de générer des transacteurs qui supportent différents types de données et différentes largeurs de bus. Un transacteur RTL-BCA, qui est un composant de la plate-forme virtuelle, se charge des communications avec l'adaptateur de bus BCA et son bus. Ce transacteur RTL-BCA renvoie aussi au transacteur TLM-RTL des informations sur le statut de l'adaptateur de bus. Dans la figure 5.5 et la figure 5.6, ce statut est utilisé pour implémenter une sémantique bloquante.

5.3 Synthèse du matériel

Après le raffinement des communications, l'application est prête pour la synthèse du matériel, qui génère, selon l'architecture choisie, une implémentation RTL de chaque module assigné en matériel, puis de l'application dans son ensemble.

5.3.1 Synthèse comportementale

Dans cette étape, chaque module TLM et son transacteur TLM-RTL sont raffinés en une implémentation RTL du module, comme le montre la figure 5.2(d). L'outil Cynthesizer de

```
struct s {
    char c1, c2;
    short s1;
    long l1;
}
```

Figure 5.4 Un exemple de structure de données à transférer

```
void write(unsigned address, s* data, unsigned size, bool blocking) {
    sc_int<32> temp;

    // Send communication parameters to adapter
    writeEnableOut.write(true);
    addressOut.write(address);
    dataLengthOut.write(size);

    // Write first 32-bit slice of struct s
    temp.range(31,24) = s->c1;
    temp.range(23,16) = s->c2;
    temp.range(15,0) = htons(s->s1);
    dataOut.write(temp);
    wait();

    // Write second 32-bit slice of struct s
    temp.range(31,0) = htonl(s->l1);
    dataOut.write(temp);
    wait();

    // Unset communication parameters
    zeroOutputSignals();

    if(blocking) {
        // Wait for the receipt of an acknowledgment
        while(writeStatusIn.read() != OK) wait();
    }
}
```

Figure 5.5 Un exemple d’une sérialisation standardisée au niveau RTL

```
bool read(unsigned address, s* data, unsigned size, bool blocking) {
    sc_int<4> status;
    sc_int<32> temp;

    // Send communication parameters to adapter
    readEnableOut.write(true);
    addressOut.write(address);
    dataLengthOut.write(size);

    // Wait for the arrival of the data if read is blocking
    do {
        wait();
        status = readStatusIn.read() ;
    } while(blocking && status != OK)

    if(status == OK) { // status != EMPTY
        // Read first 32-bit slice of struct s
        temp = dataIn.read();
        s->c1 = temp.range(31,24);
        s->c2 = temp.range(23,16);
        s->s1 = ntohs(temp.range(15,0));
        wait();

        // Read second 32-bit slice of struct s
        temp = dataIn.read();
        s->l1 = ntohl(temp.range(31,0));
        wait();

        // Unset communication parameters
        zeroOutputSignals();
    }
    return (status == OK);
}
```

Figure 5.6 Un exemple d'une désérialisation standardisée au niveau RTL

Forte Design Systems (Meredith, 2008) a été intégré dans cette méthodologie afin d'effectuer cette synthèse comportementale. La configuration de Cynthesizer, le lancement de la synthèse comportementale et la récupération des résultats de synthèse sont automatisés. Cynthesizer peut synthétiser les opérations arithmétiques et logiques de C/C++ et SystemC, le flot de contrôle, les boucles, les appels de fonction, les *templates*, les tableaux ainsi que les structures de données définies par l'utilisateur. Les pointeurs sont supportés dans la mesure où ils peuvent être statiquement déréférencés au moment de la synthèse et qu'ils ne sont pas utilisés pour manipuler la représentation interne des types de données. Les modules TLM suivent ces règles si la spécification exécutable respecte les contraintes présentées à la section 3.2.2 alors que les transacteurs TLM-RTL générés les suivent par construction.

Cynthesizer incorpore (*inline*) les fonctions et propage les constantes lors de la synthèse comportementale. Les fonctions de sérialisation générées pour les transacteurs sont donc synthétisées de manière efficace. Plusieurs options de Cynthesizer sont également activées afin d'optimiser la synthèse des modules de l'application : le déroulage de boucles, l'élimination des sous-expressions communes, l'élimination des bits non utilisés et l'aplanissement des tableaux. La synthèse comportementale peut être configurée pour minimiser soit la quantité de ressources matérielles, soit la latence.

La synthèse comportementale produit, pour chaque module de l'application, deux implémentations RTL équivalentes du module : un en SystemC et l'autre dans le HDL Verilog. La première est utilisée pour réaliser une simulation mixte du système de la figure 5.2(d) avec des modules matériels précis au cycle près et avec des modèles BCA de la plate-forme virtuelle, ce qui permet notamment une caractérisation du temps d'exécution des modules matériels tel que décrit à la section 7.2.2. La seconde est utilisée dans un flot de synthèse logique en aval de même qu'à la section 7.3.1 pour la caractérisation des ressources matérielles utilisées par les modules matériels

La méthode de synthèse matérielle présentée dans ce chapitre devrait également être applicable aux autres outils de synthèse comportementale qui prennent en entrée un code SystemC, tels que C-to-Silicon (Cadence Design Systems, Inc., 2008) et AutoESL (Zhang *et al.*, 2008). Cette méthode peut également s'appliquer à un outil de synthèse comportementale qui prend en entrée seulement un code C/C++. Elle a ainsi été appliquée à l'outil Catapult de Mentor Graphics (Bollaert, 2008) dans des travaux connexes à cette thèse. Ainsi, les transacteurs TLM-RTL sont générés pour Catapult en un C/C++ qui respecte le guide de codage de cet outil quant à la spécification des communications. Les types de données primitifs entiers précis au bit près de SystemC sont convertis en des types de données entiers équivalents (`ac_int`) fournis par Catapult (Mentor Graphics Corp., 2009). Le fil d'exécution du module est extrait de sa spécification SystemC afin de fournir à Catapult une entrée pu-

rement en C/C++. Cette adaptation de la synthèse matérielle à Catapult tire avantage du fait que SPACE encapsule la plupart des fonctionnalités SystemC par des fonctions TLM de communication.

5.3.2 Synthèse de la plate-forme

Après l'étape de la synthèse comportementale, tous les modules de l'application ont été synthétisés, selon l'architecture choisie, soit en tant que matériel RTL, soit en tant que logiciel embarqué. L'étape de la synthèse de la plate-forme, effectuée à l'aide de l'outil GenX de SPACE (Filion *et al.*, 2007; Space Codesign Systems Inc., 2008), remplace tous les modèles BCA de la plate-forme virtuelle pour les bus, les adaptateurs, les processeurs, les mémoires et les périphériques par des blocs IP au niveau RTL. Ces blocs IP sont directement fournis par la plate-forme cible, à l'exception des adaptateurs de bus qui ont été implémentés dans SPACE pour le bus OPB (IBM Corp., 1999).

Il est possible qu'un composant de la plate-forme virtuelle soit remplacé par plusieurs composants RTL. Par exemple, dans la plate-forme virtuelle, un seul pont relie deux bus quel que soit le nombre de composants qui sont assignés à ces bus, mais un pont dans une plate-forme cible FPGA Virtex de Xilinx (Xilinx Inc., 2002) peut avoir au maximum 4 esclaves. Si plus de 4 composants sont assignés à un des bus, il est donc nécessaire de remplacer le pont de la plate-forme virtuelle par plusieurs ponts en parallèle au niveau RTL.

Comme le montre la figure 5.2(e), la synthèse de la plate-forme produit une implémentation pleinement RTL du système, qui peut servir d'entrée à un flot de conception RTL pour une cible FPGA ou ASIC. La synthèse de plate-forme réalisée par GenX permet de cibler une plate-forme FPGA Virtex de Xilinx. GenX génère également un projet pour le logiciel Embedded Development Kit (EDK) de Xilinx, qui permet d'effectuer une simulation logique de l'implémentation RTL avec ModelSim (Mentor Graphics Corp., 2008) ou d'en faire une synthèse logique pour un FPGA Virtex avec XST (Xilinx Inc., 2006).

CHAPITRE 6

PROFILAGE AU NIVEAU SYSTÈME

Nous avons vu aux chapitres 3 et 5 que SPACE peut simuler la spécification exécutable ou l'implémentation d'une application embarquée sur une plate-forme virtuelle. Pour tirer pleinement bénéfice de ces simulations, il est nécessaire d'avoir des outils de profilage qui permettent d'extraire des données et des métriques de performance sur l'application embarquée qui est simulée. Ces outils de profilage peuvent être utilisés pour trouver les goulots d'étranglement dans la performance du système ou pour l'exploration architecturale, que celle-ci soit faite manuellement par un ingénieur ou automatiquement par un algorithme tel que ceux présentés au chapitre 8. Ces outils peuvent également être utilisés pour valider une méthode d'estimation de performance ou pour extraire les données nécessaires à la construction d'un modèle d'estimation de performance, tel que présenté au chapitre 7.

Dans ce chapitre, on présente comment l'exécution d'un code logiciel peut être profilée de manière non intrusive lors d'une simulation afin d'en extraire des données sur les temps de début et de fin de l'exécution de chaque fonction du code ainsi que les valeurs des paramètres et la valeur de retour de chacune de ces fonctions. Cette méthode de profilage non-intrusif d'un code logiciel est appliquée à l'instrumentation de la plate-forme virtuelle de SPACE afin de réaliser un profilage au niveau système. Ainsi, les modèles de processeurs (ISS) sont instrumentés afin de permettre le profilage des ISR, des changements de contexte entre les modules logiciels et des appels aux fonctions de communication de l'API logicielle. Étant donné que les modules matériels sont également instrumentés, cela permet d'extraire des métriques sur le temps d'exécution des modules logiciels et matériels ainsi que sur les communications entre tous les modules, peu importe que ces communications se fassent entre modules matériels, entre modules logiciels sur un même processeur, entre modules logiciels sur deux processeurs différents ou entre un module matériel et un module logiciel. Cette capacité d'extraire les mêmes types de métriques pour l'ensemble des modules qu'ils se trouvent en matériel ou en logiciel est ce qui fait la spécificité du profilage au niveau système. L'instrumentation des bus et des mémoires, en plus de celle des processeurs, permet aussi de recueillir des métriques sur l'utilisation des bus, des mémoires et des processeurs.

6.1 Profilage non-intrusif de code logiciel

Le premier objectif du profilage de code logiciel présenté dans cette section est d'extraire des données sur le déroulement et la performance de l'exécution du code logiciel. En particulier, ce profilage extrait différentes informations à chaque fois que l'exécution du code logiciel arrive au début ou à la fin d'une fonction profilée. À chaque fois que l'exécution arrive au début d'une fonction profilée, le profilage produit un enregistrement de données qui contient le nom de la fonction, le temps de cet évènement et les valeurs des arguments passés en paramètre à la fonction. De manière analogue, à chaque fois que l'exécution arrive à la fin d'une fonction profilée, le profilage produit un enregistrement de données qui contient le nom de la fonction, le temps de cet évènement et la valeur de retour de la fonction. De plus, le profilage peut ajouter à chacun de ces enregistrements différentes informations complémentaires : le contenu, au moment où l'évènement se produit, d'un ou plusieurs registres du processeur, variables globales du code logiciel ou adresses mémoire.

Le deuxième objectif du profilage de code logiciel présenté dans cette section est d'être non-intrusif, c'est-à-dire de ne perturber ni la fonctionnalité ni la performance de l'exécution de ce même code logiciel. En d'autres termes, du point du vue du code logiciel, les opérations nécessaires au profilage doivent s'exécuter en un temps zéro. Cela implique qu'il n'est pas possible de modifier ce code logiciel pour y ajouter des instructions spéciales de profilage, car ces nouvelles instructions prendraient un temps non-nul pour s'exécuter sur le processeur. Le temps pris pour exécuter le code logiciel avec ces instructions ajoutées différerait alors du temps pris pour exécuter le code logiciel sans ces instructions.

Il existe une solution dans le cas où le code logiciel est exécuté sur un ISS, par exemple dans le cadre d'une simulation SystemC (IEEE, 2005). Cette solution repose sur la différence entre le temps pris par la simulation (WCT : Wall Clock Time) et le temps interne à la simulation. Ainsi, toutes les opérations dans la simulation, incluant éventuellement les opérations relatives au profilage, s'effectuent en un temps non-nul selon le référentiel du WCT. Par contre, ces mêmes opérations peuvent s'effectuer en un temps nul selon le référentiel du temps interne à la simulation si elles ne font pas avancer l'horloge interne de simulation. Dans le contexte d'une simulation SystemC, cela signifie qu'elles ne doivent pas causer un appel à la fonction `wait`. Les métriques de performance qui importent lorsqu'on simule un code logiciel sur un ISS sont les métriques relatives au temps interne à la simulation. Dans une telle simulation, il est donc possible de profiler le code logiciel de manière non-intrusive en autant que les opérations de profilage ne modifient pas l'horloge interne de la simulation. Il demeure donc impossible de modifier le code logiciel lui-même pour y ajouter des instructions de profilage : l'ISS prendrait un certain nombre de cycles à l'intérieur de la simulation pour exécuter ces

instructions et cela modifierait donc le temps interne de simulation. Par contre, il est alors possible de profiler le code logiciel de manière non-intrusive si les opérations de profilage se trouvent à l'extérieur du code logiciel simulé sur l'ISS.

6.1.1 Fonctionnement général du profileur d'ISS

Un profileur d'ISS a été implémenté pour extraire non-intrusivement des données sur le déroulement et la performance de l'exécution du code logiciel. Comme son nom l'indique, le profileur d'ISS vient se greffer à un ISS pour profiler de manière non-intrusive le code logiciel qui est exécuté par cet ISS. L'implémentation du profileur d'ISS repose sur les prémisses suivantes :

1. Le profileur d'ISS a à profiler un seul fil d'exécution à la fois. Pour simuler un système multi-processeur, il doit y avoir un profileur d'ISS par processeur.
2. L'horloge de l'ISS est synchronisée avec celle de la plate-forme virtuelle.
3. L'ISS ou la plate-forme virtuelle permet de non-intrusivement :
   (a) Appeler le profileur d'ISS lorsque le PC de l'ISS atteint une adresse donnée, puis reprendre l'exécution du code logiciel ;
   (b) Extraire le contenu d'un registre de l'ISS ou d'une adresse mémoire.

En d'autres termes, il doit être possible au profileur d'ISS d'accéder directement ou indirectement aux registres et à la mémoire de l'ISS, incluant le PC. La prémisse 3a est conceptuellement équivalente à l'ajout, pour une adresse donnée, d'un point d'arrêt matériel, qui ne modifie pas le code logiciel.

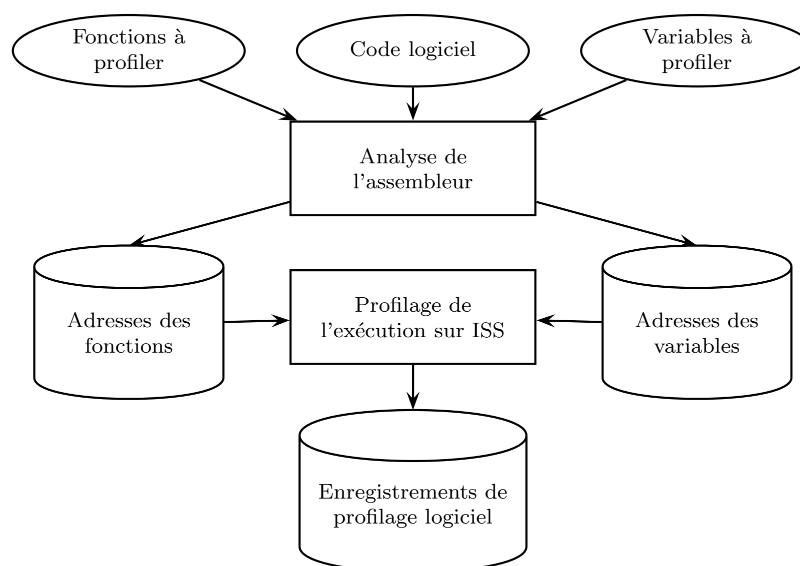

Figure 6.1 Fonctionnement général du profileur d'ISS

Le profileur d'ISS automatise le flot illustré à la figure 6.1 pour un ensemble donné de registres, d'adresses mémoire, de variables globales et de fonctions à profiler. Ainsi, il commence par désassembler le code logiciel à profiler et par extraire la taille et l'adresse mémoire de chaque variable globale à profiler. Ces adresses mémoires sont ajoutées aux adresses à profiler. Il extrait également du code désassemblé les adresses de début et de fin de chaque fonction à profiler et ajoute un point d'arrêt à chacune de ces adresses. Il est à noter que, bien que toute fonction ait un seul point d'entrée, il est possible qu'elle ait plusieurs points de sortie, car rien ne garantit que le programmeur ou le compilateur du code logiciel utilisent un seul point de sortie par fonction. La simulation démarre ensuite et le profileur d'ISS est appelé lorsque l'exécution du code logiciel sur l'ISS arrive à un point d'arrêt. Le profileur d'ISS produit alors un enregistrement de données avec les informations suivantes : le nom de la fonction correspondant à ce point d'arrêt, la valeur courante du temps interne à la simulation SystemC et la valeur courante de chaque registre ou adresse mémoire profilé. Si le point d'arrêt correspond au début d'une fonction, le profileur d'ISS ajoute à cet enregistrement les valeurs des arguments passés en paramètre à la fonction. S'il s'agit plutôt d'une fin de fonction, le profileur d'ISS y ajoute alors la valeur de retour de la fonction. Le profileur d'ISS laisse ensuite l'exécution du code logiciel reprendre son cours jusqu'au prochain point d'arrêt et ainsi de suite jusqu'à la fin de la simulation.

6.1.2 Implémentation du profileur d'ISS

En plus des opérations élémentaires qu'on suppose être fournies par l'ISS ou la plateforme virtuelle à la section 6.1.1 (gestion des points d'arrêts, lecture d'une adresse mémoire et lecture d'un registre de l'ISS), plusieurs autres opérations utilisées par le profileur d'ISS dépendent du processeur dont on profile le code logiciel :

1. Désassembler le code logiciel ;
2. Trouver la taille et l'adresse des variables globales dans le code désassemblé ;
3. Trouver l'adresse de début d'une fonction dans le code désassemblé ;
4. Trouver l'adresse des points de sortie d'une fonction dans le code désassemblé ;
5. Extraire les valeurs des arguments passés en paramètres à une fonction ;
6. Extraire la valeur de retour d'une fonction.

L'implémentation du profileur d'ISS suppose qu'il existe une suite d'outils GNU (Free Software Foundation, 2010) ciblant le processeur dont on profile le code logiciel. Plus précisément, avant le début de la simulation, le profileur d'ISS lance le débogueur croisé GDB (Stallman *et al.*, 2002) pour désassembler et analyser le code logiciel afin de réaliser les opérations 1, 2 et 3. Ainsi, une fois que le débogueur croisé GDB est lancé, le profileur d'ISS récupère l'adresse et la taille de chaque variable globale profilée `var` en exécutant respec-

tivement les commandes `print/x &var` et `print/x sizeof(var)` dans GDB. De la même manière, le profileur d'ISS obtient l'adresse de début de chaque fonction profilée `fct` en ajoutant temporairement un point d'arrêt avec la commande `break 'fct'` et en récupérant l'adresse à laquelle le point d'arrêt a été ajouté par GDB. Il est à noter que cette utilisation des points d'arrêt de GDB avant la simulation n'implique pas nécessairement que GDB sera également utilisé pour gérer les points d'arrêt lors de la simulation. En effet, on peut préférer manipuler directement l'ISS pour la gestion des points d'arrêt, plutôt que de passer par GDB, pour des raisons de performance (en terme de WCT), comme le montrent les résultats présentés à la section 9.4.

Cette utilisation de la suite d'outils croisés GNU simplifie l'implémentation du profileur d'ISS. Cependant, cette suite d'outils ne permet pas d'identifier les adresses des points de sortie d'une fonction dans le code désassemblé. Pour trouver ces points de sortie, le profileur d'ISS demande donc à GDB de désassembler le code de la fonction puis examine une par une les instructions en code assembleur de cette fonction. Si l'opcode de l'instruction est un opcode de sortie, alors le profileur d'ISS ajoute l'adresse de cette instruction à la liste des points de sortie de la fonction. L'identité des opcodes qui correspondent à des points de sortie dépend du jeu d'instructions du processeur cible.

Les autres opérations utilisées par le profileur d'ISS sont implémentées soit au travers du débogueur croisé GDB, soit en accédant directement à l'ISS, à ses registres et à sa mémoire.

6.1.2.1 Implémentation du profileur d'ISS pour le MicroBlaze

Le profileur d'ISS a été implémenté pour le processeur MicroBlaze de Xilinx (Xilinx Inc., 2005). Le profileur d'ISS MicroBlaze délègue au débogueur croisé GDB le désassemblage du code logiciel, l'extraction de la taille et de l'adresse des variables profilées ainsi que de l'adresse de début des fonctions profilées. Aussi, ce profileur d'ISS trouve l'ensemble des points de sortie de chacune de ces fonction en considérant comme des points de sortie les instructions avec l'opcode `rtid` ou `rtsd`, tel que défini dans le jeu d'instructions du MicroBlaze.

L'ISS MicroBlaze implémenté dans la plate-forme virtuelle de SPACE permet au profileur d'ISS d'accéder directement à ses registres et à sa mémoire pendant la simulation. Cela signifie que ces accès ne passent pas par le bus et prennent un temps nul dans le référentiel du temps interne à la simulation. Ainsi, le profileur d'ISS MicroBlaze peut lire la valeur courante d'une variable profilée en lisant la valeur de la mémoire de l'ISS à l'adresse de la variable. Selon l'interface binaire (ABI) du MicroBlaze, les 6 registres R5 à R10 du MicroBlaze sont utilisés pour passer des arguments en paramètre à une fonction alors que le registre R3 est utilisé pour la valeur de retour d'une fonction. Ainsi, le profileur d'ISS MicroBlaze extrait les valeurs des registres R5 à R10 à chaque fois que l'exécution du code logiciel atteint le

début d'une fonction et il extrait la valeur du registre R3 à chaque fois que l'exécution arrive à la fin d'une fonction. Cette implémentation extrait les valeurs des 6 premiers paramètres de chaque fonction profilée et suppose qu'elles utilisent le passage par valeur pour leurs paramètres et leur valeur de retour. Cette implémentation pourrait être étendue d'abord pour supporter tous les paramètres des fonctions ayant plus de 6 paramètres en récupérant les valeurs des paramètres supplémentaires sur la pile. Une autre extension permettrait de supporter le passage par adresse en déréférençant les pointeurs passés en paramètre ou comme valeur de retour. Finalement, l'ISS MicroBlaze implémenté dans SPACE contient tous les registres présents dans le processeur MicroBlaze en plus de registres virtuels qui comptent le nombre d'instructions de contrôle, d'accès à la mémoire et d'opérations arithmétiques et logiques exécutées depuis le démarrage du processeur. Ces registres virtuels sont profilés par le profileur d'ISS pour fournir plus d'informations sur le type d'instructions exécutées par chaque fonction.

Le profileur d'ISS MicroBlaze utilise les fonctionnalités offertes par cet ISS pour ajouter des points d'arrêt et être appelé lorsqu'un point d'arrêt est atteint. Ainsi, l'ISS MicroBlaze surveille la valeur du PC pour voir si elle est égale à l'adresse d'un point d'arrêt. Si c'est le cas, l'ISS MicroBlaze appelle la fonction de rappel (*callback*) associée au point d'arrêt, soit le profileur d'ISS dans ce cas-ci. Le profileur d'ISS MicroBlaze traite alors le point d'arrêt en déterminant au début ou à la fin de quelle fonction correspond ce point d'arrêt. Après ce traitement, le profileur d'ISS MicroBlaze demande à l'ISS de poursuivre l'exécution du code logiciel. Étant donné que le traitement du point d'arrêt effectué par le profileur d'ISS demande un temps nul dans le référentiel du temps interne à la simulation, la gestion de ces points d'arrêt ne perturbe ni la fonctionnalité ni la performance du code logiciel profilé.

6.1.2.2 Implémentation générique avec GDB

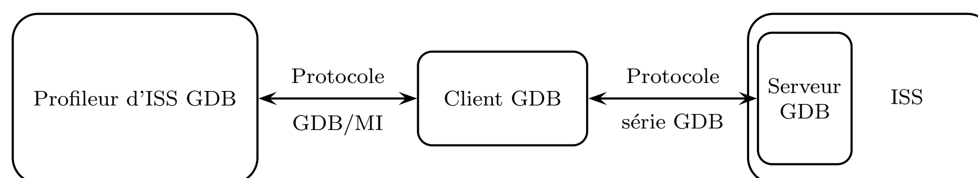

Figure 6.2 Implémentation générique du profileur d'ISS avec client/serveur GDB

Une version générique du profileur d'ISS a été implémentée. Cette implémentation générique délègue au débogueur croisé GDB non seulement les tâches d'analyse du code logiciel préalables à la simulation, mais également les tâches qui se déroulent lors de la simulation comme la gestion des points d'arrêt et l'extraction des valeurs des arguments passés en pa-

ramètre. On l'appelle donc le profileur d'ISS GDB.

Tel qu'illustré à la figure 6.2, le profileur d'ISS GDB communiqué avec le débogueur croisé GDB au moyen de l'interface de communication GDB/MI (Stallman *et al.*, 2002). Le débogueur croisé GDB agit comme client GDB et il communique à son tour, au moyen du protocole série de GDB, avec le serveur GDB qui se trouve au sein de l'ISS. Le serveur GDB a un accès direct à la mémoire et aux registres de l'ISS et on suppose que l'ISS et son serveur GDB supportent les points d'arrêt matériels pour que leur utilisation par le profileur d'ISS soit non-intrusive. Dans le contexte du processeur MicroBlaze, le serveur GDB se trouve dans une situation analogue au profileur d'ISS MicroBlaze présenté à la section 6.1.2.1. L'avantage de passer par GDB est qu'on peut s'attendre à ce que le serveur GDB ait déjà été implémenté pour l'ISS, indépendamment du profilage, afin de permettre le débogage du code logiciel exécuté par l'ISS.

Ainsi, le profileur d'ISS GDB envoie au client GDB différentes commandes du protocole GDB/MI telles que `-data-evaluate-expression`, `-data-read-memory` et `-data-list-register-values` pour lire la valeur courante d'une variable, d'un registre ou d'une adresse mémoire. Le client GDB demande alors au serveur GDB de lire le registre ou l'adresse mémoire appropriée et le serveur GDB fait les accès pertinents pour répondre à cette requête. Le serveur GDB renvoie la valeur lue au client GDB, qui renvoie alors au profileur d'ISS GDB une réponse textuelle suivant le protocole GDB/MI. Cette réponse est une chaîne de caractères semblable à `^done,value="Y"` et le profileur d'ISS GDB en extrait la valeur `Y`, qui correspond à la valeur lue.

De manière analogue, le profileur d'ISS GDB insère des points d'arrêt au début et aux points de sortie de chaque fonction profilée en envoyant au client GDB la commande `-break-insert`. Après l'insertion de chaque point d'arrêt, le client GDB renvoie un identificateur unique correspondant au point d'arrêt qui vient d'être inséré. Lorsque l'exécution atteint un point d'arrêt, le client GDB notifie alors le profileur d'ISS GDB en lui envoyant un message textuel, selon le protocole GDB/MI, qui contient l'identificateur unique du point d'arrêt atteint. Cela permet au profileur d'ISS GDB de déterminer au début ou à la fin de quelle fonction ce point d'arrêt correspond. Le message textuel envoyé par le client GDB contient également les valeurs des arguments passés en paramètre à la fonction courante. Ces valeurs sont donc extraites du message textuel par le profileur d'ISS GDB si le point d'arrêt correspond au début d'une fonction profilée.

Lorsqu'un point d'arrêt est atteint à un point de sortie d'une fonction, le profileur d'ISS GDB obtient la valeur de retour de la fonction en envoyant la commande `-exec-finish` au client GDB. La fonction termine alors son exécution et le client GDB renvoie au profileur d'ISS GDB un message textuel qui contient la valeur de retour de la fonction. L'exécution du

code logiciel reprend alors son cours après que le profileur d'ISS GDB ait envoyé la commande `-exec-continue` au client GDB.

Le profileur d'ISS GDB est caractérisé par une grande généricité et un grand potentiel de réutilisation, étant donné que le porter vers un processeur cible implique seulement de spécifier la liste des opcodes correspondant à un point de sortie de fonction dans le jeu d'instructions de ce processeur.

6.1.3 Profilage du RTOS et de l'API logicielle de SPACE

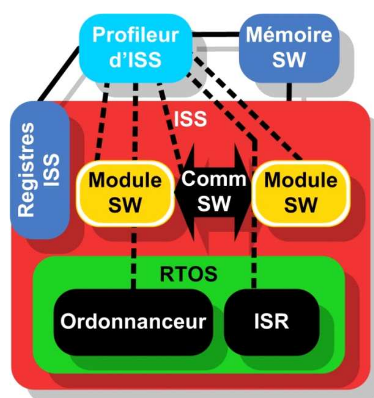

Figure 6.3 Le profileur d'ISS SPACE surveille la mémoire et les registres pour détecter les communications et les changements de contexte

Le profilage non-intrusif du code logiciel est appliqué à la plate-forme virtuelle de SPACE pour profiler les évènements relatifs aux opérations du RTOS et des modules implémentés en logiciel. La figure 6.3 montre que le profileur d'ISS SPACE utilise son accès (direct ou indirect) aux registres et à la mémoire de l'ISS pour extraire des informations sur l'ordonnancement des tâches logicielles, l'exécution des ISR et les communications faites par les modules logiciels. Ainsi, le profileur d'ISS SPACE utilise les mécanismes décrits à la section 6.1.2 pour profiler les fonctions de communications de la plate-forme SPACE (qui sont indépendantes à la fois du processeur et du RTOS utilisé), les ISR et les fonctions de changement de contexte du RTOS. Le profileur d'ISS SPACE profile également la variable globale du RTOS qui contient l'identificateur de la tâche courante sur le processeur.

6.1.3.1 Algorithmes du profileur d'ISS SPACE

On présente ici comment le profileur d'ISS SPACE traite les enregistrements de profilage obtenus, qui ont été décrits à la section 6.1.1. Le profilage des ISR et des changements de contexte sert à déterminer quand les modules cessent ou reprennent leur exécution. Ainsi,

un enregistrement associé à la fin d'une ISR ou à la fin d'une fonction de changement de contexte est considéré comme un évènement signalant le début d'une période d'exécution d'une tâche logicielle. On suppose que le RTOS peut exécuter une tâche inactive si toutes les tâches associées à des modules logiciels sont bloquées. Le profileur d'ISS SPACE extrait le temps t de cet enregistrement ainsi que l'identificateur de la tâche courante du RTOS. Si cet identificateur est associé à un module de l'application, alors le profileur d'ISS SPACE consigne un évènement de début de période d'exécution pour le module courant au temps t. Sinon, il consigne un évènement de début de période d'exécution pour la tâche inactive (*idle*) au temps t. Un évènement de fin d'ISR ou de fin de changement de contexte est également consigné au temps t, selon le cas.

À l'opposé, un enregistrement associé au début d'une ISR ou au début d'une fonction de changement de contexte est considéré comme un évènement signalant la fin d'une période d'exécution d'une tâche logicielle. Le profileur d'ISS extrait le temps t de cet enregistrement et trouve la tâche qui a cessé son exécution, soit la dernière tâche à avoir commencé son exécution avant le temps t. Un évènement de fin d'exécution d'un module de l'application ou de la tâche inactive est alors consigné au temps t, selon le cas. Un évènement de fin d'ISR ou de fin de changement de contexte est également consigné au temps t, selon le cas.

Le traitement de ces enregistrements permet de déterminer, pour tous les temps de la simulation, ce que le processeur est en train d'exécuter, soit une ISR, soit une fonction de changement de contexte, soit la tâche inactive, soit une tâche associée à un module logiciel. Étant donné qu'on a supposé à la section 6.1.1 que le processeur exécute un seul fil d'exécution à la fois, ces possibilités sont mutuellement exclusives pour tout temps donné de la simulation. Pour tous les temps où le processeur exécute une tâche associée à un module logiciel, le profileur d'ISS SPACE peut également déterminer quel module logiciel détient alors le processeur.

D'un enregistrement associé au début d'une fonction de communication SPACE, le profileur d'ISS SPACE extrait un temps t, le type de communication SPACE (écriture à un module, lecture d'un module, écriture à un périphérique ou lecture d'un périphérique) de même que ses paramètres (identificateur du module distant, taille, caractère bloquant). Un évènement de début de communication est alors consigné pour le temps t avec ces paramètres et avec l'identificateur du module qui appelle cette fonction de communication, soit le module qui s'exécute sur l'ISS au temps t. Cela permet donc de déterminer quels sont les modules qui communiquent entre eux.

Le profileur d'ISS SPACE extrait le temps t et la valeur de retour d'un enregistrement associé à la fin d'une fonction de communication. Un évènement de fin de communication est consigné au temps t avec cette valeur de retour, qui indique si la communication a réussi ou

échoué. Le profileur d'ISS SPACE trouve l'évènement associé au début de cette communication, qui est la dernière communication qui a été commencée avant le temps t par le module qui s'exécute sur l'ISS au temps t. Cette paire d'évènements de début et de fin correspond à une communication complétée.

Il y a plusieurs éléments des algorithmes du profileur d'ISS SPACE qui dépendent du RTOS utilisé par le code logiciel : l'identité des variables et des fonctions associées au traitement des interruptions et des changements de contexte, la manière d'extraire l'identificateur spécifique au RTOS pour une tâche donnée et la manière d'associer cet identificateur à l'identificateur SPACE d'un module. La section suivante explique comment le profileur d'ISS SPACE a été implémenté pour le RTOS μCOS/II (Labrosse, 2002).

6.1.3.2 Implémentation pour μCOS/II

Les différentes tâches logicielles ordonnancées par le RTOS μCOS/II ont chacune une priorité unique et fixe, qui leur sert aussi d'identificateur. La variable globale `OSPrioHighRdy` contient la priorité de la tâche présentement exécutée par le RTOS et est donc profilée par le profileur d'ISS SPACE pour μCOS/II. Avant le début de la simulation, le profileur construit, à partir des métadonnées sur l'architecture à simuler fournies par Space Studio (Filion *et al.*, 2007), un tableau associatif qui fait le lien entre la priorité d'une tâche logicielle et l'identificateur SPACE d'un module implémenté en logiciel. Le profileur peut ainsi savoir quelle est la priorité de la tâche qui s'exécute à un temps donné et, si cette priorité est associée à un module, quel est l'identificateur SPACE du module qui s'exécute à ce même temps donné.

Le profileur d'ISS SPACE pour μCOS/II profile également différentes fonctions de traitement d'interruption et de changement de contexte :

1. `OSCtxSw` (changement de contexte à partir d'une tâche logicielle)
2. `OSIntCtxSw` (changement de contexte à partir d'une interruption)
3. `OSStartHighRdy` (début de l'exécution du RTOS)
4. La routine de traitement d'interruption, dont le nom dépend du processeur.

Les enregistrements associés à la fin de chacune de ces fonctions sont considérés comme des évènements signalant le début d'une période d'exécution d'une tâche logicielle, tel que décrit à la section 6.1.3.1. De la même manière, les enregistrements associés au début de la fonction `OSCtxSw` et de la routine de traitement d'interruption sont considérés comme des évènements signalant la fin d'une période d'exécution d'une tâche logicielle. Par contre, les enregistrements correspondant au début de la fonction `OSStartHighRdy` sont ignorés, car aucune tâche logicielle ne s'exécute avant que le RTOS ne démarre son exécution. Les enregistrements associés au début de la fonction `OSIntCtxSw` sont également ignorés parce

que cette fonction est seulement appelée à partir de la routine de traitement d'interruption et que la tâche logicielle a donc déjà été interrompue. Cela complète l'implémentation du profileur d'ISS SPACE pour μCOS/II.

6.1.4 Profilage exhaustif du code logiciel

Le profileur d'ISS présenté aux sections 6.1.1 et 6.1.2 peut également être utilisé pour profiler exhaustivement le code logiciel, c'est-à-dire profiler, pour toutes les fonctions contenues dans le code logiciel, toutes les occurrences d'une entrée ou d'une de sortie de fonction. En plus des informations sur les temps de début et de fin, les valeurs des arguments passés en paramètre et les valeurs de retour des fonctions, ce profilage exhaustif peut profiler l'identité de la tâche courante qui s'exécute sur le processeur en utilisant les méthodes décrites à la section 6.1.3. Un tel profileur exhaustif a été implémenté et est utilisé à la section 7.2.3.4 pour la caractérisation des paramètres de performance du RTOS et de l'API logicielle de SPACE.

Par contre, pour le profilage du RTOS et de l'API logicielle dans le cadre du profilage au niveau système présenté à la section 6.2, on préfère profiler seulement les fonctions énumérées à la section 6.1.3 afin de minimiser le nombre de points d'arrêt et l'impact sur le WCT de la simulation.

6.2 Profilage au niveau système dans SPACE

Le profilage au niveau système a pour objectif de fournir des informations détaillées sur la performance et les opérations du système embarqué simulé avec SPACE ainsi que les métriques de performance suivantes :

1. Le temps d'exécution de chaque module du système, que ce dernier se trouve en matériel ou en logiciel ;
2. Le temps et le volume de communications entre chaque paire de modules, que ceux-ci se trouvent tous les deux en matériel, tous les deux en logiciel (sur le même processeur ou non) ou que l'un soit en matériel et l'autre en logiciel ;
3. Le nombre et la répartition des accès mémoire pour chaque composant de mémoire et pour chaque module du système ;
4. Le taux d'utilisation de chaque bus et processeur du système.

Ce profilage au niveau système est réalisé selon l'architecture générale présentée à la figure 6.4. Ainsi, la plate-forme virtuelle de SPACE a été instrumentée afin que des notifications soient envoyées à un module central de profilage SystemC lors de la simulation du système embarqué qu'on veut profiler. Ce module traite et assemble les différents évènements de manière à ce qu'un générateur de métriques puisse en extraire des métriques de performances.

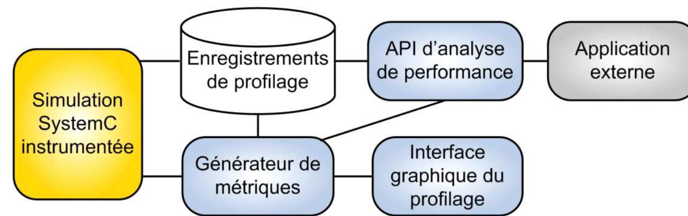

Figure 6.4 Architecture générale du co-profilage

Les informations et les métriques extraites peuvent ensuite soit être analysées par un autre programme (comme les algorithmes de caractérisation présentés au chapitre 7), soit visualisées dans une interface graphique.

6.2.1 Instrumentation de la plate-forme SPACE

La figure 6.5 présente un aperçu général de l'instrumentation de la plate-forme virtuelle SPACE. Ainsi, nous avons implémenté des macros génériques d'instrumentation qui peuvent être rapidement insérées dans les composants de la plate-forme virtuelle. Ces macros collectent des informations sur les évènements (début et fin d'une communication, d'un accès mémoire, d'un calcul, etc.) qui se produisent dans la simulation et les transmettent au module central de profilage, qui coordonne le profilage du système.

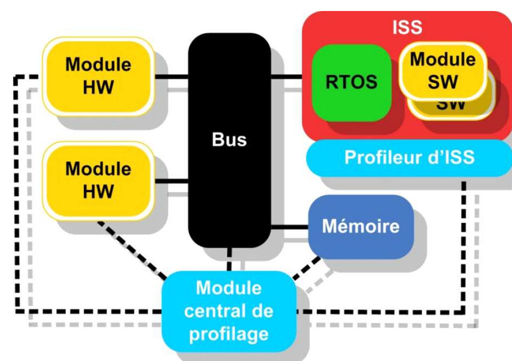

Figure 6.5 Instrumentation de la plate-forme SPACE. Les lignes pointillées représentent les accès faits par le profilage

Ces macros ont directement été insérées dans différents modèles transactionnels de composants ciblant la plate-forme cible FPGA Virtex de Xilinx (Xilinx Inc., 2002), soit des modèles transactionnels de bus OPB, de bus PLB et de mémoires RAM avec interface OPB ou PLB (IBM Corp., 1999). Ces macros ont également été insérées dans le crossbar TLM et le modèle comportemental de mémoire RAM qu'on trouve au niveau Elix de SPACE. Finalement, ces macros sont aussi appelées par les fonctions de communication utilisées par les modules à

haut niveau de la spécification exécutable, c'est-à-dire les modules qui n'ont ni été synthétisés en une implémentation RTL selon les méthodes présentées au chapitre 5, ni compilés en un code logiciel. Dans tous ces cas, il est possible de modifier le code SystemC de ces modèles en y insérant les macros d'instrumentation, étant donné que ce code fait partie de la plate-forme virtuelle de SPACE. L'appel de ces macros par ces modèles ne perturbe pas non plus le déroulement ou la performance de la simulation, dans le référentiel du temps interne à la simulation, car ces macros ne font pas appel à la fonction `wait` de SystemC et elles s'exécutent donc en un temps nul selon ce référentiel.

Cependant, les implémentations RTL des modules synthétisés selon les méthodes présentées au chapitre 5 ne sont pas directement instrumentées. En effet, le code RTL généré par les outils de synthèse comportementale est à un niveau bien plus bas que les macros d'instrumentation, qui sont à un niveau transactionnel, et le contenu de ce code dépend du module fourni en entrée par l'utilisateur et n'est donc pas sous le contrôle direct de SPACE. Ces macros ne peuvent donc pas être insérées à l'avance dans ce code et développer un algorithme qui tenterait d'insérer automatiquement des macros transactionnelles dans un code RTL semble hasardeux. Ces macros d'instrumentation ont donc plutôt été insérées dans les transacteurs RTL-BCA, qui font partie de la plate-forme virtuelle. Tel que décrit à la section 5.2.3, un tel transacteur est associé à chaque instance d'un module RTL et cela permet donc de profiler les communications faites par ces modules. Ces transacteurs profilent aussi indirectement le temps d'exécution des modules RTL, car on suppose qu'un module RTL est en train d'effectuer des calculs si il n'est pas en train d'effectuer une communication ou d'attendre une réponse.

Les macros d'instrumentation ne sont pas insérées dans les modules logiciels ou le RTOS s'exécutant sur un ISS pour éviter de perturber la simulation selon le référentiel du temps interne à la simulation. Pour les profiler, on transmet plutôt au module central de profilage les évènements d'exécution et de communication extraits de manière non intrusive par le profileur d'ISS SPACE décrit à la section 6.1.3. Si l'implémentation à profiler contient plusieurs processeurs, alors un profileur d'ISS SPACE est instancié pour chaque processeur.

À partir des données extraites par les macros d'instrumentation et le profileur d'ISS SPACE, le module central de profilage produit les enregistrements décrits à la section 6.2.2 et qui portent sur les opérations des modules, des bus, des mémoires et des processeurs. L'instrumentation et donc le profilage de l'application simulée s'ajustent automatiquement si l'implémentation de cette application est modifiée entre deux simulations, par exemple si le partitionnement logiciel/matériel ou le nombre de processeurs est modifié. De plus, le profilage de l'application peut extraire des métriques de performance sur l'application et ses modules qui peuvent être comparées pour les différentes implémentations. On considère donc

qu'il s'agit d'un profilage au niveau système.

6.2.2 Informations et métriques extraites du profilage

Le profilage au niveau système fournit comme résultat un ensemble d'enregistrements qui peuvent être analysés pour en extraire des métriques de performance. Cette section présente les différents types d'enregistrements, les métriques qui en sont extraites et l'interface graphique utilisée pour les afficher. Une version précédente de l'interface graphique d'affichage des métriques a été présentée dans (Moss *et al.*, 2007).

6.2.2.1 Exécution des modules et du RTOS

Le profilage au niveau système génère un enregistrement d'exécution pour chaque période d'exécution ininterrompue d'un module matériel, d'un module logiciel ou du RTOS. Chacun de ces enregistrements contient les moments de début et de fin de la période d'exécution qui lui est associée. Si le profileur d'ISS est capable d'extraire cette information pour les modules logiciels, alors cet enregistrement contient également le nombre d'instructions, ventilé par type d'instructions, qui ont exécutées pendant cette période d'exécution. Pour un module donné, on additionne le temps d'exécution et le nombre d'instructions de chacune de ces périodes d'exécution pour obtenir son temps total d'exécution et son nombre total d'instructions exécutées. Cela est illustré à la figure 6.6 dans un fichier résumé produit par le profilage au niveau système pour le module Y2R dans l'exemple du décodeur JPEG présenté à la section 9.1.2.

Pour les périodes d'exécution du RTOS, l'enregistrement indique aussi si cette exécution est due à une routine de traitement d'interruption, à une fonction de changement de contexte ou à la tâche inactive. En agrégeant les périodes d'exécution se produisant sur un même processeur, le générateur de métriques obtient la charge totale de chaque processeur ainsi que la contribution à cette charge de chaque module logiciel, des routines de traitement d'interruption, des fonctions de changement de contexte et de la tache inactive. Ces métriques peuvent ensuite être affichées dans l'interface graphique de Space Studio et la figure 6.7 l'illustre pour l'implémentation de l'exemple du décodeur JPEG.

Les enregistrements sur les périodes d'exécution des modules sont particuliers en ce sens qu'il s'agit du seul type d'enregistrement dont le contenu diffère selon qu'un module donné de l'application soit implémenté en logiciel ou en matériel.

```
Data for module Y2R:

0.325518920 seconds of computation (89.27%): 4824450 ALU instructions (43.91%),
1217296 control instructions (11.08%), 4944791 memory instructions (45.01%)

780 bytes read from module IDCT in 0.037098550 seconds
12 bytes written to module IDCT in 0.000007740 seconds
98312 bytes read from device bitmapRAM in 0.089883170 seconds
65536 bytes written to device bitmapRAM in 0.058429600 seconds

Data for communications on bus 1 initiated by module Y2R: 164256 bytes transmitted in
0.001807870 seconds (0.50%)

Memory accesses on bitmapRAM by module mY2R: 163848 bytes from address 0 to 0x18020
```

Figure 6.6 Fichier de métriques de performance produit par le profilage au niveau système pour le module Y2R de l'exemple du JPEG

6.2.2.2 Communication des modules de bout en bout

Ce type d'enregistrement contient de l'information sur une communication de bout en bout entre un module logiciel ou matériel et un autre composant, qui est soit un deuxième module (logiciel ou matériel), soit un périphérique. Cette information inclut l'identificateur des composants source et destination, le temps de début et de fin de l'opération de lecture d'une part et de l'opération correspondante d'écriture d'autre part, ainsi que le nombre d'octets transférés. Cet enregistrement indique également si la lecture et l'écriture sont bloquantes.

Les macros d'instrumentation présentées à la section 6.2.1 et le profileur d'ISS SPACE présenté à la section 6.1.3 permettent de produire un enregistrement pour chaque opération de lecture ou d'écriture effectuée par chacun des modules matériels ou logiciels. Le module central de profilage fusionne un enregistrement de lecture et l'enregistrement d'écriture correspondant pour produire un enregistrement de communication bout en bout. Pour trouver quelle lecture correspond à quelle écriture, le module central de profilage se base sur le modèle de calcul RTPN de SPACE présenté à la section 4.3.5, qui stipule qu'un module peut commencer une nouvelle communication seulement si il a terminé la communication précédente et que les communications entre toute paire de modules sont traitées dans un ordre FIFO. Ce profilage des communications d'une implémentation d'une application SPACE correspond également à un profilage du RTPN équivalent à cette application. Ces enregistrements indiquent donc également les temps auxquels se produisent les évènements du RTPN et cela permet l'extraction des séquences de bits caractérisant l'ordonnancement réalisé pour le RTPN, tel que décrit à la section 4.2.2.

On obtient le temps et le volume total de communications entre deux composants en faisant la somme des volumes et des temps de chacun de leurs enregistrements de communi-

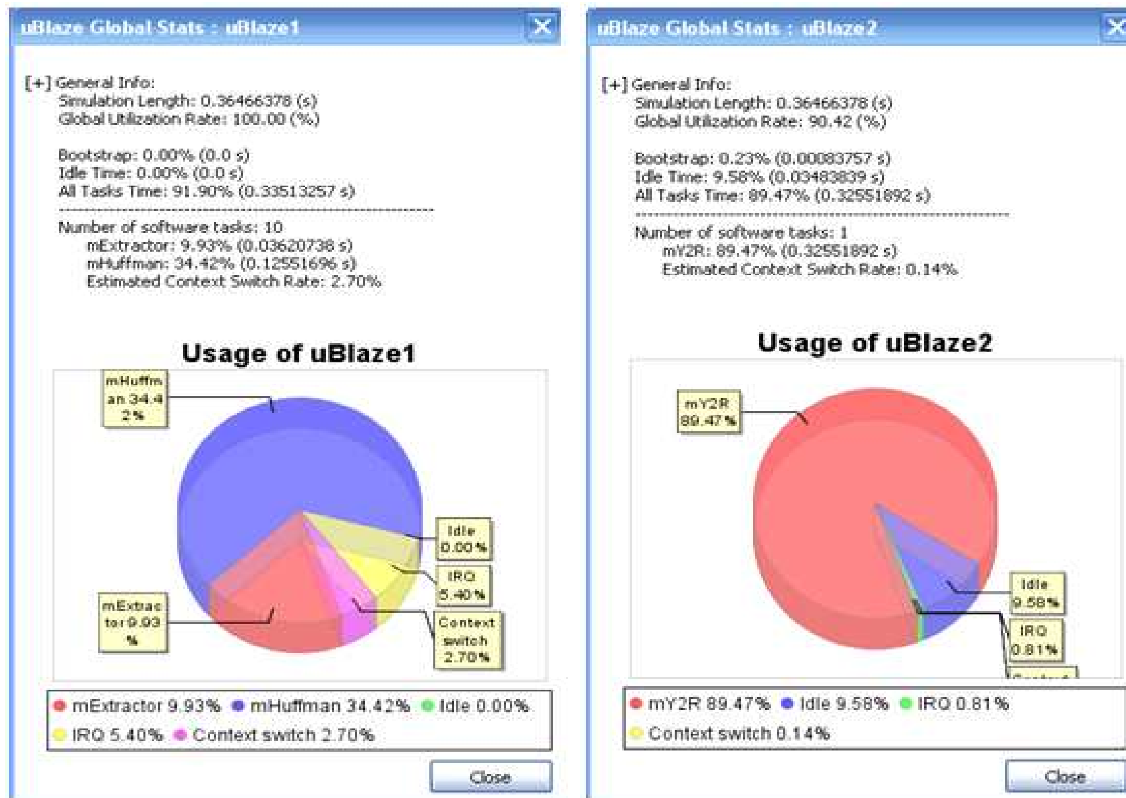

Figure 6.7 Charge des processeurs pour l'exemple du JPEG

cations de bout en bout, tel que présenté à la figure 6.6. Les périodes d'exécution de même que les communications réalisées par un module avec d'autres composants peuvent être affichées dans l'interface graphique de Space Studio sous la forme d'un diagramme de Gantt, tel qu'illustré à la figure 6.8 pour le module de quantification inverse dans l'exemple du JPEG.

6.2.2.3 Transferts sur le bus

L'instrumentation des modèles transactionnels de bus produit une série d'enregistrements portant sur les transferts sur chaque bus. Chaque enregistrement contient de l'information sur le transfert d'un paquet de données sur le bus, notamment l'identificateur du bus, le temps auquel le paquet tente d'accéder au bus, les temps auxquels le transfert du paquet sur le bus débute et se termine, l'identificateur des composants de source et de destination ainsi que le nombre d'octets transférés. Une seule communication de bout en bout peut générer plus d'une communication sur bus, car le paquet peut avoir besoin de traverser plus d'un bus pour atteindre sa destination et parce qu'il peut être nécessaire de renvoyer un acquittement. Ces enregistrements sont agrégés pour déterminer la bande passante utilisée et la charge de chaque bus ainsi que la contribution de chaque module à cette charge, comme le montre la

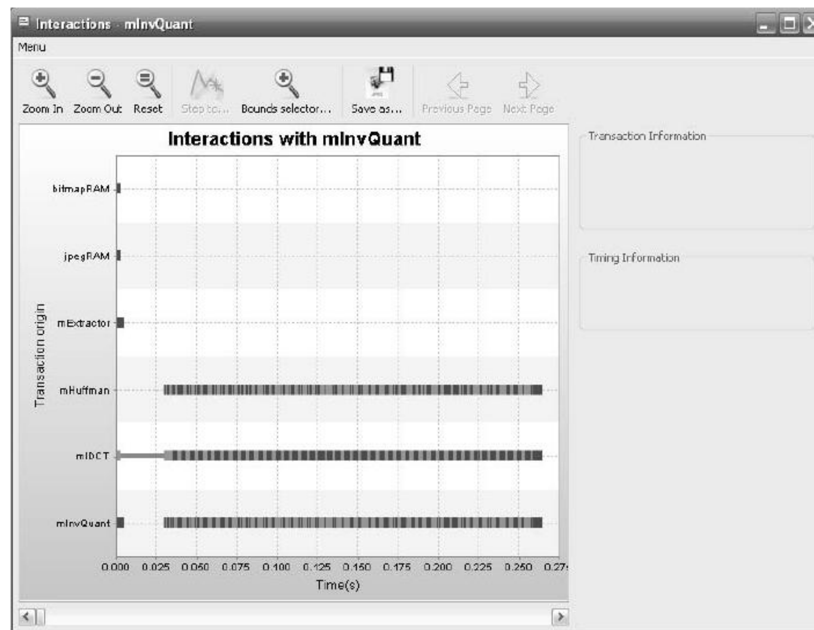

Figure 6.8 Diagramme de Gantt des opérations de calcul et de communication du quantificateur inverse dans l'exemple du JPEG

figure 6.6. Il est également possible d'afficher un diagramme de Gantt des transferts sur le bus, tel qu'illustré à la figure 6.9 pour l'exemple du JPEG.

6.2.2.4 Accès à la mémoire

L'instrumentation des modèles transactionnels de mémoire produit un enregistrement pour chaque accès mémoire en lecture ou en écriture. Chacun de ces enregistrements contient l'identificateur de la mémoire et du module qui y accède, ainsi que le temps, l'adresse et la taille de l'accès mémoire. Ces enregistrements sont agrégés pour obtenir le volume total des accès faits à la mémoire, la contribution de chaque module à ce volume ainsi que la plage d'adresse accédée par chaque module, comme le montre la figure 6.6.

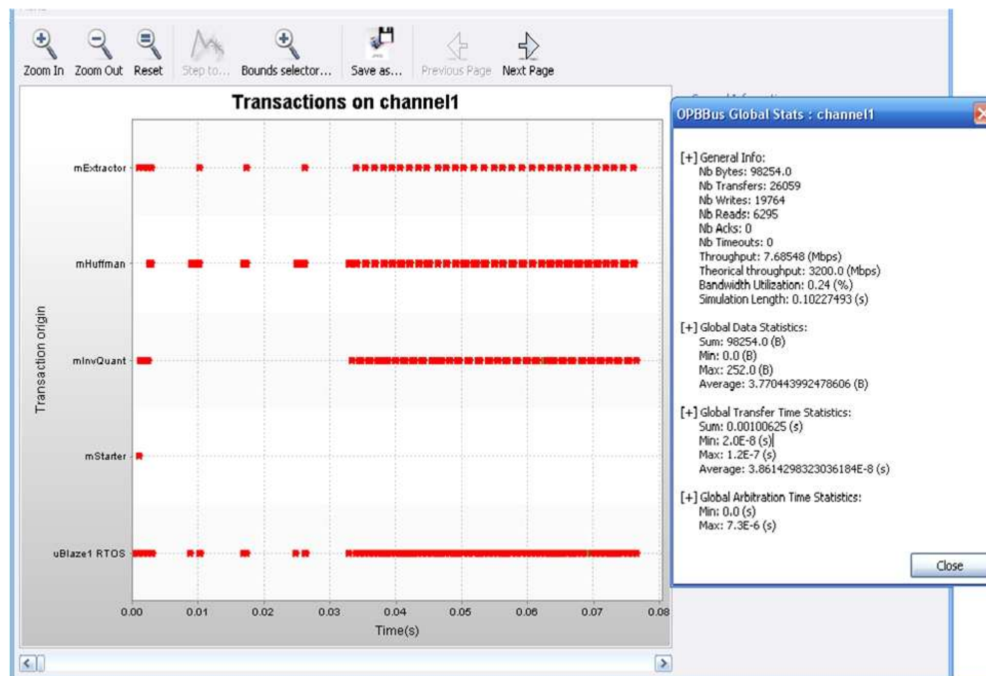

Figure 6.9 Diagramme de Gantt des transferts sur le bus et métriques sur la charge du bus pour l’exemple du JPEG

CHAPITRE 7

CARACTÉRISATION ET ESTIMATION

Le chapitre 6 a présenté comment profiler l'implémentation d'un système embarqué afin d'en extraire des métriques de performance. Cependant, ce profilage nécessite une simulation qui peut demander plusieurs minutes, voire plusieurs heures, et ce même en utilisant des modèles transactionnels des bus, processeurs et périphériques. De la même manière, le chapitre 5 a présenté comment raffiner une architecture donnée jusqu'à une implémentation au niveau RTL. Il est alors possible d'obtenir la quantité de ressources matérielles de cette implémentation en effectuant une synthèse logique sur la plate-forme cible. Cependant, une telle synthèse logique demande encore une fois plusieurs minutes. Or les algorithmes d'exploration architecturale présentés au chapitre 8 auront besoin d'obtenir rapidement les valeurs de ces métriques pour un grand nombre d'architectures possibles, qui se compte en milliers, voire plus. Pour éviter que le temps d'exploration de ces algorithmes ne soit prohibitif, nous présentons dans ce chapitre une méthode rapide et précise d'estimation de performance et de quantité de ressources matérielles qui est basée sur la caractérisation de la plate-forme cible et de l'application.

7.1 Définition des métriques

Une métrique est une propriété quantifiable d'une architecture implémentant une application embarquée. La valeur d'une telle métrique peut être obtenue par différentes méthodes, par exemple la mesure, la simulation ou l'estimation. Ainsi, soit A l'ensemble des architectures pouvant être générées par SPACE, alors une métrique M est une fonction $M : A \to \mathbb{R}$. Les métriques sont évaluées pour les différentes architectures d'une application selon le stimulus fourni à sa spécification exécutable par le banc d'essai qui lui est associé. On suppose que ce stimuli est représentatif des cas auxquels l'application fera face une fois déployée. De plus, toute comparaison de deux architectures d'une même application est effectuée avec le même stimulus, qui doit donc être déterministe ou, à tout le moins, rejouable. L'objectif de ce chapitre est de présenter des méthodes d'estimation qui accélèrent l'évaluation de métriques pertinentes à l'exploration architecturale, soit le nombre de violations fonctionnelles, le temps d'exécution et la quantité de ressources matérielles.

La métrique du nombre de violations fonctionnelles $V : A \to \mathbb{N}$ indique le nombre de fois que le comportement d'une architecture implémentant l'application dévie du comportement

attendu selon la spécification exécutable. Par exemple, il y a déviation si une lecture non-bloquante sur un canal vide est effectuée dans l'architecture mais non dans la spécification exécutable. Cette métrique est égale à 0 si le comportement de l'architecture est tout à fait conforme au comportement attendu de l'application. On considère dans cette thèse que les applications sont modélisées par des réseaux de processus temps-réel (RTPN), qui ont été présentés au chapitre 4. Il a été démontré à la section 4.2.2 que les implémentations (ou ordonnancements) d'un RTPN peuvent être caractérisées par un ensemble de séquences de bits et que deux implémentations sont fonctionnellement équivalentes si et seulement si leurs séquences de bits sont égales. La métrique V est ainsi évaluée comme la somme des distances de Hamming (Hamming, 1950) entre chaque séquence de bits de l'architecture et la séquence de bits correspondante de la spécification exécutable. Cette métrique permet de quantifier à quel point le comportement d'une architecture donnée dévie (ou non) de celui de la spécification exécutable.

La métrique du temps d'exécution total $T : A \to \mathbb{R}$ indique le temps que prend l'architecture pour traiter l'ensemble du stimulus fourni en entrée par le banc d'essai. Comme dans le chapitre 6, il s'agit ici du temps interne à l'architecture, et non du temps pris (WCT) par un simulateur ou un estimateur pour trouver le temps d'exécution de l'architecture. Pour une application donnée, on cherchera généralement à minimiser ce temps d'exécution. Quant aux autres contraintes temporelles spécifiées pour l'application, par exemple une fréquence minimale de production d'une sortie, elles sont modélisées dans le RTPN de l'application et elles sont plutôt prises en compte dans la métrique V présentée ci-haut.

Finalement, pour une plate-forme cible donnée, la métrique de quantité de ressources matérielles $R_j : A \to \mathbb{R}$ indique combien de ressources matérielles de type j sont utilisées par l'architecture. Si la plate-forme cible est un ASIC, il y aura généralement un seul type de ressource matérielle, soit la surface ou les portes logiques, et les architectures seront comparées selon la surface qu'elles occupent ou le nombre de portes logiques qu'elles nécessitent. Si la plate-forme cible est un FPGA, les types de ressources matérielles dépendront alors de la technologie ciblée. Par exemple, pour un FPGA de technologie Virtex (Xilinx Inc., 2002), les types de ressources matérielles sont les bascules (« flip-flops »), les LUTs, les multiplieurs et les mémoires BRAM. Les architectures sont alors comparées selon leur utilisation de chaque type de ressource matérielle.

7.2 Performance et validité fonctionnelle

La figure 7.1 illustre dans son ensemble la méthode de caractérisation et d'estimation du nombre de violations fonctionnelles et du temps d'exécution. La plate-forme subit tout

d'abord une caractérisation initiale. Étant donné que la même plate-forme peut être utilisée pour un grand nombre d'applications, on peut se permettre de caractériser manuellement certains de ses éléments, comme les temps de communications des bus ou les délais des mémoires. D'autres éléments de la plate-forme sont caractérisés automatiquement : soit les temps d'exécution et de communication associés aux opérations du RTOS et de l'API logicielle de SPACE. Différentes applications, spécifiées par l'utilisateur de la méthodologie, peuvent ensuite cibler cette plate-forme et chaque application est alors caractérisée automatiquement. La fonctionnalité de cette application est d'abord caractérisée à l'aide un profilage au niveau système de sa spécification exécutable et par l'extraction d'une trace abstraite de cette application. Puis, l'implémentation logicielle et l'implémentation matérielle de chaque module sont séparément profilées afin de caractériser chacune d'entre elles. Cela indique le temps d'exécution logiciel et matériel de chaque opération effectuée par chaque module dans la trace fonctionnelle. Toutes ces informations de caractérisation sont utilisées pour configurer un simulateur rapide par évènements discrets qui peut alors estimer le nombre de violations fonctionnelles et la performance d'une architecture donnée (qui comprend un partitionnement logiciel/matériel, une allocation des processeurs, une assignations des tâches logicielles aux processeurs et un choix de la topologie de communication).

7.2.1 Caractérisation de la fonctionnalité de l'application

La première étape est la caractérisation de la fonctionnalité de l'application. On commence par définir un type de graphe qui permet de représenter la trace fonctionnelle de la spécification exécutable, soit l'ensemble des opérations qu'elle réalise et la relation de précédence entre celles-ci. On indique ensuite comment un tel graphe peut être extrait en appliquant le profilage au niveau système présenté au chapitre 6 à la spécification exécutable de l'application.

7.2.1.1 Définition de la trace fonctionnelle

Pour représenter la trace fonctionnelle, on définit un nouveau type de graphe, soit un graphe de précédence de communications (CPG : Communication Precedence Graph). Un CPG est un graphe acyclique orienté $G(C, P)$ tel que les noeuds C sont des opérations de communication et les arcs P sont les contraintes de précédence entre ces opérations. Un CPG représente une exécution, qui se déroule dans le temps, d'un RTPN et chaque noeud fait donc partie d'une série d'opérations de communication réalisées par un des processus séquentiels du RTPN. Cela implique que l'ensemble des opérations de communication effectuées par un processus donné forme une chaîne : si les opérations du processus sont les noeuds $c_1, c_2, \ldots, c_n$,

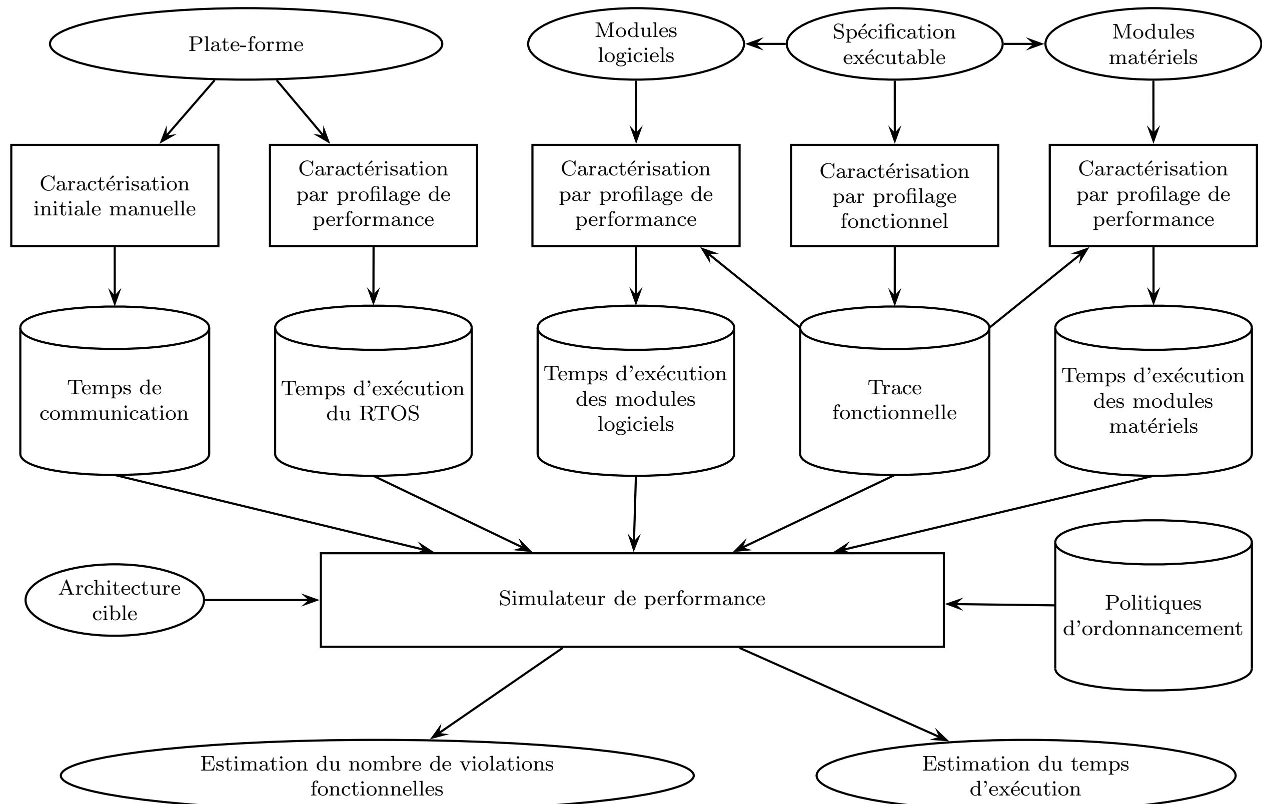

Figure 7.1 Méthode de caractérisation et d'estimation de la performance et du nombre de violations fonctionnelles

alors il y a un ensemble d'arcs, appelés arcs de séquence, qui relient c_1 à c_2, c_2 à c_3, ..., c_{n-1} à c_n. Un arc de séquence représente l'exécution d'un segment de programme, soit un code se trouvant entre deux communications (Wolf et Ernst, 2001; Posadas *et al.*, 2004). L'ensemble des arcs de séquence est noté S.

Les communications peuvent également amener des contraintes de précédence entre les opérations de communication de deux processus distincts, par exemple lors d'une lecture bloquante, lors de l'attente d'un acquittement suite à une écriture bloquante ou lors de l'attente d'une réponse suite à une requête à un périphérique. Une telle contrainte est représentée par un arc, qu'on appelle arc de blocage, et l'ensemble des arcs de blocage est noté B. On a $S \cup B = P$ et $S \cap B = \varnothing$. Il est à noter qu'une lecture non-bloquante n'amène pas une telle contrainte de précédence et qu'on ne lui associe donc pas d'arc de blocage.

On associe à chaque opération de communication $c \in C$ les attributs présentés au tableau 7.1. L'ensemble $C_m \subseteq C$ des opérations appartenant à un processus m, et donc reliées entre elles par des arcs de séquence, est tel que $lp(c) = m$ pour tout $c \in C_m$. Dans le cas où un arc de blocage relie x à y, alors cela signifie que x est une opération d'écriture effectuée par un processus, dont le résultat est lu par l'opération de lecture y d'un autre processus. On a alors $lp(x) = rp(y)$, $lp(y) = rp(x)$, $len(x) = len(y)$, $rnw(x) = 0$, $rnw(y) = 1$ et $step(x) = step(y)$. L'attribut *step* est utilisé pour permettre au CFG de modéliser une communication qui s'effectue en plusieurs étapes. Si une série d'opérations $c_1, c_2, \ldots, c_n$ représentent les différentes étapes d'une telle communication, alors $step(c_i) = i$. Par exemple, une écriture bloquante (avec acquittement) de SPACE est modélisée en deux étapes, tel que présenté à la section 4.3.5 : d'abord une opération d'écriture non-bloquante (w), puis une opération de lecture bloquante d'un acquittement (a). On a alors $step(w) = 1$ et $step(a) = 2$.

Une opération x de lecture non-bloquante lit d'un canal dans lequel écrivent les opérations de communications $y \in C$ tel que $rp(y) = lp(x)$, $lp(y) = rp(x)$, $rnw(y) = 0$ et $step(x) = step(y)$. Toutes les autres opérations de lecture sur le même canal ($c \in C$ tel que $lp(c) = lp(x)$, $rp(c) = rp(x)$, $rnw(c) = 1$ et $step(c) = step(x)$) doivent alors être également non-bloquantes, afin d'éviter qu'un processus alterne les lectures bloquantes et non-bloquantes sur un même canal, tel que décrit à la section 4.3.5.

7.2.1.2 Extraction d'un CPG par profilage

Une procédure automatisée permet d'obtenir le CPG d'une application. Elle commence par une simulation de la spécification exécutable au niveau fonctionnel (Elix) de SPACE tout en appliquant le profilage au niveau système présenté au chapitre 6. Dans une telle simulation fonctionnelle, les calculs et les communications des modules ne sont pas précises au cycle près. Un délai nominal ($\Delta > 0$) est tout de même associé à chaque communication afin d'obtenir

Tableau 7.1 Définition des attributs des opérations de communication

Nom	Description
$lp(c)$	Le processus qui effectue l'opération de communication c.
$rp(c)$	Le processus avec lequel cette opération c communique.
$len(c)$	Nombre d'octets à transmettre ou recevoir lors de la communication c.
$rnw(c)$	Un bit qui indique si c est une opération de lecture (1) ou d'écriture (0)
$step(c)$	Un entier i indiquant que l'opération c est la i^{eme} étape d'une communication en plusieurs étapes. Vaut 1 si la communication n'a qu'une étape.

un ordre chronologique entre les communications d'un même processus (module ou périphérique). Cette delta-causalité des communications fait également en sorte que la simulation fonctionnelle de la spécification exécutable est déterministe, tel que décrit à l'annexe A.4.

Le profilage de la spécification exécutable produit un ensemble d'enregistrements de profilage, tel que décrit à la section 6.2.2. L'extraction d'un CPG se fait en analysant les enregistrements de communication de bout en bout qui sont produits par le profilage et dont les attributs pertinents sont présentés au tableau 7.2. Cette analyse est réalisée en deux étapes. La première étape extrait l'ensemble C des opérations de communication et l'ensemble B des arcs de blocage entre celles-ci. La deuxième étape extrait l'ensemble S des opérations de séquence en triant par ordre chronologique les communications de chaque processus. Le résultat de l'analyse est un CPG $G(C,P)$, avec $P = B \cup S$.

Tableau 7.2 Attributs des enregistrements de communication de bout en bout utilisés pour l'extraction du CPG

Nom	Description
$s(e)$	Le processus source (qui écrit) de la communication représentée par e.
$d(e)$	Le processus destinataire (qui lit) de la communication de e.
$l(e)$	Le nombre d'octets transmis lors de cette communication de e.
$b_w(e)$	Bit indiquant si l'écriture associée à la communication de e est bloquante.
$b_r(e)$	Bit indiquant si la lecture associée à la communication de e est bloquante.
$t_{wb}(e)$	Le temps auquel a débuté l'écriture associée à la communication de e.
$t_{rb}(e)$	Le temps auquel a débuté la lecture associée à la communication de e.
$ok(e)$	Un bit qui indique si la communication de e s'est complétée normalement (vaut 0 dans le cas d'une lecture non-bloquante sur un canal vide).

Les figures 7.2 et 7.3 présentent le code d'un exemple simplifié composé d'un périphérique D_1 et de deux modules M_2 et M_3. La figure 7.4 montre la structure du CPG qui est extrait suite à l'exécution de ce code. On constate que les écritures non-bloquantes de M_2 à M_3

s'effectuent en une seule étape alors que l'écriture au périphérique D_1 et l'écriture bloquante de M_3 à M_2 s'effectuent chacune en deux étapes.

```
device_write(D1_ID, &x);
y = doProcessing1(x);
write(M3_ID, &y, NON_BLOCKING);
write(M3_ID, &z, NON_BLOCKING);
read(M3_ID, &r, BLOCKING);
```

Figure 7.2 Opérations du module M_2

```
read(M2_ID, &a1, BLOCKING);
read(M2_ID, &a2, BLOCKING);
b = doProcessing2(a1,a2);

write(M2_ID, &b, BLOCKING);
```

Figure 7.3 Opérations du module M_3

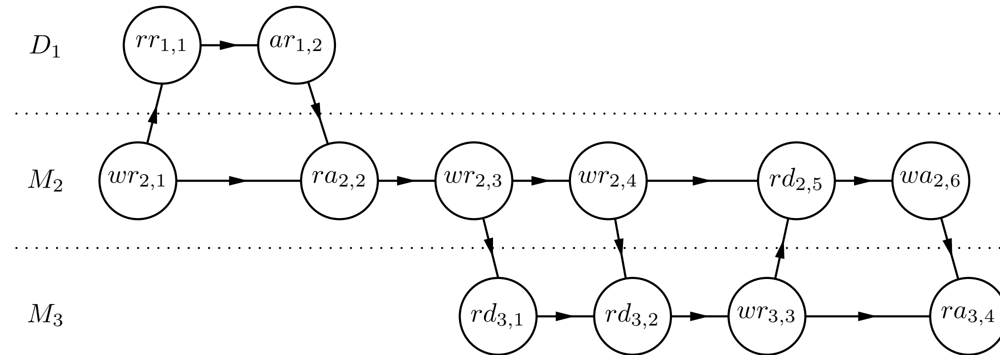

Figure 7.4 CPG des communications du périphérique P_1 et des modules M_2 et M_3

Ajout des noeuds de communication et des arcs de blocage Lors de l'extraction du CPG, un attribut t est ajouté à chaque opération de communication $c \in C$ tel que $t(c)$ désigne le temps auquel cette opération s'est produit. Cet attribut ne sera pas conservé dans le CPG final et sert plutôt à trier les opérations d'un processus pour l'ajout des arcs de séquence. Pour construire le graphe G, on définit une fonction AjoutNoeud$(lp, rp, len, rnw, step, t)$ qui ajoute à G un noeud c dont les attributs prennent les valeurs passées en paramètre. On définit également une fonction AjoutPaire$(p_w, p_r, len, step, t_w, t_r, b_r)$ qui ajoute à G un noeud c_w qui représente une écriture du processus p_w au processus p_r [AjoutNoeud$(p_w, p_r, len, 0, step, t_w)$], un noeud c_r qui représente la lecture associée [AjoutNoeud$(p_r, p_w, len, 1, step, t_r)$], ainsi qu'un arc de blocage b de c_w à c_r si la lecture est bloquante [$b_r == 1$].

Notre procédure construit les noeuds de communication et les arcs de blocage du CPG en traitant chaque enregistrement e de communication de bout en bout :

- Si e est une lecture non-bloquante sur un canal vide [$ok(e) == 0$], alors elle ajoute un seul noeud correspondant à cette lecture [AjoutNoeud($d(e)$,$s(e)$,$l(e)$, 1,1,$t_{rb}(e)$].
- Si e est une communication entre deux modules, alors elle ajoute une première paire de noeuds pour le transfert de données [AjoutPaire($s(e)$,$d(e)$,$l(e)$,1, $t_{wb}(e)$,$t_{rb}(e)$,$b_r(e)$)]

et, si l'écriture est bloquante [$b_w(e) == 1$], une deuxième paire pour l'acquittement [AjoutPaire($d(e), s(e), l_{ack}, 2, t_{rb}(e), t_{wb}(e), 1$)].

Les communications synchrones d'un module avec un périphérique esclave s'effectuent toutes en deux étapes : le module effectue une écriture non-bloquante suivie d'une lecture bloquante alors que le périphérique effectue une lecture bloquante suivie d'une écriture non-bloquante. Ainsi :

- Si e est une écriture sur un périphérique, alors la première étape est l'envoi des données [AjoutPaire($s(e), d(e), l(e), 1, t_{wb}(e), t_{rb}(e), 1$)] et la deuxième étape est la réception de la réponse du périphérique [AjoutPaire($d(e)$,$s(e)$,l_{rep},2,$t_{rb}(e)$, $t_{wb}(e)$,1)].
- Si e est une lecture sur un périphérique, la première étape est l'envoi d'une requête [AjoutPaire($d(e), s(e), l_{req}, 1, t_{rb}(e), t_{wb}(e), 1$)] et la deuxième étape est la réception des données [AjoutPaire($s(e), d(e), l(e), 2, t_{wb}(e), t_{rb}(e), 1$)].

Les paramètres l_{ack}, l_{rep} et l_{req} sont des paramètres fixes de la plate-forme qui indiquent le nombre d'octets respectivement pris par un message d'acquittement, de réponse et de requête. Dans SPACE, on a $l_{ack} = l_{rep} = l_{req} = 1$.

Ajout des arcs de séquence Cette étape ajoute les arcs de séquence entre les noeuds de communication extraits à l'étape précédente. On considère successivement chacun des processus m de la spécification exécutable et on définit l'ensemble $C_m \subseteq C$ tel que $lp(c) = m$ pour tout $c \in C_m$. Les noeuds de l'ensemble C_m sont triés selon un ordre lexicographique défini par leurs attributs t et *step*. En d'autres termes, pour tout $x, y \in C_m$ on a $x < y$ si et seulement si $t(x) < t(y)$ ou ($t(x) = t(y)$ et $step(x) < step(y)$). Étant donné la séquentialité des processus et la delta-causalité des communications, on a $t(x) = t(y)$ et $step(x) = step(y)$ seulement si $x = y$. Cet ordre lexicographique permet donc de trier l'ensemble C_m en une séquence $c_1, c_2, \ldots, c_n$ et on ajoute des arcs de séquence de c_1 à c_2, c_2 à c_3, …, c_{n-1} à c_n. Les attributs t des noeuds sont ensuite retirés et cela termine l'extraction du CPG $G(C, P)$ de l'application.

L'analyse des enregistrements de profilage permet également d'extraire, selon la procédure présentée à la section 4.2.2, les séquences de bits qui caractérisent l'ordonnancement du RTPN réalisé par la spécification exécutable.

7.2.2 Caractérisation du temps d'exécution des modules

Le temps d'exécution, pour une architecture donnée, d'une application dont la fonctionnalité est caractérisée par un CPG dépend en partie des caractéristiques de performance des composants de la plate-forme virtuelle (tel que les bus, les périphériques, les fonctions de

communication et le RTOS), qui ne sont pas spécifiques à l'application. La caractérisation de ces éléments peut être effectuée avant que l'application soit spécifiée et elle est présentée à la section 7.2.3. Par contre, les modules de la spécification exécutable sont spécifiques à l'application et la caractérisation de la performance des implémentations logicielles ou matérielles des modules doit donc être répétée pour chaque application différente. Cette section présente une méthode qui automatise cette caractérisation.

7.2.2.1 Définition des fonctions de caractérisation des modules

Soit un CPG $G(C, P)$, soit l'ensemble $C_m \subseteq C$ des opérations de communication effectuées par le module m et soit $G_m(C_m, S_m)$ le sous-graphe de G obtenu en ne conservant que les noeuds C_m et les arcs de séquence S_m reliant ces noeuds. Tel que décrit à la section 7.2.1.1, on obtient alors la série des communications $c_1, c_2, \ldots, c_n$ effectuées par m et les arcs de séquence de c_i à c_{i+1} représentent chacun l'exécution d'un segment de programme. Une fonction de caractérisation du temps d'exécution des segments de programme du module m est une fonction $f : S_m \to \mathbb{R}$. On peut définir une fonction de caractérisation différente pour chaque implémentation différente d'un même module. Ce concept peut donc être appliqué même si il existe plusieurs implémentations logicielles (ciblant différents processeurs ou avec différentes optimisations de compilateur) et différentes implémentations matérielles (avec différentes caractéristiques de performance et de coût matériel). Pour des raisons de simplification, on considère que chaque module a au plus deux implémentations possibles : une en logiciel et une en matériel et, tel que présenté au tableau 7.3, on définit la fonction de caractérisation sw pour l'implémentation logicielle et hw pour l'implémentation matérielle. La caractérisation automatisée est successivement appliquée à l'implémentation logicielle et à l'implémentation matérielle de chaque module m de la spécification exécutable afin de construire les fonctions hw et sw. Le nombre de fonctions à caractériser croît linéairement ($2n$) avec le nombre n de modules alors que l'espace de recherche croît au moins exponentiellement avec n, tel qu'il sera présenté au chapitre 8.

Tableau 7.3 Fonctions de caractérisation du temps d'exécution des segments de programme d'un module m

Nom	Description
$sw(s)$	Le temps d'exécution du segment s lorsque m est implémenté en logiciel.
$hw(s)$	Le temps d'exécution du segment s lorsque m est implémenté en matériel.

7.2.2.2 Profilage d'une implémentation d'un module

La caractérisation d'une implémentation donnée d'un module m s'effectue à l'aide du profilage d'une simulation SPACE à niveau mixte. Tous les modules de la spécification exécutable demeurent alors à un niveau comportemental, à l'exception du module m qui est remplacé par son implémentation à caractériser (un bloc SystemC au niveau RTL dans le cas d'une implémentation matérielle, un code embarqué exécuté sur un ISS dans le cas d'une implémentation logicielle). Le résultat de cette simulation est un ensemble d'enregistrements de profilage et la procédure décrite à la section 7.2.1.2 est appliquée pour en extraire un CPG, de même que les séquences de bits qui caractérisent l'ordonnancement du RTPN réalisé par la simulation. Ces séquences de bits sont comparées avec celles de la spécification exécutable afin de s'assurer qu'elles sont égales. Si c'est le cas, cela indique que la fonctionnalité simulée lors de la caractérisation de l'implémentation est équivalente à la fonctionnalité simulée lors de la caractérisation de la spécification exécutable, qui a été décrite à la section 7.2.1.2. Cela implique aussi que les CPG des deux simulations sont identiques et que chaque module a exécuté la même série de communications et de segments de programme dans les deux cas. Cela implique également que, pour l'exécution de chaque segment de programme, le chemin parcouru dans le graphe de flot de contrôle (CFG) du code du module est équivalent dans les deux cas (branchements équivalents), et ce même si le CFG subit des transformations (tel qu'un déroulage de boucle) lors du raffinement du niveau comportemental vers son implémentation.

7.2.2.3 Analyse des enregistrements de profilage

À partir du CPG obtenu suite à la simulation mixte, le sous-graphe $G'_m(C'_m, S'_m)$, composé des noeuds C'_m et des arcs de séquence S'_m du module m, est extrait. Si les séquences de bits associées à cette simulation et à celle de la spécification exécutable sont égales, alors ce sous-graphe G'_m est identique au sous-graphe $G_m(C_m, S_m)$ du CPG de la trace fonctionnelle. La caractérisation des segments de programme S_m de G_m consiste donc à trouver le temps d'exécution de chaque segment S'_m dans la simulation mixte. L'extraction du temps d'exécution des segments S'_m se fait par une analyse des enregistrements des périodes d'exécution du module m, qui sont produits par le profilage tel que présenté à la section 6.2.2.1. Les temps d'exécution extraits doivent exclure les temps pris pour l'exécution du RTOS, des interruptions, des changements de contexte ou des fonctions de communication, qui sont plutôt pris en compte dans la caractérisation de la plate-forme présentée à la section 7.2.3 et qui ne doivent donc pas être comptés en double. Ces enregistrements des périodes d'exécution de m excluent tous ces temps sauf le temps d'exécution des fonctions de communication. Une

analyse des enregistrements de communication de bout en bout est donc également nécessaire et elle permet également de délimiter le début et la fin de l'exécution de chaque segment de programme de m.

Pour effectuer cette analyse, on définit un intervalle de nombres réels, noté $i = [a, b]$, comme un ensemble $i = \{x \in \mathbb{R} | a \leq x \leq b\}$. Soit I l'ensemble des intervalles de réels, alors la longueur $\mu : I \to \mathbb{R}$ est une mesure tel que $i = [a, b]$ implique $\mu(i) = b - a$ pour tout $i \in I$. Soit E_x l'ensemble des enregistrements de périodes d'exécution associés à un module m, alors on associe à chaque enregistrement $e \in E_x$ un intervalle $i(e) = [t_b(e), t_e(e)]$ où les attributs t_b et t_e représentent les temps de début et de fin de la période d'exécution de l'enregistrement. De la même manière, soit E_c l'ensemble des enregistrements de communications de bout en bout effectuées par un module m, alors on associe à chaque enregistrement un intervalle $i(e) = [t_{wb}(e), t_{we}(e)]$ si cet enregistrement correspond à une écriture effectuée par m, ou $i(e) = [t_{rb}(e), t_{re}(e)]$ si il s'agit d'une lecture. Pour un module m donné, l'ensemble E_x est tel que tous les enregistrements générés par le profilage ont des intervalles disjoints (pour tout $x, y \in E_x$, on a que $x \neq y$ implique que $\mu(x \cap y) = 0$) et l'ensemble E_c a la même propriété.

Chaque enregistrement $e \in E_c$ représente une série d'opérations de communication dans C'_m qui sont les différentes étapes d'une même communication. Pour chaque communication qui est en plusieurs étapes, on fusionne en un seul noeud dans C'_m les noeuds correspondant à ces plusieurs étapes, ce qui permet d'établir, après avoir trié E_c en ordre chronologique, une bijection (correspondance un-à-un) entre les enregistrements de E_c et les noeuds restants dans C'_m. Ces fusions éliminent également les arcs de séquence qui se trouvaient entre les différentes étapes d'une communication. Ceux-ci représentent des segments de programme qui sont exécutés à l'intérieur des fonctions de communication et qui sont donc plutôt caractérisés avec la plate-forme à la section 7.2.3. Les segments de programme restants dans S'_m correspondent à la fonctionnalité propre au module, dont la performance doit être caractérisée. Sans perte de généralité, supposons qu'un segment $s \in S'_m$ relie (après fusion) le noeud de communication c_1 au noeud c_2 tel que l'intervalle de l'enregistrement associé au noeud c_1 est $[a_1, b_1]$ et que celui associé au noeud c_2 est $[a_2, b_2]$, alors le temps d'exécution associé au segment s est égal à la somme des intervalles des enregistrements d'exécution se trouvant entre ces deux communications. Plus précisément, si f est la fonction de caractérisation à construire, alors on a :

$$f(s) = \sum_{e \in E_x} \mu(e \cap [b_1, a_2]) \tag{7.1}$$

7.2.3 Caractérisation de la plate-forme

La caractérisation de la plate-forme consiste en la caractérisation des éléments de la plate-forme qui sont génériques et ne sont pas spécifiques à une application particulière. Ces éléments comprennent notamment des composants matériels tels que des périphériques, des adaptateurs de bus, des bus, des processeurs ainsi que des composants logiciels tels qu'un RTOS et une API logicielle. Comme les composants matériels d'une plate-forme sont généralement stables et que leurs caractéristiques de performance sont souvent standardisées, on réalise une caractérisation manuelle de la performance de ces composants matériels. Étant donné que les composants logiciels sont plus malléables et que leurs caractéristiques de performance peuvent facilement changer d'une version à l'autre, on présente une méthode automatisée de caractérisation de la performance des composants logiciels.

La caractérisation de la plate-forme permet notamment d'estimer le temps pris par les communications, les changements de contexte et le traitement des interruptions pour une architecture donnée ciblant cette plate-forme. Un paramètre important pour la caractérisation des composants matériels est leur largeur en octets : ce paramètre désigne le nombre maximal d'octets que le composant matériel est capable de recevoir ou envoyer en un seul transfert sur le bus. Si un module effectue une communication de n octets avec un composant matériel de largeur l tel que $n > l$, alors cette communication doit être scindée par l'adaptateur de bus en plusieurs transferts de taille inférieure ou égale à l, soit en $\lceil n/l \rceil$ transferts.

Les méthodes présentées dans cette section pourraient s'appliquer à différentes plates-formes utilisant différentes types de bus, de processeurs et de RTOS. Ces méthodes sont présentées ici en utilisant la plate-forme virtuelle SPACE, qui cible un FPGA Xilinx muni de bus OPB CoreConnect IBM Corp. (1999), de processeurs MicroBlaze (Xilinx Inc., 2005) et d'un RTOS μC/OS II (Labrosse, 2002).

7.2.3.1 Caractérisation des périphériques

Les principaux paramètres de performance d'un périphérique sont sa largeur l en octets et le temps t qu'il prend pour traiter chaque requête. On suppose que le périphérique prend un temps t identique pour traiter deux requêtes différentes qui sont chacune de taille inférieure ou égale à l. Le temps que le périphérique prend pour traiter une requête de n octets est donné par $\lceil n/l \rceil t$. Ce temps caractérise l'arc de séquence correspondant au traitement d'une requête par le périphérique dans le CPG de l'application. Il n'inclut pas les délais causés par le transfert de la requête ou de la réponse sur le bus : ceux-ci sont plutôt pris en compte dans la caractérisation du bus. Le tableau 7.4 présente les valeurs de ces paramètres de performance pour différents périphériques qui peuvent se brancher sur un bus OPB dans un

FPGA Xilinx. Les valeurs de t sont données en cycles d'horloge pour que cette caractérisation puisse s'appliquer à différentes fréquences d'horloge.

Tableau 7.4 Paramètres de performance pour différents périphériques avec interface OPB

Nom	l	t
BRAM (sur puce)	4 octets	1 cycle
SDRAM (externe)	4 octets	14 cycles
UART	1 octets	1 cycle

7.2.3.2 Caractérisation des adaptateurs de bus

Les adaptateurs de bus fournis par SPACE connectent chaque module implémenté en matériel à un bus et leur permettent de communiquer via leur bus respectif. Il est à noter que les autres composants matériels (tels les périphériques et les processeurs) n'ont pas besoin de ces adaptateurs de bus, car leur protocole de communication est déjà adapté au bus. Dans la plate-forme virtuelle SPACE, la largeur des adaptateurs de bus est présentement fixée à 4 octets. Ainsi, toute requête ou réponse envoyée ou reçue du bus par un module est scindée en transferts de 4 octets lors de la communication avec l'adaptateur de bus. Pour tous ces transferts à l'exception des acquittements, il y a un délai d'un cycle pour initier la série de transferts et l'adaptateur prend ensuite exactement un cycle pour recevoir 4 octets du module ou pour lui transmettre 4 octets. Il y a donc un délai de $1 + \lceil n/4 \rceil$ pour une communication de n octets.

Cette valeur ne tient pas compte des délais de communications entre l'adaptateur et le bus : ceux-ci sont plutôt pris en compte lors de la caractérisation du bus. Dans le cas d'une écriture bloquante, le module transmetteur doit également attendre que le module récepteur ait retourné son acquittement. L'adaptateur de bus du module récepteur se charge de répondre avec un acquittement dès que le module récepteur a lu cette communication. Ce délai de réception de l'acquittement est indéterminé tant que les opérations de ces modules n'ont pas été ordonnancées : c'est donc l'estimateur d'ordonnancement qui permet d'obtenir ce délai.

7.2.3.3 Caractérisation des bus et des ponts

Un bus est un canal de communication qui peut être partagé par plusieurs maîtres pour communiquer avec un ou plusieurs esclaves. Si plusieurs maîtres attendent d'obtenir le bus et que celui-ci se libère, un arbitre choisit quel maître obtient le bus. Une caractéristique d'un

bus est donc sa politique d'arbitrage. Un autre paramètre de performance est le délai t_a que prend le bus pour effectuer chaque arbitrage. Comme pour les autres composants matériels, la largeur l_b en octets d'un bus constitue un paramètre de performance important. Finalement, le dernier paramètre de performance est le délai t_c pour un transfert de l_b octets ou moins sur le bus lorsque le maître détient le bus. Si on suppose que le maître ne libère pas le bus tant qu'il n'a pas fini de transférer le paquet au complet, alors le temps nécessaire au transfert d'un paquet de n octets vers un destinataire de largeur l_d se trouvant sur le même bus est donné par :

$$t_a + \lceil n/min(l_b, l_d) \rceil t_c \tag{7.2}$$

Ce temps donne le délai à partir du moment où s'effectue l'arbitrage qui donne le contrôle du bus à ce maître pour ce transfert. Le délai entre le moment où le maître demande l'accès au bus et le moment où cet arbitrage s'effectue ne sera connu que lorsque les opérations du système seront ordonnancées, étant donné qu'il dépend du moment où les autres maîtres demandent l'accès au bus.

Dans le cas où le destinataire se trouve sur un bus différent, il y a un délai d'arbitrage et de transfert sur chacun des deux bus en plus d'un délai de transfert sur le pont qui relie les deux bus. Le délai sur chaque bus est donné par l'équation 7.2. Le délai associé à chaque transfert sur le pont (de taille inférieure ou égale à $min(l_b, l_d)$) est égal à t_p. On trouve donc que le temps pour transférer un paquet de n octets d'un transmetteur sur un bus vers un récepteur sur un autre bus est donné par :

$$2t_a + \lceil n/min(l_b, l_d) \rceil (2t_c + t_p) \tag{7.3}$$

Ce temps exclut, encore une fois, le temps pendant lequel un des deux bus est occupé par une autre communication.

Le bus OPB supporte deux politiques d'arbitrage : la première utilise des priorités statiques assignées à chaque maître et la seconde est un algorithme LRU qui donne le bus au maître qui a le moins récemment obtenu le bus. Ces deux politiques sont modélisées dans la plate-forme SPACE et dans la caractérisation du bus OPB. Pour les paramètres de performance du bus OPB et du pont OPB-OPB tels que modélisés dans SPACE, on obtient les valeurs présentées au tableau 7.5.

7.2.3.4 Caractérisation du RTOS et de l'API logicielle

Le dernier élément de la plate-forme à caractériser est le temps requis pour les opérations du RTOS et de l'API logicielle. Cela inclut la gestion des communications logicielles, que

Tableau 7.5 Paramètres de performance du bus OPB et du pont OPB-OPB

Paramètre	Description	Valeur
l_b	Largeur du bus	4 octets
t_a	Délai d'arbitrage	3 cycles
t_c	Délai de transfert	5 cycles
t_p	Délai du pont	4 cycles

celles-ci se fassent à l'intérieur d'un même processeur, avec d'autres processeurs, avec des modules matériels ou avec des périphériques. Cela comprend également les changements de contexte et les traitements des interruptions du processeur. Les paramètres de performance du RTOS et de l'API logicielle sont présentés au tableau 7.6 et sont décrits plus en détails à l'annexe B.

Étant donné que le code logiciel associé au RTOS ou à l'API logicielle peut subir des mises à jour et que leur performance peut en être modifiée, une méthode automatisée est présentée pour caractériser automatiquement leurs paramètres de performance. Cette méthode utilise une application synthétique, dont la spécification exécutable est définie avec SPACE, pour exercer les différents cas d'utilisation des fonctions de communications de SPACE : lectures et écritures bloquantes ou non-bloquantes vers des modules ou des périphériques avec différentes tailles de paquet. Selon l'architecture qui l'implémente, l'application synthétique exerce aussi (indirectement) les fonctions de changement de contexte et les ISR du RTOS. Cette application synthétique est simulée et profilée avec différentes architectures, qui représentent différents partitionnements logiciel/matériel sur un ou deux processeurs. Cette méthode examine ensuite l'ensemble des enregistrements générées par le profilage de ces simulations et en extrait les paramètres de performance présentés au tableau 7.6. Les détails de cette méthode sont présentés à l'annexe B.

On constate que les délais associés aux communications effectuées par les modules logiciels sont élevés. On propose à la section 10.2.5 des pistes de solution pour diminuer ces délais, mais l'optimisation du RTOS ou de l'API logicielle de SPACE déborde du cadre de cette thèse. En l'absence de telles optimisations et pour éviter que les temps d'exécution du RTOS et de l'API logicielle deviennent le facteur dominant de l'exploration architecturale réalisée au chapitre 8, on définit un facteur d'accélération a du RTOS et de l'API logicielle. Ainsi, si on effectue une estimation de performance avec un facteur d'accélération de a, alors les valeurs des paramètres de performance du RTOS et de l'API logicielle seront divisées par a par rapport aux valeurs présentées dans le tableau 7.6. L'ajout d'un tel facteur d'accélération se fait sans perte de généralité puisqu'il demeure possible d'utiliser directement les valeurs

Tableau 7.6 Paramètres de performance du RTOS μCOS II et de l'API logicielle

Nom	Description	Valeur
t_{ctx}	Délai de changement de contexte	489 cycles
t_{isr0}	Délai de base de l'ISR de réception d'un message	2600 cycles
t_{isr+}	Délai additionnel de cette ISR pour chaque 4 octets	798 cycles
$t_{isrunblk}$	Délai additionnel si l'ISR débloque la tâche destinataire	1023 cycles
t_{israck}	Délai de l'ISR de réception d'un acquittement	1885 cycles
t_{per0}	Délai de base d'une communication avec un périphérique	272 cycles
t_{per+}	Délai additionnel pour chaque 4 octets	83 cycles
t_{hw0}	Délai de base d'une écriture à un module externe	618 cycles
t_{hw+}	Délai additionnel de l'écriture pour chaque 4 octets	83 cycles
t_{hwblk}	Délai additionnel si cette écriture est bloquante	494 cycles
$t_{hwackblk}$	Délai additionnel si cette écriture bloque	930 cycles
t_{sw0}	Délai de base d'écriture à un module du même processeur	2035 cycles
t_{sw+}	Délai additionnel de cette écriture pour chaque 4 octets	679 cycles
$t_{swunblk}$	Délai additionnel si elle débloque le module destinataire	938 cycles
t_{swblk}	Délai additionnel si l'écriture est bloquante	342 cycles
$t_{swackblk}$	Délai additionnel si l'écriture bloque	466 cycles
t_{rd0}	Délai de base d'une lecture d'un message	1487 cycles
t_{rd+}	Délai additionnel pour chaque 4 octets	684 cycles
t_{rdblk}	Délai additionnel d'une lecture qui bloque	699 cycles
$t_{rdempty}$	Délai d'une lecture non-bloquante sur un canal vide	473 cycles
t_{hwack}	Délai d'un acquittement à un module externe	276 cycles
t_{swack}	Délai d'un acquittement à un module du même processeur	677 cycles
$t_{swackunblk}$	Délai additionnel si il débloque le module destinataire	321 cycles

du tableau 7.6 en spécifiant $a = 1$.

7.2.4 Estimation par une simulation de performance

Le temps d'exécution total d'une architecture implémentant une application embarquée avec un ordonnancement dynamique dépend de facteurs complexes à modéliser tels que la contention sur les bus, le nombre et les moments des changements de contexte sur les processeurs ainsi que le nombre de fois et les moments où les tâches bloquent et sont débloquées. Il semble difficile de déduire le temps d'exécution total d'une architecture de manière analytique à partir des paramètres de performance des différents composants de cette architecture. Il faut donc avoir recours à la simulation pour modéliser ces aspects et tenir compte de l'arbitrage des communications sur les bus et de l'ordonnancement des tâches sur les processeurs.

La simulation complète d'une architecture avec SystemC peut prendre plusieurs minutes, voire plusieurs heures. C'est trop long si on désire évaluer un grand nombre d'architectures, comme dans les algorithmes d'exploration architecturale présentés au chapitre 8. Cette section présente donc une méthode d'estimation basée sur l'utilisation d'un simulateur de performance configuré selon la caractérisation présentée aux sections 7.2.1 à 7.2.3. Trois facteurs permettent d'accélérer grandement la vitesse du simulateur de performance par rapport à une simulation au niveau Simtek de SPACE (ou par rapport à un simulateur précis au cycle près) :

1. Le simulateur de performance ne modélise pas les opérations du système qui ont lieu à chacun des cycles d'exécution, mais temporise plutôt la simulation à un niveau plus élevé. Par exemple, un délai est associé à l'ensemble d'un segment de programme plutôt que de simuler cycle par cycle chacune de ses instructions.
2. Le simulateur de performance n'exécute pas les calculs internes aux modules. Par exemple, pour un module DCT, seul un délai est associé au calcul de la DCT d'un ensemble de points : le calcul lui-même n'est pas effectué.
3. Le simulateur de performance s'exécute en un seul fil d'exécution, contrairement à une simulation SystemC ou un fil d'exécution est associé à chaque module. Ces changements de contexte du simulateur (à ne pas confondre avec les changements de contexte sur les processeurs de l'architecture simulée) seraient le facteur limitant la vitesse de simulation suite aux seules étapes 1 et 2 et c'est pourquoi nous avons conçu le simulateur de performance de manière à les éliminer.

7.2.4.1 Fonctionnement général du simulateur de performance

La méthode utilisée pour estimer la performance d'une architecture est une simulation par évènements discrets de la trace fonctionnelle (le graphe CPG) selon les paramètres de performance extraits pour les modules de l'application et la plate-forme. Ainsi, on parcourt la liste des opérations de communication de chaque module en suivant les arcs de séquence qui les relient. Une opération o peut être exécutée seulement après que toutes les opérations qui la précèdent, soit toutes les opérations x tel qu'il existe un arc de x vers o, aient déjà été exécutées et après avoir tenu compte des délais sur ces arcs. Le temps d'exécution d'un arc de séquence s, qui représente un segment de programme, est donné par $hw(s)$ ou $sw(s)$ selon que le module auquel appartient cet arc est implémenté en logiciel ou en matériel. Si le module est implémenté est matériel, chaque opération ou arc de séquence peut s'exécuter dès que ses précédences sont satisfaites. Si le module est plutôt implémenté en logiciel, il faut de plus que le module détienne le contrôle du processeur. L'ordonnancement des tâches sur le processeur est modélisé selon la politique d'ordonnancement du RTOS et un délai de changement de contexte t_{ctx}, obtenu lors de la caractérisation de la plate-forme, se produit lorsqu'un module obtient ou cède le contrôle du processeur. Si la politique d'ordonnancement du RTOS est préemptive (comme dans le cas de μCOS/II), alors il est possible qu'une tâche soit interrompue pour céder sa place à une tâche plus prioritaire. Le simulateur détermine alors quelle proportion de l'opération de communication ou de l'arc de séquence a eu le temps de s'exécuter avant cette préemption, puis un délai associé au temps d'exécution restant est ajouté une fois que la tâche interrompue reprend le contrôle du processeur.

Les opérations et les arcs de communication sont modélisés selon le CPG, l'architecture de l'implémentation et la caractérisation des bus, des adaptateurs de bus, des périphériques, du RTOS et de l'API logicielle. Ainsi, pour chaque opération de communication faite par un module implémenté en matériel, le simulateur modélise les délais associés aux adaptateurs de bus, aux bus et, si il y a lieu, aux périphériques. Pour chaque opération de communication faite par un module logiciel, on modélise les délais associés à l'API logicielle de même que, si nécessaire, aux bus, aux interruptions et aux changements de contexte. Le simulateur modélise les délais nécessaires à l'obtention des bus, les délais de transfert et, dans le cas d'une communication bloquante, les délais d'attente selon les paramètres de performance extraits lors de la caractérisation. L'accès au bus est aussi modélisé selon la politique d'arbitrage du bus. La simulation de performance se termine lorsque toutes les opérations du CPG ont été ordonnancées : le temps de fin de l'opération qui se termine le plus tardivement constitue le temps d'exécution estimé pour l'architecture a simulée. Cela correspond donc à la valeur de la métrique $T(a)$.

L'exécution du simulateur de performance assigne un temps de début et de fin à chacune

des opérations de communication du CPG et, selon ces temps et selon la spécification du RTPN, le simulateur extrait les séquences de bits associées au RTPN selon la procédure présentées à la section 4.2.2. Cela permet de calculer la distance de Hamming entre chaque séquence de bits simulée et la séquence correspondante de la spécification exécutable et donc d'évaluer la métrique $V(a)$ de l'architecture a simulée.

Dans certains cas, il est possible qu'on veuille seulement considérer l'impact de certains modules sur le temps d'exécution, par exemple si on évalue une solution partielle dans laquelle seulement certains modules ont été assignés en matériel ou en logiciel et où les autres modules demeurent à un niveau comportemental. Dans ce cas, le délai associé aux opérations de communication et aux arcs de séquence de ces modules comportementaux est considéré comme nul et on considère donc que leurs délais d'accès ou de transfert avec les adaptateurs, les bus, les ponts et les périphériques sont nuls.

7.2.4.2 Implémentation du simulateur de performance

Il existe différentes manières d'implémenter une simulation par évènements discrets, parmi lesquelles on compte l'approche par processus et l'approche par évènements (Derrick *et al.*, 1989). Dans l'approche par processus, chaque objet significatif du système est modélisé par un processus qui suit un fil d'exécution dans lequel il génère, attend et traite des évènements et où il réalise des opérations. Une simulation SystemC utilisant des `SC_THREAD` est un exemple d'une approche par processus. Dans l'approche par évènements, il n'y a pas de processus, mais plutôt un ensemble d'évènements qui sont ordonnancés par ordre chronologique. Une routine d'évènement est associée à chaque évènement et est exécutée lorsque la simulation est rendue à cet évènement. Cette routine peut alors ajouter à l'ordonnanceur d'autres évènements à des temps futurs et la simulation prend fin lorsqu'il n'y a plus d'évènements qui sont ordonnancés.

Un premier simulateur par évènements discrets utilisant une approche par processus a été bâti avec la librairie Java DESMO-J (Page et Kreutzer, 2005). Ce simulateur fait correspondre un fil d'exécution à chaque module, bus et processeur de l'architecture tel que les modules exécutent séquentiellement leurs opérations de communication et arcs de séquence, que les bus arbitrent et traitent les requêtes de communication et que les processeurs ordonnancent les modules logiciels selon la politique du RTOS. Pour accélérer la simulation, les adaptateurs de bus, les périphériques et les ponts sont modélisés comme de simples délais à l'intérieur des modules et des bus. Bien que ce simulateur de performance soit bien plus rapide que la simulation complète avec Simtek, il demeure relativement lent et peut prendre une minute pour évaluer certaines solutions. Le profilage Java du simulateur a démontré que celui-ci passait environ 80% de son temps d'exécution à effectuer des changements de contexte entre ses différents fils d'exécution.

Pour remédier à ce problème, un deuxième simulateur par évènements discrets a été implémenté en utilisant une approche par évènements. C'est ce simulateur qui a été retenu pour l'exploration architecturale réalisée au chapitre 8. La librairie DESMO-J n'a pas pu être utilisée pour ce simulateur, car elle ne supporte pas la modélisation de la préemption dans une approche par évènements. Nous avons donc implémenté en Java notre propre ordonnanceur d'évènements qui supporte la préemption. L'ordonnanceur possède une liste d'évènements E tel que sont associés, pour tout évènement $e \in E$, le temps $t(e)$ auquel l'évènement doit être exécuté et la routine d'évènement $f(e)$ qui doit alors être exécutée. Cette routine d'évènement peut alors ajouter d'autres évènements à l'ordonnanceur et, afin de modéliser une préemption, elle peut retirer de E un évènement qui n'a pas encore été exécuté par l'ordonnanceur (par exemple l'évènement de fin d'exécution de la tâche qui a été préemptée). L'ordonnanceur maintient un temps t interne et exécute par ordre chronologique, tout en incrémentant son temps t, les routines d'évènements jusqu'à ce qu'il ne reste plus d'évènements à exécuter.

Des exemples d'évènements dans notre simulateur de performance sont le début et la fin d'un arc de séquence, le début et la fin d'une opération de communication, la réception d'un acquittement ou d'une interruption, ou l'arbitrage d'une communication sur le bus. Une routine d'évènement est associée à chacun de ces évènements. Par exemple, l'exécution de la routine d'évènement pour l'arbitrage du bus choisit la prochaine communication à arbitrer, puis ajoute à l'ordonnanceur un évènement correspondant à la réception du paquet par l'adaptateur de bus ou le processeur du destinataire après le délai du transfert sur le bus.

Il a été constaté que la simulation de performance peut rencontrer une situation d'interblocage dans l'accès aux bus pour deux communications qui passent chacune sur deux bus : une première communication peut obtenir le bus #1 et chercher à obtenir le bus #2 alors que celui-ci est détenu par une deuxième communication qui a besoin d'obtenir le bus #1 pour se compléter. Cette situation d'interblocage avait déjà été constatée au niveau Simtek de SPACE (Migliorini, 2008) et elle est reflétée par le simulateur de performance. Celui-ci détecte l'interblocage juste avant qu'il ne se produise et incrémente un compteur du nombre d'interblocages. Exceptionnellement et pour éviter l'interblocage, la première communication est simulée sans la forcer à obtenir le bus #2, tout en simulant néanmoins le délai de transfert sur le pont et le bus #2. Cette capacité d'éviter l'interblocage et de compter le nombre d'interblocages évités permet de quantifier la gravité de ce problème pour l'architecture simulée. Ce nombre d'interblocages est ajouté à la métrique $V(a)$ de l'architecture simulée a.

7.2.4.3 Validité de l'estimation

Le résultat obtenu par le simulateur de performance est un ordonnancement du CPG extrait pour l'application et donc un ordonnancement t de son RTPN, tel que défini à la

section 4.2. Cet ordonnancement t est le résultat des caractéristiques de l'architecture et est une estimation de ce qui se produirait lors d'une simulation complète de cette architecture. Le temps d'exécution estimé suppose que l'architecture exécute le même ensemble d'opérations de communication et de segments de programme que ceux du CPG. En d'autres termes, l'estimation du temps d'exécution suppose que l'ordonnancement t réalisé par l'architecture est fonctionnellement équivalent à l'ordonnancement du RTPN réalisé par la spécification exécutable.

La méthode présentée dans cette section vérifie cette hypothèse en comparant entre elles les séquences de bits caractérisant leurs ordonnancements respectifs et en évaluant une métrique V qui quantifie à quel point ces ordonnancements sont fonctionnellement différents. Si la métrique V est égale à 0, alors cela signifie que leurs ordonnancements sont fonctionnellement équivalents et que l'hypothèse est vérifiée : on peut donc avoir confiance en la valeur obtenue pour la métrique T du temps d'exécution. Sinon, on obtient une contradiction car l'ordonnancement réalisé par l'architecture diffère de celui de la spécification exécutable et le CPG extrait pour une simulation complète de l'architecture ne serait donc pas nécessairement le même que celui (de la spécification exécutable) utilisé pour estimer son temps d'exécution. Une métrique $V > 0$ vient donc lever un doute sur la valeur T estimée. C'est une raison supplémentaire qui explique pourquoi les architectures a telles que $V(a) > 0$ sont pénalisées dans l'exploration architecturale présentée au chapitre 8.

La méthode de caractérisation et d'estimation présentée dans cette section peut être appliquée pour des ordonnancements arbitraires réalisés par la spécification exécutable. Cependant, notre implémentation actuelle de cette méthode exige, pour des raisons de simplification, que les séquences de bits associées à la spécification exécutable soient exclusivement composées de 1, c'est-à-dire que l'ordonnancement réalisé par la spécification exécutable appartienne à la classe T_{KPN} des ordonnancements fonctionnellement équivalents à un KPN classique.

7.3 Quantité de ressources matérielles

La figure 7.5 illustre dans son ensemble la méthode de caractérisation et d'estimation de la quantité de ressources matérielles. Ainsi, la plate-forme subit d'abord une caractérisation initiale de ses différents composants matériels. En faisant varier les paramètres de chaque composant matériel (par exemple le nombre de maîtres et d'esclaves pour un bus) et en faisant une synthèse logique pour chacun de ces cas, on obtient une base de données de coût matériel qui indique combien de ressources matérielles sont consommées par chaque composant en fonction de ses paramètres. Lorsqu'une application cible cette plate-forme, on caractérise

alors automatiquement l'implémentation matérielle de chaque module de l'application, qui a été générée selon la méthode présentée au chapitre 5. Cette caractérisation via la synthèse logique produit une banque de données de coût matériel pour les implémentations matérielles des modules de l'application. Finalement, on caractérise les implémentations logicielles des modules pour déterminer leurs besoins en mémoire RAM. Un estimateur utilise ensuite ces banques de données pour estimer les métriques R_j de quantité de ressources matérielles pour une architecture donnée.

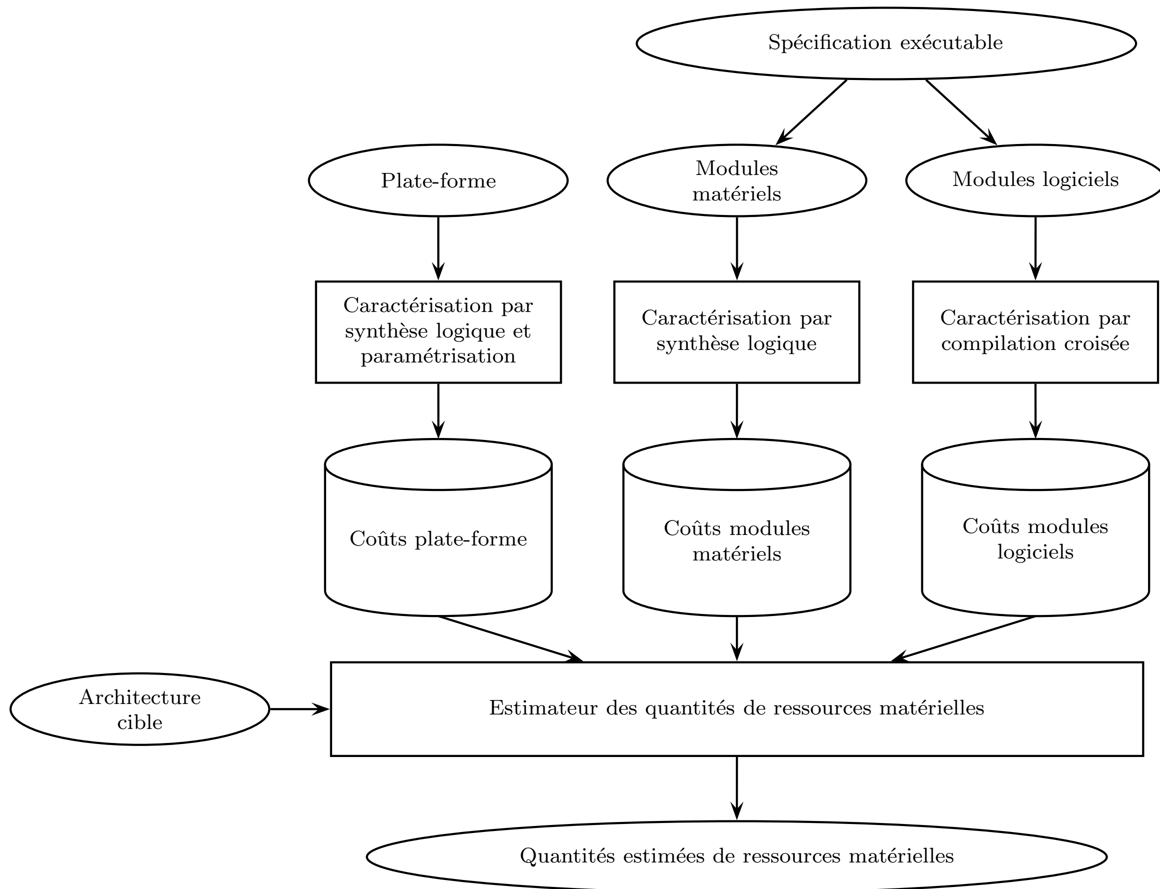

Figure 7.5 Méthode de caractérisation et d'estimation de la quantité de ressources matérielles

7.3.1 Caractérisation des modules matériels

L'implémentation matérielle d'un module, telle que générée avec la méthode de synthèse matérielle présentée au chapitre 5, est une implémentation du module au niveau RTL. Cette méthode permet de générer un code dans un langage HDL, tel que VHDL ou Verilog, pour cette implémentation RTL. Pour une cible ASIC ou FPGA donnée, la caractérisation de la quantité de ressources matérielles de l'implémentation matérielle d'un module consiste donc à

réaliser une synthèse logique de son code VHDL ou Verilog à l'aide d'un synthétiseur logique, puis à extraire du résultat de la synthèse logique les quantités de ressources matérielles nécessaires au module.

L'outil de synthèse comportementale Cynthesizer (Meredith, 2008), qui est utilisé dans notre méthode de synthèse matérielle, génère un code Verilog pour l'implémentation matérielle d'un module. La caractérisation de celle-ci pour une cible FPGA Virtex de Xilinx s'effectue à l'aide d'une synthèse logique de son code Verilog par le synthétiseur XST (Xilinx Inc., 2006). Les quantités de ressources matérielles, soit les quantités de bascules (« flip-flops »), de LUTs, de mémoires BRAM et de multiplieurs, sont ensuite extraites en analysant le rapport de synthèse produit par XST. Ces données sont ajoutées dans une banque de données de coût matériel des modules. Autant le lancement des synthèses logiques que la récupération des résultats sont automatisés.

7.3.2 Caractérisation des modules logiciels

Les ressources matérielles utilisées par l'implémentation logicielle d'un module, excluant les ressources consommées par le processeur lui-même, sont la mémoire RAM nécessaire pour stocker les instructions et les données de ce module logiciel. Une contrainte de SPACE que les modules de l'application doivent respecter est que ces modules ne font pas d'allocation dynamique de mémoire. De plus, on suppose que, à l'instar de μC/OS-II, le RTOS est tel que la pile (*stack*) de chaque tâche logicielle est un tableau alloué statiquement. Dans ce cas, il devient possible de déterminer statiquement la mémoire requise par les modules logiciels. Ainsi, pour chaque processeur cible d'une architecture donnée, il suffit de compiler le code logiciel (incluant le RTOS, l'API logicielle et les modules assignés au processeur) du processeur suite à la synthèse logicielle présentée à la section 3.2.5.1, puis d'analyser cet exécutable à l'aide de l'utilitaire `size` de la suite GNU (Barr et Massa, 2006). Cela indique la mémoire RAM nécessaire au stockage et au fonctionnement du logiciel de ce processeur. Il est à noter que cette quantité de mémoire n'inclut pas une éventuelle mémoire cache, qui serait plutôt caractérisée avec le processeur et le reste de la plate-forme.

Il est relativement long de synthétiser et compiler le code logiciel pour un processeur cible : ce temps est de l'ordre de plusieurs secondes, voire d'une minute. Ce temps peut devenir prohibitif si, comme dans les algorithmes d'exploration architecturale du chapitre 8, il est nécessaire d'évaluer des milliers d'implémentations. Il est possible de conserver dans une base de données la quantité de mémoire requise par un ensemble donné de modules sur un processeur, afin de ne pas avoir à compiler deux fois le même exécutable avec le même ensemble de modules. Cependant, si il y a n modules qui peuvent être implémentés en logiciel, alors il y a $2^n - 1$ ensembles de modules qui peuvent être assignés à un processeur

donné (excluant l'ensemble vide), ce qui représente une croissance exponentielle. Dans le cadre d'une exploration architecturale, un tel nombre d'exécutables à compiler peut être prohibitif.

Pour limiter le nombre d'exécutables à compiler, on définit un modèle linéaire où la mémoire $M(p)$ requise pour le logiciel sur un processeur p est évaluée, pour une application donnée avec n modules pouvant être implémentés en logiciel, par l'équation suivante :

$$M(p) = \beta_0 + \sum_{i=1}^{n} \beta_i s_i(p) + e(p) \tag{7.4}$$

Dans cette équation :

- Les variables $s_i(p)$ sont des variables booléennes qui indiquent si le module i est implémenté en logiciel sur le processeur p ou non ;
- Le paramètre β_0 représente la mémoire requise par le RTOS et l'API logicielle ;
- Le paramètre β_i représente la mémoire requise par le module i ;
- Le terme $e(p)$ est l'erreur entre le modèle linéaire et la véritable quantité de mémoire nécessaire au code du processeur p.

Les valeurs des paramètres β_0 et β_i sont estimées par une régression linéaire multiple. On obtient alors l'estimé suivant pour la mémoire nécessaire au code du processeur.

$$\hat{M}(p) = \hat{\beta}_0 + \sum_{i=1}^{n} \hat{\beta}_i s_i(p) \tag{7.5}$$

Notre méthode de caractérisation des modules logiciels automatise l'extraction d'un nombre suffisant d'échantillons initiaux et la régression linéaire sur ceux-ci. Pour chaque module, un échantillon est généré tel que le module est assigné seul sur un processeur. Le code logiciel associé à ce processeur est synthétisé, compilé et analysé avec `size` pour trouver sa taille M. Dans ce cas, les variables s_i valent toutes 0 sauf celle qui correspond au module assigné. De même, pour chaque paire de modules, un échantillon est généré tel que ces deux seuls modules sont assignés à un processeur et la taille M du code logiciel correspondant est mesuré avec `size`. Les variables s_i sont alors toute à 0 sauf les deux qui correspondent aux modules assignés. La méthode des moindres carrés ordinaires (Weisberg, 2005) est ensuite appliquée à cet ensemble d'échantillons pour trouver les valeurs estimées de $\hat{\beta}_0$ et $\hat{\beta}_i$. Le nombre d'exécutables à compiler est alors égal à $n(n+1)/2$, ce qui est bien moins que les $2^n - 1$ exécutables qu'il serait autrement nécessaire de compiler.

7.3.3 Caractérisation de la plate-forme

Lors de la synthèse de la plate-forme, qui a été décrite à la section 5.3.2, les modèles transactionnels de bus, de processeurs et de périphériques de la plate-forme virtuelle sont

remplacés par les composants RTL équivalents de la plate-forme cible. Une caractérisation initiale de la synthèse de plate-forme permet d'établir, pour chaque composant de l'architecture à haut niveau, quel est l'ensemble de composants RTL qui lui est associé. Il est ensuite nécessaire de caractériser les quantités de ressources matérielles de ces composants RTL afin de pouvoir estimer les besoins en ressources matérielles d'une architecture qui cible cette plate-forme. Ces quantités de ressources matérielles sont automatiquement caractérisées par une synthèse logique de chaque composant et par une analyse du rapport de synthèse selon une méthode similaire à celle présentée à la section 7.3.1.

Il est possible que les paramètres de certains composants RTL de la plate-forme, et donc leurs quantités de ressources matérielles, dépendent de l'architecture choisie. Ces composants doivent donc être caractérisés pour différentes valeurs de leurs paramètres. Le tableau 7.7 présente les paramètres variables des composants RTL d'un FPGA Virtex de Xilinx ciblé par SPACE.

Tableau 7.7 Paramètres des composants RTL d'un FPGA Virtex ciblé par SPACE

Composant	Paramètre	Valeurs possibles
Adaptateur de bus	Nombre de FIFO de réception	$1, 2, \ldots, n$
Bus	Nombre de maîtres	$1, 2, \ldots, n$
Bus	Nombre d'esclaves	$1, 2, \ldots, n$
Bus	Arbitrage avec priorités dynamiques	$0, 1$
Pont	Nombre d'esclaves	$1, 2, 3, 4$

On observe que les valeurs de certains paramètres ne sont pas bornées et pourraient être arbitrairement élevées. Cela signifie que la caractérisation de ces composants pour l'ensemble des valeurs possibles de leurs paramètres demanderait un nombre arbitrairement élevé de synthèses logiques, ce qui est bien sûr impossible. La solution retenue est donc de faire une caractérisation initiale de ces composants seulement pour des valeurs de paramètres qui sont inférieures à une borne maximale N donnée, par exemple $N = 16$. Les résultats extraits suite à ces synthèses logiques sont placés dans une base de données qui indique combien de ressources matérielles sont consommées par chaque composant en fonction des valeurs de ses paramètres. Si il s'avère qu'on a besoin, lors de l'évaluation d'une architecture, de connaître la quantité de ressources matérielles d'un composant pour des valeurs de paramètres qui n'ont pas été caractérisées, alors une synthèse logique sera automatiquement faite avec ces nouvelles valeurs et le résultat sera ajouté à la base de données.

Il est à noter que, dans le cadre de l'exploration architecturale avec SPACE, certains composants de la plate-forme cible, tels les processeurs, sont utilisés toujours avec exactement les mêmes valeurs pour leurs paramètres. Ces composants sont donc caractérisés seulement

pour ces valeurs fixes. Si ces paramètres cessaient d'être fixes et devenaient variables, par exemple si l'exploration architecturale permettait d'ajouter une cache au processeur et de faire varier sa taille, alors il serait possible de caractériser ces composants pour les différentes valeurs de ces paramètres.

7.3.4 Estimation des quantités des ressources matérielles

L'estimation des quantités de ressources matérielles nécessaires à une architecture donnée consiste simplement à faire la somme des quantités de ressources matérielles consommées par chaque composant RTL alloué dans l'implémentation RTL de l'architecture et de la mémoire requise par chaque processeur de l'architecture.

Ainsi, selon la caractérisation de la synthèse de plate-forme présentée à la section 7.3.3, l'estimateur obtient la liste des composants RTL qui seraient alloués dans une implémentation RTL de l'architecture. Les quantités de ressources matérielles de chacun de ces composants RTL sont évaluées selon les banques de données de coût matériel construites à la section 7.3.1 (implémentations matérielles des modules) et 7.3.3 (composants de la plate-forme). De manière analogue, la quantité de mémoire RAM utilisée par le logiciel embarqué de chaque processeur de l'architecture est évaluée selon la méthode présentée à la section 7.3.2. L'estimateur obtient ensuite par sommation, pour chaque type j de ressource matérielle de la technologie cible, une estimation pour l'architecture de la métrique R_j qui représente la quantité de ressources j utilisées par l'architecture.

Une telle estimation ne tient pas compte des ressources matérielles qui pourraient être nécessaires au routage des composants du système, mais elle devrait être assez précise pour guider l'exploration architecturale.

CHAPITRE 8

EXPLORATION ARCHITECTURALE

Les métriques et les méthodes d'estimation présentées au chapitre 7 permettent d'évaluer rapidement une architecture donnée d'une application SPACE selon des critères de validité fonctionnelle, de quantité de ressources matérielles et de performance. Il devient ainsi possible d'évaluer plusieurs architectures d'une même application afin de les comparer et de choisir celle qui répond le mieux à ces critères de qualité : il s'agit de l'exploration architecturale.

Ce chapitre commence par préciser l'espace de recherche des problèmes d'exploration architecturale comme étant le choix du partitionnement logiciel/matériel, de l'allocation des processeurs, de l'assignation des tâches logicielles aux processeurs et de la topologie de communication de l'architecture. On poursuit par la présentation des différents problèmes d'exploration architecturale qu'on vise à résoudre, soit l'étude de faisabilité, la maximisation de la performance et la minimisation de la quantité des ressources matérielles. On présente ensuite les fonctions objectives permettant d'évaluer et de comparer les architectures pour chacun de ces problèmes. On discute ensuite de la complexité combinatoire et algorithmique de ces problèmes. Puis, on présente différents algorithmes pour résoudre ces problèmes, soit un algorithme glouton, le parcours en profondeur, la marche aléatoire, la descente, le recuit simulé adaptatif et la recherche tabou réactive.

8.1 Définition de l'espace de recherche

Soit S l'ensemble des spécifications exécutables d'applications embarquées pouvant être créées avec une méthodologie (par exemple un système de guidage d'un robot marcheur ou un décodeur JPEG) et soit P l'ensemble des plates-formes pouvant être ciblées par cette méthodologie (par exemple un FPGA Xilinx, un FPGA Altera ou un ASIC) et soit A l'ensemble des architectures pouvant être générées par cette méthodologie. Une fonction $f : S \times P \to \wp(A)$ est définie comme une fonction de définition de l'espace de recherche. Concrètement, cela signifie que pour une spécification $s \in S$ donnée et une plate-forme $p \in P$ donnée, alors $f(s,p)$ correspond à un sous-ensemble de A qui est l'ensemble des architectures qui constituent l'espace de recherche pour cette spécification à implémenter sur cette plate-forme. De la même manière, une fonction de définition de l'espace de recherche peut s'appliquer sur un sous-ensemble $P' \subseteq P$ de plates-formes et avoir la forme $f : S \times P' \to \wp(A)$

L'exploration architecturale présentée dans ce chapitre cible l'ensemble $P_{1,1} \subseteq P$ des plates-

formes ayant un seul type de processeur et un seul type de bus. On a choisi deux fonctions de définition de l'espace de recherche soit $f_1 : S \times P_{1,1} \to \wp(A)$ et $f_2 : S \times P_{1,1} \to \wp(A)$. Supposons que pour une plate-forme $p \in P_{1,1}$ donnée et pour une spécification $s \in S$ donnée on ait n_s tâches logicielles, n_h composants matériels et n_m modules qui peuvent être implémentés soit en matériel, soit en logiciel sur la plate-forme p. (Dans le contexte d'une spécification exécutable SPACE, les n_s tâches logicielles sont des modules qui doivent nécessairement être implémentés en logiciel alors que les n_h composants matériels sont soit des périphériques, soit des modules qui doivent nécessairement être implémentés en matériel.) Alors l'espace de recherche défini par $f_1(s, p)$ est constitué de l'ensemble des architectures qu'on peut obtenir en faisant un choix à chacune des étapes suivantes :

(a) Le partitionnement logiciel/matériel : choisir, parmi les n_m modules de l'application, m_s modules qui seront implémentés en tant que tâches logicielles et donc les $m_h = n_m - m_s$ autres modules qui seront implémentés en tant que composants matériels ;

(b) L'allocation des processeurs et l'assignation des tâches aux processeurs : choisir le nombre k de processeurs à allouer, puis assigner chacune des $n_s + m_s$ tâches logicielles du système à un des k processeurs alloués ;

(c) Le choix d'une topologie de communication : choisir le nombre l de bus à allouer, puis assigner chacun des $n_h + m_h$ composants matériels et des k processeurs du système à ces l bus.

De plus, les choix faits lors de ces étapes doivent répondre aux contraintes suivantes :

1. Chaque module a une et une seule implémentation logicielle ainsi qu'une et une seule implémentation matérielle ;
2. On doit implémenter un module soit en tant qu'une et une seule tâche logicielle, soit en tant qu'un et un seul composant matériel, mais pas les deux à la fois ;
3. Une tâche logicielle doit être assignée à un et un seul processeur ;
4. Un composant matériel doit être assigné à un et un seul bus.
5. Si un processeur est alloué, alors au moins une tâche logicielle doit lui être assignée ;
6. Si un bus est alloué, alors au moins un processeur ou deux composants matériels doivent lui être assignés ;

La contrainte 1 signifie que l'optimisation de l'implémentation logicielle ou de l'implémentation matérielle d'un module donné ne fait pas partie de l'espace de recherche. On suppose que les implémentations logicielle et matérielle fournies pour chaque module ont déjà été optimisées et l'exploration architecturale proposée se situe donc à un niveau plus élevé. Les contraintes 2 et 3 signifient que l'espace de recherche défini par f_1 ne fait pas varier le nombre d'instances d'un même module. La contrainte 4 signifie que cet espace de recherche contient seulement des architectures où les composants matériels sont connectés à un seul bus. La seule

exception est les ponts qui relient les bus entre eux : on suppose que chaque paire de bus dans une architecture donnée est connectée par un pont bidirectionnel. Cependant, comme le même ensemble de ponts sera toujours alloué et assigné de la même manière pour un même ensemble de bus, leur allocation et assignation ne font pas partie de l'espace de recherche, bien que les valeurs des métriques tiennent compte de la présence des ponts. La contrainte 5 assure qu'aucun processeur superflu n'est alloué. Finalement, la contrainte 6 fait en sorte qu'au moins deux composants matériels (excluant les ponts) soient connectés à chaque bus qui est alloué. Lorsqu'un processeur est alloué, d'autres composants matériels qui lui sont dédiés (par exemple un contrôleur d'interruption ou une minuterie) sont également alloués et on suppose que, lorsqu'un processeur est assigné à un bus, ces composants matériels dédiés sont également assignés au même bus. Dans ce travail, l'exploration de l'allocation et de l'assignation de ces composants dédiés ne fait donc pas partie de l'espace de recherche, mais les ressources matérielles utilisées par ceux-ci sont tout de même prises en compte.

Il est possible que les bus, les adaptateurs de bus, les ponts et les processeurs d'une technologie cible puissent être configurées. On considère que, pour une plate-forme p donnée, la configuration de chaque bus, adaptateur, pont ou processeur est fixe et identique, sauf dans les cas où elle doit varier strictement pour prendre en considération les choix faits dans les étapes (a), (b) ou (c). Par exemple, l'arbitre d'un bus sur lequel on a assigné dix composants matériels diffèrera de l'arbitre d'un bus sur lequel seulement deux composants matériels ont été assignés, mais la configuration de cet arbitre découle directement de l'assignation des composants aux bus. Bien qu'on tienne compte de l'impact de ces configurations sur le nombre de ressources matérielles utilisées, l'exploration de ces configurations ne fait donc pas partie de l'espace de recherche.

La fonction $f_2 : S \times P_{1,1} \to \wp(A)$ est semblable à la fonction f_1, avec la différence que, parmi les n_h composants matériels de la spécification, il y a n_d composants matériels double-port qui ont deux interfaces de communication. Cela entraine un changement à la contrainte 4 pour f_2 : un composant matériel qui a une seule interface de communication doit être assigné à un et un seul bus ; un composant matériel qui a deux interfaces de communication doit être assigné soit à un et un seul bus, soit à exactement deux bus distincts.

8.2 Problèmes d'exploration architecturale

Les problèmes d'exploration architecturale étudiés dans ce chapitre ont tous la même forme générale. On a un ensemble de métriques qui permettent d'évaluer et de comparer entre elles les architectures possibles d'une spécification donnée sur une plate-forme donnée et on cherche l'architecture qui minimise une métrique donnée tout en respectant les contraintes

sur chacune des autres métriques. Formellement, soit une spécification $s \in S$, un ensemble de plates-formes $P' \subseteq P$, une plate-forme $p \in P'$, une fonction de définition d'espace de recherche $f : S \times P' \to \wp(A)$, un ensemble de métriques $M_1, M_2, \ldots, M_n : A \to \mathbb{R}$ et un ensemble de contraintes $k_1, k_2, \ldots k_{n-1}$, alors l'exploration architecturale consiste à trouver l'architecture $a \in f(s, p)$ telle que :

(i) $M_j(a) \leq k_j, \forall 1 \leq j \leq n-1$ (respect des contraintes) ;
(ii) $M_n(a) \leq M_n(b), \forall b \in f(s, p)$ (solution optimale).

Cette formulation suppose que d'optimiser une métrique implique de la minimiser. Cela se fait sans perte de généralité, car si une métrique M doit être maximisée, il suffit alors de minimiser la métrique $-M$ telle que $(-M)(a) = -(M(a))$ pour tout a. De la même manière, si on ne cherche pas une solution optimale, mais seulement à vérifier l'existence d'une solution qui répond à un ensemble de contraintes, il suffit de définir la métrique 0 telle que $0(a) = 0$ pour tout a et de la définir comme la métrique à optimiser.

À partir de cette définition générale, on définit les problèmes d'exploration architecturale qui sont étudiés dans ce chapitre et qui utilisent les métriques présentées au chapitre 7.

Problème 8.1 (Étude de faisabilité)**.** Soit une spécification $s \in S$, un ensemble de plates-formes $P' \subseteq P$, une plate-forme $p \in P'$, une fonction de définition d'espace de recherche $f : S \times P' \to \wp(A)$, la métrique du nombre de violations V, un nombre maximal de violations V_{max}, la métrique du temps d'exécution T, un temps d'exécution maximal T_{MAX} et, pour chaque ressource matérielle j, une métrique de quantité R_j et une quantité maximale $R_{j,MAX}$. Déterminer si il existe une architecture $a \in f(s, p)$ telle que :

(i) $V(a) \leq V_{MAX}$ (contrainte sur le nombre de violations) ;
(ii) $T(a) \leq T_{MAX}$ (contrainte sur le temps d'exécution) ;
(iii) $R_j(a) \leq R_{j,MAX}, \forall j$ (contrainte sur la quantité de ressources).

Comme son nom l'indique, le problème 8.1 est une étude de faisabilité qui consiste à déterminer s'il est possible d'implémenter une spécification donnée sur une plate-forme donnée selon des contraintes de validité fonctionnelle, de quantité de ressources matérielles et de performance. Tel que décrit ci-haut, il est trivial d'exprimer cette étude de faisabilité comme un problème d'optimisation qui minimise la métrique 0.

Problème 8.2 (Maximisation de la performance)**.** Soit une spécification $s \in S$, un ensemble de plates-formes $P' \subseteq P$, une plate-forme $p \in P'$, une fonction de définition d'espace de recherche $f : S \times P' \to \wp(A)$, la métrique du nombre de violations V, un nombre maximal de violations V_{max}, la métrique du temps d'exécution T et, pour chaque ressource matérielle j, une métrique de quantité R_j et une quantité maximale $R_{j,MAX}$. Trouver une architecture $a \in f(s, p)$ telle que :

(i) $V(a) \leq V_{MAX}$ (contrainte sur le nombre de violations) ;
(ii) $R_j(a) \leq R_{j,MAX}, \forall j$ (contrainte sur la quantité de ressources) ;
(iii) $T(a) \leq T(b), \forall b \in f(s,p)$ (minimisation du temps d'exécution).

Le problème 8.2 ressemble au problème 8.1, avec la différence qu'on cherche maintenant à maximiser la performance du système (donc à minimiser son temps d'exécution) tout en respectant des contraintes de validité fonctionnelle et de quantité de ressources matérielles.

Un autre problème est la minimisation de la quantité des ressources matérielles. Ce problème peut être formulé directement selon la définition générale s'il y a un seul type de ressource matérielle à minimiser, tel que le nombre de portes logiques dans le cas d'un ASIC. Cependant, la situation se complique dans le cas d'un FPGA, où il y a plusieurs types de ressources matérielles (des LUTs, des bascules, des mémoires, etc.) dont on pourrait vouloir minimiser la quantité. On pourrait définir, pour chaque ressource matérielle, un problème différent visant à minimiser la quantité de celle-ci. En pratique, il est souvent possible de substituer un type de ressources matérielles par un autre type (par exemple un ensemble de bascules et une mémoire RAM, ou un ensemble de LUTs et un bloc DSP dédié) et il serait donc préférable de minimiser la quantité de toutes les ressources matérielles plutôt que seulement celle d'un seul type.

L'approche retenue consiste donc à combiner linéairement les métriques associées à chaque ressource matérielle en une seule métrique de quantité de ressources matérielles, qui sera minimisée. Ainsi, pour chaque ressource matérielle j, soit un coefficient α_j et une métrique de quantité de ressources matérielles $R_j : A \to \mathbb{R}$, alors la métrique combinée de quantité de ressources de matérielle $R : A \to \mathbb{R}$ est définie telle que $R(a) = \sum \alpha_j R_j(a)$ pour tout a. Il est alors possible de définir le problème de minimisation de la quantité des ressources matérielles selon la définition générale du problème d'exploration architecturale.

Problème 8.3 (Minimisation de la quantité des ressources matérielles)**.** Soit une spécification $s \in S$, un ensemble de plates-formes $P' \subseteq P$, une plate-forme $p \in P'$, une fonction de définition d'espace de recherche $f : S \times P' \to \wp(A)$, la métrique du nombre de violations V, un nombre maximal de violations V_{max}, la métrique du temps d'exécution T, un temps d'exécution maximal T_{MAX} et une métrique combinée de quantité de ressources matérielles R. Trouver une architecture $a \in f(s,p)$ telle que :
(i) $V(a) \leq V_{MAX}$ (contrainte sur le nombre de violations) ;
(ii) $T(a) \leq T_{MAX}$ (contrainte sur le temps d'exécution) ;
(iii) $R(a) \leq R(b), \forall b \in f(s,p)$ (minimisation de la quantité de ressources matérielles).

On peut se demander comment choisir les coefficients α_j pour obtenir la métrique combinée de quantité de ressources matérielles. Intuitivement, pour un type de ressource matérielle,

plus il y a une grande quantité disponible, moins le coefficient associé à cette ressource devrait être grand et vice-versa. Si la plate-forme ciblée est un modèle de FPGA en particulier, alors la quantité disponible pour chaque type de ressource matérielle est connue et, pour une ressource matérielle j, on peut définir cette quantité disponible comme étant $R_{j,MAX}$. Dans ce chapitre, nous utilisons des coefficients $\alpha_j = 1/R_{j,MAX}, \forall j$.

8.3 Fonctions objectives

Pour guider les algorithmes d'exploration architecturale visant à résoudre les problèmes présentés à la section précédente, on définit pour chaque problème une fonction objective $o : A \to \mathbb{R}$ qui évalue les architectures. Plus la valeur de la fonction objective pour une architecture donnée est petite, plus cette architecture répond aux contraintes et aux critères d'optimisation du problème.

La forme générale des fonctions objectives utilisées dans ce chapitre est une fonction objective avec pénalités (Lopez-Vallejo *et al.*, 2000). Soit un ensemble $M_1, M_2, \ldots, M_n$ de métriques à optimiser, soit un ensemble de contraintes $k_1, k_2, \ldots, k_n$ pour chaque métrique, soit un ensemble de coefficient de poids $\omega_1, \omega_2, \ldots, \omega_n$ pour chacune des métriques et soit un ensemble de coefficients de pénalité $\phi_1, \phi_2, \ldots, \phi_n$, alors la fonction d'évaluation $o : A \to \mathbb{R}$ est donnée par :

$$o(a) = \sum_{i=1}^{n} \omega_i \frac{M_i(a)}{k_i} + \sum_{i=1}^{n} \phi_i \left[\frac{\max(0, M_i(a) - k_i)}{k_i} \right]^2 \tag{8.1}$$

La première sommation cherche à optimiser l'ensemble des métriques. On normalise chacune des métriques en la divisant par la contrainte qui lui est associée, puis on effectue une combinaison linéaire des métriques normalisées selon leurs coefficients de poids. La deuxième sommation représente les pénalités associées aux violations de contraintes. Si une contrainte est respectée, alors le terme $M_i(a) - k_i$ qui lui est associé sera inférieur à zéro et la pénalité pour cette contrainte sera nulle. Par contre, si la contrainte n'est pas respectée, alors la pénalité sera proportionnelle au carré de la valeur normalisée du dépassement de la contrainte. On effectue ensuite une combinaison linéaire des pénalités selon les coefficients de pénalité associés à chaque contrainte. La valeur des coefficients de poids ω_i est typiquement inférieure ou égale à 1 alors que celle des coefficients de pénalité ϕ_i est supérieure à 100. Cela assure que cette fonction objective privilégiera les solutions qui respectent toutes les contraintes tout en permettant l'exploration de solutions qui violent légèrement certaines contraintes (Lopez-Vallejo *et al.*, 2000).

Avec les métriques définies à la section 7.1, soit V, T et, pour chaque ressource matérielle, R_j, on obtient donc la forme générale suivante pour les fonctions objectives se rapportant à

l'exploration architecturale :

$$o(a) = \omega_{V_+}\frac{V_+(a)}{k_{V_+}} + \omega_T\frac{T(a)}{k_T} + \sum_{j=1}^{n}\omega_{R_j}\frac{R_j(a)}{k_{R_j}} + \phi_{V_+}\left[\frac{\max(0, V_+(a) - k_{V_+})}{k_{V_+}}\right]^2 + \phi_T\left[\frac{\max(0, T(a) - k_T)}{k_T}\right]^2 + \sum_{j=1}^{n}\phi_{R_j}\left[\frac{\max(0, R_j(a) - k_{R_j})}{k_{R_j}}\right]^2 \quad (8.2)$$

Dans cette équation, $V_+ : A \to \mathbb{R}$ est une métrique dérivée de V telle que $V_+(a) = V(a) + 1$ pour tout a. La raison pour laquelle on utilise V_+ plutôt que V est qu'on voudra typiquement n'avoir aucune violation fonctionnelle, soit une contrainte k_V égale à 0. Comme ces contraintes se trouvent au dénominateur, cela impliquerait une division par zéro. En utilisant V_+, la contrainte k_{V_+} associée à un nombre maximal de 0 violations (soit aucune violation) est égale à 1, ce qui évite ce problème.

Pour le problème 8.1 de l'étude de faisabilité, la fonction objective doit seulement chercher à assurer le respect des contraintes et non l'optimisation des métriques. Les coefficients de poids ω_{V_+}, ω_T et ω_{R_j} ont donc tous la valeur zéro : aucune métrique n'est optimisée. Ensuite, les contraintes k_{V_+}, k_T et k_{R_j} de la fonction objective sont respectivement égales aux contraintes $V_{+,MAX} = V_{MAX} + 1$, T_{MAX} et $R_{j,MAX}$ du problème 8.1. On obtient donc la fonction objective suivante pour ce problème :

$$o(a) = \phi_{V_+}\left[\frac{\max(0, V_+(a) - V_{+,MAX})}{V_{+,MAX}}\right]^2 + \phi_T\left[\frac{\max(0, T(a) - T_{MAX})}{T_{MAX}}\right]^2 + \sum_{j=1}^{n}\phi_{R_j}\left[\frac{\max(0, R_j(a) - R_{j,MAX})}{R_{j,MAX}}\right]^2 \quad (8.3)$$

Pour le problème 8.2 de la maximisation de la performance, la seule métrique à optimiser est le temps d'exécution. Cela implique que ω_{V_+} et ω_{R_j} ont tous la valeur zéro et on fixe $\omega_T = 1$. On a également $k_{V_+} = V_{+,MAX}$ et $k_{R_j} = R_{j,MAX}$. Quant à la contrainte k_T de la fonction objective, la contrainte T_{MAX} équivalente n'est pas nécessairement présente dans le problème 8.2 étant donné que le temps d'exécution est le critère à optimiser. Cependant, supposer qu'une telle valeur soit également fournie n'implique aucune perte de généralité, car il suffit de donner à T_{MAX} une valeur arbitrairement grande (mais non infinie) pour que cette contrainte soit toujours respectée. On obtient alors la fonction objective suivante pour le problème 8.2 :

$$o(a) = \frac{T(a)}{T_{MAX}} + \phi_{V_+}\left[\frac{\max(0, V_+(a) - V_{+,MAX})}{V_{+,MAX}}\right]^2 + \phi_T\left[\frac{\max(0, T(a) - T_{MAX})}{T_{MAX}}\right]^2 + \sum_{j=1}^{n}\phi_{R_j}\left[\frac{\max(0, R_j(a) - R_{j,MAX})}{R_{j,MAX}}\right]^2 \quad (8.4)$$

Pour le problème 8.3 de la minimisation de la quantité de ressources matérielles, on cherche à optimiser seulement la métrique $R = \sum_j R_j/R_{j,MAX}$, où $R_{j,MAX}$ est la quantité maximale de ressources matérielles de type j disponibles sur la plate-forme cible. ω_{V_+} et ω_T ont donc chacun la valeur zéro. Si n est le nombre de ressources matérielles, alors les coefficients de poids ω_{R_j} sont tous égaux à $1/n$. On obtient alors la fonction objective suivante pour le problème 8.3 :

$$o(a) = \frac{1}{n}\sum_{j=1}^{n}\frac{R_j(a)}{R_{j,MAX}} + \phi_{V_+}\left[\frac{\max(0, V_+(a) - V_{+,MAX})}{V_{+,MAX}}\right]^2 + \phi_T\left[\frac{\max(0, T(a) - T_{MAX})}{T_{MAX}}\right]^2 + \sum_{j=1}^{n}\phi_{R_j}\left[\frac{\max(0, R_j(a) - R_{j,MAX})}{R_{j,MAX}}\right]^2 \quad (8.5)$$

On observe que le premier terme de la fonction objective est égal à R/n. Cela assure que cette fonction objective cherche bel et bien à minimiser la métrique R du problème 8.3. En divisant la métrique R par n, on s'assure que le terme à optimiser soit, comme pour les deux fonctions objectives précédentes, égal à 1 lorsque cette métrique respecte tout juste la contrainte qui lui est associée.

Pour toutes ces fonctions objectives, on fixe les coefficients de pénalité ϕ_{V_+} et ϕ_T égaux à 1000 alors que les coefficients de pénalités ϕ_{R_j} sont fixés à $1000/n$.

8.4 Algorithmes d'exploration architecturale

Cette section présente différents algorithmes pour résoudre les problèmes d'exploration architecturale. On présente ainsi un algorithme exact de parcours en profondeur qui permet d'explorer exhaustivement l'espace de recherche et de trouver la solution optimale au problème. Cependant, une analyse de la complexité de l'exploration architecturale démontre que le nombre d'architectures possibles croît au moins exponentiellement et que ces problèmes sont fortement NP-difficiles. Il faut donc avoir recours à des algorithmes heuristiques pour les résoudre en un temps acceptable et cette section présente une heuristique gloutonne ainsi que des heuristiques de marche aléatoire, de descente, de recuit simulé adaptatif et de recherche

tabou réactive. L'annexe E présente également une heuristique pour la génération aléatoire d'une architecture selon une distribution uniforme. Tous ces algorithmes d'exploration architecturale utilisent les fonctions objectives définies à la section 8.3, qui sont évaluées à l'aide des méthodes d'estimation présentées au chapitre 7.

8.4.1 Complexité de l'exploration architecturale

La complexité des problèmes d'exploration architecturale définis à la section 8.2 peut se mesurer de deux manières différentes, soit selon la complexité combinatoire ou la complexité algorithmique. D'abord, l'espace de recherche de ces problèmes est un ensemble d'architectures et, dans le pire cas, il pourrait être nécessaire d'évaluer chacune de ces architectures pour obtenir une réponse exacte au problème. La complexité combinatoire d'un problème d'exploration architecturale indique donc le nombre total d'architectures qui constituent l'espace de recherche. L'annexe C présente une analyse combinatoire inédite qui démontre que le nombre $E(n)$ d'architectures possibles croît extrêmement rapidement avec le nombre n de modules de l'application. On obtient notamment que, pour l'espace de recherche f_1 défini à la section 8.1, $E(n)$ croît plus vite que la factorielle $n!$ pour $n \in [0, 147]$, comme le montre la figure 8.1 selon une échelle logarithmique pour $0 \leq n \leq 10$. Une application contenant seulement 10 modules a déjà plus de 379 millions d'architectures possibles selon f_1. Un algorithme exhaustif est donc viable seulement pour l'exploration d'une application comportant un nombre très restreint de modules.

La complexité algorithmique indique avec quelle efficacité des algorithmes peuvent résoudre ces problèmes. Tel que discuté à l'annexe D, ces problèmes sont fortement NP-difficiles et le temps d'exécution d'un algorithme exact croît donc exponentiellement avec la taille n du problème. Une contribution de l'annexe D est une nouvelle démonstration de la non-existence d'un schéma d'approximation en temps polynomial (PTAS) pour le partitionnement logiciel/matériel, et donc pour l'exploration architecturale. Il est donc nécessaire d'avoir recours à des algorithmes heuristiques pour résoudre ces problèmes en un temps acceptable.

8.4.2 Algorithme de parcours en profondeur

L'algorithme de parcours en profondeur construit et explore exhaustivement l'ensemble des solutions au problème d'exploration architecturale, comme le montre la figure 8.2 pour une fonction objective $o : A \to \mathbb{R}$. Il s'agit donc d'un algorithme exact, mais dont le temps d'exécution risque d'être prohibitif. Il serait possible d'améliorer cet algorithme avec une technique de séparation et évaluation (*branch and bound*). Ainsi, on pourrait par exemple choisir de ne générer et évaluer aucune solution pour un partitionnement logiciel/matériel donné

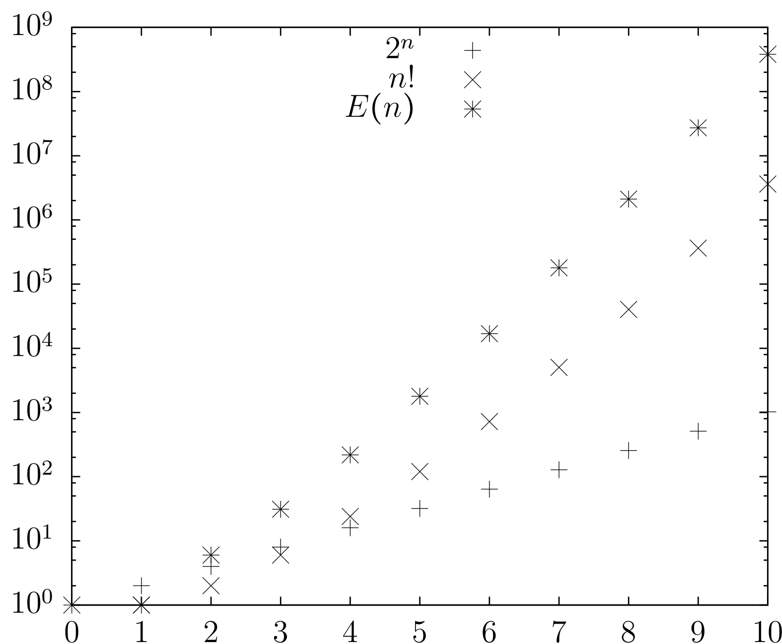

Figure 8.1 Comparaison de la croissance du nombre $E(n)$ d'architectures pour n modules avec celle de 2^n et $n!$

si on peut déterminer qu'aucune de ces solutions n'est meilleure que la meilleure solution trouvée jusqu'à date. Il faudrait par contre dans ce cas avoir un estimateur capable d'évaluer une borne inférieure pour l'ensemble des solutions ayant un partitionnement logiciel/matériel donné quel que soit le nombre de processeurs, l'assignation des tâches aux processeurs et la topologie de communications. De plus, l'ajout d'une telle technique ne change pas le caractère fortement NP-difficile du problème et le temps d'exécution de cet algorithme croît donc aussi de manière exponentielle avec le nombre de modules.

```
Pour chaque partitionnement logiciel/matériel des modules
  Pour chaque allocation de processeurs
    Pour chaque assignation des tâches et modules SW aux processeurs
      Pour chaque topologie de communication
        Construire l'architecture "a" correspondant à cette combinaison
Retourner l'architecture "a" visitée qui avait le plus petit o(a)
```

Figure 8.2 Fonctionnement général de l'algorithme de parcours en profondeur

8.4.3 Algorithme glouton

Un algorithme glouton construit une solution en une série d'étapes. À chaque étape, l'algorithme glouton effectue le choix qui est localement optimal pour cette étape sans revenir sur les choix faits lors des étapes précédentes. Étant donné que les choix faits aux différentes étapes ne sont généralement pas indépendants entre eux, cette série de choix localement optimaux produit rarement l'optimum global. Néanmoins, un algorithme glouton permet de générer rapidement une solution qui est meilleure en moyenne qu'une solution générée aléatoirement. Cette solution gloutonne peut ensuite être utilisée comme solution initiale dans un algorithme itératif plus élaboré, telle que la recherche locale ou les algorithmes génétiques.

Cette section présente un algorithme glouton pour les problèmes d'exploration architecturale de ce chapitre. Soit une application avec des ensembles M de modules, S de tâches logicielles et H de composants matériels. L'algorithme glouton construit une architecture en y ajoutant un par un chacun des composants, tâches et modules de l'application. Pour des raisons de simplification, l'algorithme glouton ne considère que les topologies composées d'un seul bus partagé.

L'ordre dans lequel les composants, tâches et modules sont assignés est déterminé à l'aide des métriques présentées au tableau 8.1. Ce tableau introduit notamment la métrique d'affinité matérielle : plus un module a une accélération élevée lorsqu'on le déplace du logiciel vers le matériel et moins il consomme de ressources matérielles, plus son affinité envers une implémentation matérielle est grande et, à l'opposé, plus son affinité envers une implémentation logicielle est faible.

Les valeurs des métriques $t_{sw}(x)$, $t_{hw}(x)$ et $r_j(x)$ sont obtenues par la caractérisation et l'estimation présentées au chapitre 7. Les métriques de quantité de ressources matérielles sont combinées selon des coefficients de poids en une métrique globale de quantité de ressources matérielles tel que décrit à la section 8.2.

Étant donné que l'heuristique gloutonne doit évaluer des implémentations partiellement construites, on adapte les méthodes d'estimation définies au chapitre 7. Ainsi, on considère comme nul le temps d'exécution, les temps de communication et la quantité de ressources matérielles de tout composant, tâche ou module qui n'a pas encore été assigné par l'algorithme glouton. L'estimation des métriques $V(a)$, $T(a)$ et $R_j(a)$ est donc ajustée en conséquence. On utilise pour chaque problème la même fonction objective que celle définie pour le problème à la section 8.3.

Le fonctionnement général de l'algorithme glouton pour l'exploration architecturale est présenté à la figure 8.3. L'initialisation de la solution consiste d'abord à instancier une architecture vide, à allouer un bus et à y assigner l'ensemble H des composants matériels. Ensuite,

Tableau 8.1 Définitions des métriques appliquées aux modules

Nom	Description
$t_{sw}(x)$	Le temps total que prend l'ensemble des calculs réalisés par la tâche logicielle x ou le module x lorsque celui-ci se trouve en logiciel.
$t_{hw}(x)$	Le temps total que prend l'ensemble des calculs réalisés par le composant matériel x ou le module x lorsque celui-ci se trouve en matériel.
$r_j(x)$	La quantité de ressources matérielles de type j utilisées par le composant x ou le module x lorsque celui-ci se trouve en matériel.
$r(x) = \sum_j \alpha_j r_j(x)$	Une évaluation de l'ensemble des ressources matérielles utilisées par le composant ou module x, pondérée par les coefficients α_j.
$hw(m) = \frac{t_{sw}(m) - t_{hw}(m)}{r(m)}$	Le degré d'affinité du module m envers une implémentation matérielle.

l'initialisation diffère selon que l'algorithme soit configuré pour avoir un biais matériel ou un biais logiciel. L'ensemble M est trié par ordre décroissant des affinités matérielles (pour considérer d'abord les modules avec les plus fortes affinités hw) si l'algorithme a un biais matériel et par ordre croissant si il a un biais logiciel. De plus, si l'algorithme glouton a un biais logiciel, alors le module de M avec le plus petit hw est retiré de M pour l'ajouter à S et donc forcer son implémentation en logiciel. Sans cette mesure, l'algorithme glouton tendrait à ne jamais allouer un premier processeur à moins que la quantité de ressources matérielles nécessaires à l'implémentation matérielle d'un seul module soit supérieure à celles nécessaires au processeur. L'initialisation se termine en triant l'ensemble S par ordre croissant de leur t_{sw}.

L'algorithme glouton ajoute ensuite successivement chaque tâche $s \in S$ à la solution. Il peut ajouter une tâche s donnée soit en l'ajoutant à un processeur déjà alloué, soit en allouant un nouveau processeur. Toutes ces possibilités sont évaluées et triées selon la fonction objective. De la même manière, l'algorithme ajoute à la solution chacun des modules $n \in N$ soit en ajoutant son implémentation matérielle au bus, soit en ajoutant son implémentation logicielle à un processeur déjà alloué ou à un processeur simultanément alloué. La construction de la solution est terminée alors que tous les éléments de H, S et M ont été assignés.

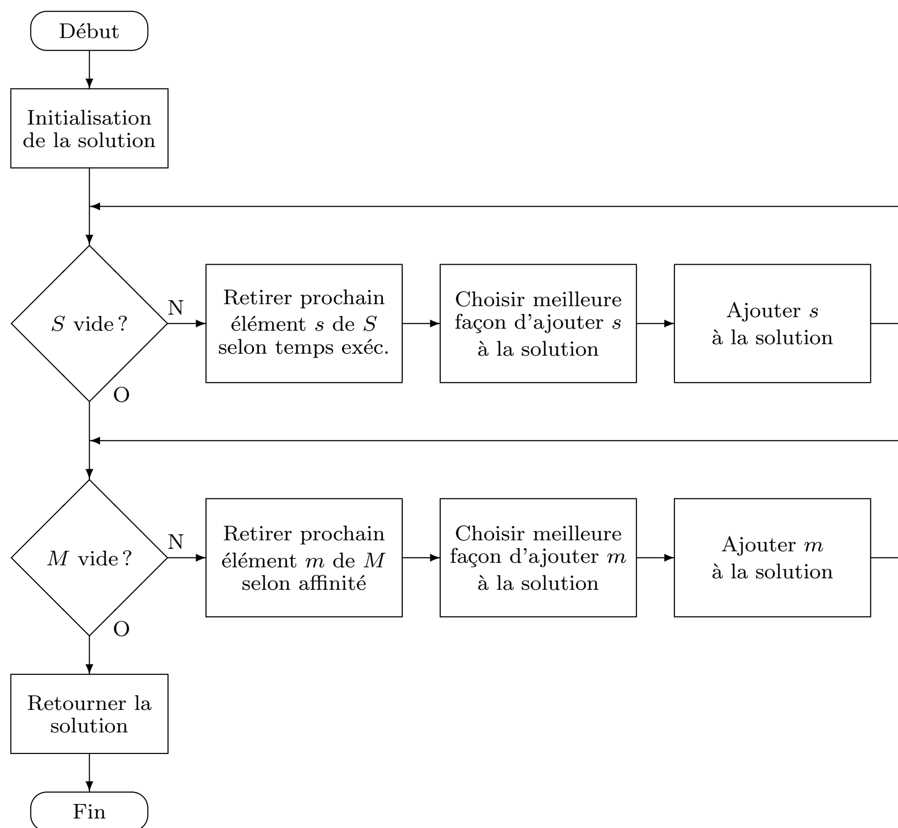

Figure 8.3 Fonctionnement général de l'algorithme glouton

8.4.4 Recherche locale

L'algorithme glouton présenté à la section précédente est un algorithme constructif, qui construit une solution morceau par morceau jusqu'à ce qu'elle soit complétée. Les algorithmes présentés dans cette section sont plutôt des algorithmes itératifs, qui partent d'une solution complète initiale et qui la raffinent itérativement. Il est possible que la solution finale d'un tel algorithme itératif ait bien peu à voir avec la solution initiale. Plus précisément, ces algorithmes sont des algorithmes de recherche locale : à chaque itération, on passe de la solution courante à une de ses solutions voisines. Ces algorithmes de recherche locale se distinguent entre eux notamment par leur manière de choisir la prochaine solution parmi les solutions voisines et par leur critère d'arrêt.

Bien que la descente, le recuit simulé et la recherche tabou soient des méta-heuristiques génériques de recherche locale qui sont bien connues, elles doivent être spécialisées selon les caractéristiques d'un problème donné avant de pouvoir être appliquées comme heuristiques à ce problème. Notre contribution dans cette section est donc principalement la spécialisation de ces méta-heuristiques en des heuristiques applicables aux problèmes d'exploration archi-

tecturale. En particulier, on présente une nouvelle formulation de ces problèmes comme un problème de recherche locale et ce qui est, au meilleur de notre connaissance, la première application d'un recuit simulé adaptatif, d'une recherche tabou réactive ou d'une liste tabou par attributs à l'exploration architecturale.

8.4.4.1 Définition du voisinage

Étant donné que la recherche locale doit évaluer les voisines de la solution courante et y choisir une solution pour la prochaine itération, il est crucial de définir pour l'exploration architecturale une fonction de voisinage $N : A \to \wp(A)$ qui permet d'associer, à toute architecture donnée, l'ensemble de ses architectures voisines.

Le voisinage est défini selon un ensemble de mouvements $m \in [A \to A]$ qui permettent de passer d'une architecture à une autre. Ainsi, le voisinage d'une architecture a est l'ensemble des architectures $N(a)$ qu'on peut obtenir en effectuant un mouvement $m(a)$ à partir de la solution a.

Dans le cas de l'espace de recherche défini par f_1 et f_2, le voisinage $N(a)$ de a est défini comme l'ensemble des architectures qu'on peut atteindre en effectuant un des mouvements suivants (on considère ici que l'implémentation logicielle d'un module est une tâche logicielle et que son implémentation matérielle est un composant matériel ; les périphériques et les processeurs sont également considérés comme des composants matériels) :

1. Assigner un composant matériel double-port à un deuxième bus (f_2 seulement) ;
2. Retirer l'assignation d'un composant matériel double-port à un de ses deux bus (f_2 seulement) ;
3. Déplacer un composant matériel d'un bus à un autre ;
4. Déplacer une tâche logicielle d'un processeur à un autre ;
5. Retirer d'un processeur l'implémentation logicielle d'un module et assigner à un bus l'implémentation matérielle de ce même module ;
6. Retirer d'un bus l'implémentation matérielle d'un module et assigner à un processeur l'implémentation logicielle de ce même module ;
7. Retirer un bus sur lequel sont assignés exactement un processeur ou deux composants matériels autres que des processeurs et assigner ces composants matériels à un autre bus ;
8. Allouer un bus, retirer des autres bus exactement un processeur ou deux composants matériels autres que des processeurs et les assigner au bus alloué ;
9. Retirer un processeur sur lequel est assigné exactement une tâche logicielle et assigner celle-ci à un autre processeur ;
10. Retirer un processeur sur lequel est assigné exactement une implémentation logicielle

d'un module et assigner l'implémentation matérielle de ce même module à un bus ;

11. Allouer un processeur, retirer d'un autre processeur exactement une tâche logicielle et assigner celle-ci au processeur alloué ;
12. Allouer un processeur, retirer d'un bus exactement une implémentation matérielle d'un module et assigner l'implémentation logicielle de ce même module au processeur.

Dans tous les cas, un mouvement m est admissible pour une architecture a si et seulement si l'architecture $m(a)$ qu'il génère respecte les contraintes présentées à la section 8.1 (par exemple, au moins une tâche logicielle doit être assignée à chaque processeur présent dans l'architecture). Ainsi, seules les architectures qui respectent ces contraintes font partie du voisinage $N(a)$ d'une implémentation a. Les mouvements définis ci-haut permettent d'explorer le partitionnement logiciel/matériel, l'allocation des processeurs, l'assignation des tâches logicielles aux processeurs, l'allocation des bus et l'assignation des composants matériels aux bus tout en respectant ces contraintes.

Afin de comparer entre eux les algorithmes heuristiques de recherche locale, on suppose que chacun d'entre eux peut être paramétré de manière à s'arrêter après (environ) k itérations, où chaque itération est équivalente à l'exploration d'un voisinage d'une solution courante.

8.4.4.2 Marche aléatoire

La marche aléatoire avec paramètre n effectue n mouvements aléatoires à partir d'une solution initiale a_0. Ainsi, on a une séquence $a_1 = m_0(a_0), a_2 = m_1(a_1), \ldots, a_n = m_{n-1}(a_{n-1})$ où les mouvements $m_0, m_1, \ldots, m_{n-1}$ sont choisis aléatoirement respectivement parmi les voisinages $N(a_0), N(a_1), \ldots, N(a_{n-1})$. Ainsi, le résultat de la marche aléatoire est la meilleure solution visitée (selon la fonction objective) parmi la séquence $a_0, a_1, \ldots, a_n$.

Étant donné que la marche aléatoire choisit les mouvements à effectuer de manière complètement aléatoire, on s'attend à ce que cet algorithme soit peu efficace. Comme la marche aléatoire n'explore pas systématiquement le voisinage de la solution courante, mais considère une seule solution voisine, on obtient, pour fins de comparaison avec les autres algorithmes, le nombre d'itérations n en multipliant le nombre k de voisinages à explorer par $|N(a_0)|$, soit le nombre de solutions voisines à la solution initiale.

8.4.4.3 Descente

Soit une fonction objective $o : A \to \mathbb{R}$. L'algorithme de descente, tel qu'illustré à la figure 8.4, consiste simplement à prendre une solution initiale a_0 et à effectuer sur celle-ci une série de mouvements $a_1 = m_0(a_0), a_2 = m_1(a_1), \ldots$ qui améliorent chacun la solution courante et donc tel que $o(a_0) > o(a_1) > o(a_2) > \ldots$. L'algorithme de descente se termine lorsqu'il

est impossible de trouver un mouvement qui améliore la solution courante et cette dernière solution est alors le résultat de la descente.

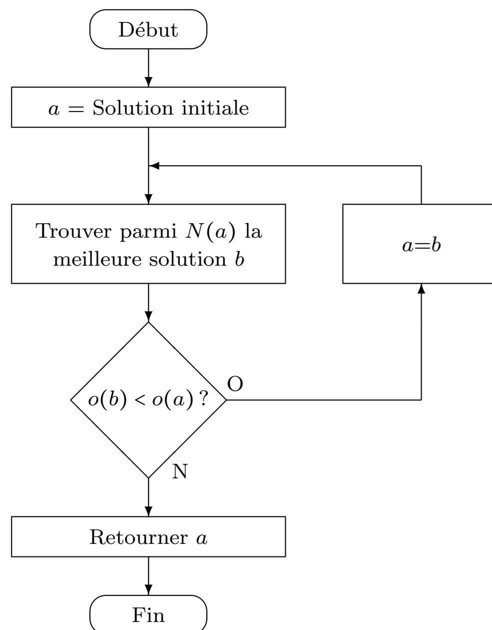

Figure 8.4 Fonctionnement général de l'algorithme de descente

Si on prend un relief de telle sorte que l'espace de recherche est un ensemble de positions géographiques et que la fonction objective est une altitude, alors l'algorithme de descente équivaut à laisser tomber une balle dans le relief et à la laisser descendre. Le point atteint par l'algorithme de descente est un optimum local, étant donné que cette solution est au moins aussi bonne que toutes ses solutions voisines. Cependant, rien ne garantit que cette solution soit un optimum global. En effet, il est possible que l'optimum global se trouve ailleurs et que le seul moyen de l'atteindre à partir d'un autre optimum local soit par une série de mouvements qui entraînent initialement une dégradation de la solution courante. Par analogie avec le relief, il est possible qu'il faille gravir une montagne pour atteindre une vallée plus profonde de l'autre côté. C'est ce que permettent, chacun à leur manière, le recuit simulé et la recherche tabou.

Étant donné que la descente a un critère d'arrêt qui est indépendant du nombre d'itérations et qu'en pratique elle s'arrête après un nombre restreint d'itérations, le nombre d'itérations de l'algorithme de descente conçu pour l'exploration architecturale ne peut pas être paramétré.

8.4.4.4 Recuit simulé adaptatif

Le recuit simulé est un algorithme heuristique de recherche locale qui s'inspire du procédé du recuit en métallurgie. Dans un recuit, on commence par chauffer le métal à très haute température de sorte que ses atomes ont assez d'énergie pour une diffusion aléatoire à l'état solide. À mesure qu'on refroidit de manière contrôlée le métal, ses atomes tendent de moins en moins à se diffuser aléatoirement et de plus en plus à se placer en une structure cristalline de manière à minimiser l'énergie totale interne du métal, notamment en minimisant les imperfections de celui-ci.

De manière analogue, dans un recuit simulé, la « température » est un paramètre qui détermine la probabilité avec laquelle est acceptée un mouvement aléatoire qui n'améliore pas la solution courante. Lorsque la température est très élevée, tous les mouvements aléatoires sont acceptés et le recuit simulé se comporte comme une marche aléatoire. À l'opposé, lorsque la température est très basse, les seuls mouvements acceptés sont ceux qui améliorent la solution et le recuit simulé se comporte alors comme un algorithme de descente. Le recuit simulé commence donc par explorer l'espace de recherche de manière très large mais très grossière à haute température, puis raffine son exploration dans les régions prometteuses à mesure que la température est refroidie (van Laarhoven et Aarts, 1987).

Le recuit simulé s'applique à l'exploration architecturale pour n itérations tel qu'illustré à la figure 8.5 (comme dans le cas de la marche aléatoire à la section 8.4.4.2, on obtient n en multipliant le paramètre k par le nombre de solutions voisines à la solution initiale). Pour une fonction objective $o : A \to \mathbb{R}$, le critère d'acceptation utilisé est le critère de Metropolis qui accepte toujours un mouvement de a vers b si il améliore la solution courante (donc si $o(b) - o(a) < 0$) et qui l'accepte avec une probabilité $e^{\frac{o(a)-o(b)}{T}}$ sinon (Metropolis *et al.*, 1953). Il reste alors à déterminer la température initiale T_0 et le schéma de refroidissement (mise à jour de la température). Ces décisions ont un grand impact sur la qualité des solutions produites et sur le temps d'exploration du recuit simulé. Il est fréquent qu'une version du recuit simulé utilise un schéma de refroidissement géométrique. Il y a alors un paramètre de refroidissement c de telle sorte que la température à l'itération j est égale à $T_j = T_0 c^j$. Un schéma de refroidissement par paliers de taille p peut aussi être utilisé : la température reste alors constante pendant des paliers de p itérations et on a $T_j = T_0 c^{\lfloor \frac{j}{p} \rfloor}$. Cependant, il reste encore à déterminer des bonnes valeurs pour T_j et c. Cette calibration du recuit simulé pour une instance d'un problème peut demander de nombreux tests et il n'est pas garanti que cette calibration soit correcte pour les autres instances du même problème.

Nous effectuons la calibration de la température initiale T_0 de manière à ce qu'un mouvement qui n'améliore pas la solution initiale a_0 soit accepté avec une probabilité de 50%. Ainsi, si $D = \{d \in N(a_0) | o(d) > o(a_0)\}$, alors on obtient :

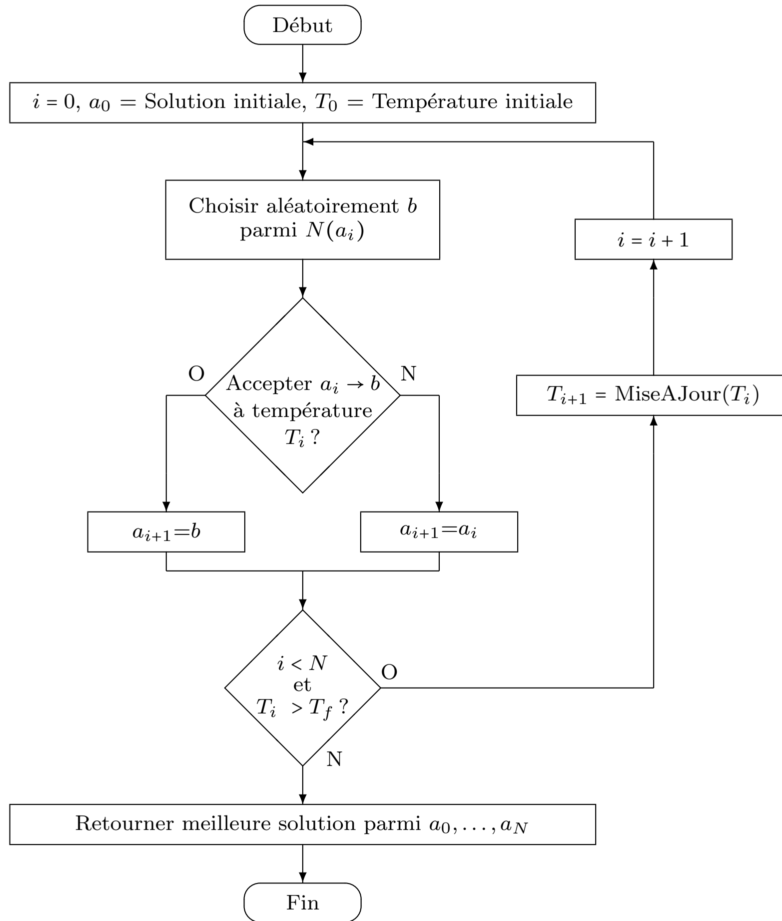

Figure 8.5 Fonctionnement général de l'algorithme de recuit simulé

$$T_0 = \frac{1}{\ln 2} \left[\frac{\sum_{d \in D} (o(d) - o(a_0))}{|D|} \right] \tag{8.6}$$

Pour le paramètre de refroidissement c, nous utilisons un recuit simulé adaptatif qui permet de calibrer c tout au long de l'exécution du recuit simulé (Ortner, 2004; Perrin *et al.*, 2005). Le recuit simulé adaptatif est basé sur un schéma de refroidissement géométrique par paliers. Pour l'exploration architecturale, on rend donc le nombre de paliers égal au paramètre k de voisinages à explorer et la taille p de chaque palier égale au nombre $|N(a_0)|$ de solutions voisines à la solution initiale. Le paramètre de refroidissement initial c_0 est fixé à 0.99 et l'algorithme s'arrête lorsqu'on atteint la température finale $T_f = T_0 c^k$ qu'atteindrait l'algorithme après k paliers si le paramètre de refroidissement demeurait constant. Cependant, la particularité du recuit simulé adaptatif est que ce paramètre varie au cours de son exécution (Ortner, 2004; Perrin *et al.*, 2005).

Ainsi, on calcule, pour chaque palier, la valeur moyenne de la fonction objective pour l'ensemble des solutions visitées pendant ce palier. Par analogie avec le recuit, cela correspond à « l'énergie » moyenne du palier. On divise aussi chaque palier en 10 sous-paliers de taille égale. La température et le paramètre de refroidissement sont mis à jour à la fin de chaque palier et on fixe un paramètre d'accélération r égal à 0.85. Si plus de 5 sous-paliers sur 10 dans le palier j ont une énergie moyenne supérieure à celle du palier $j-1$, alors on diminue la température ($T_j = T_{j-1}c_{j-1}$) et on accélère le refroidissement ($c_j = c_{j-1}^{1/r}$). Si aucun des sous-paliers n'a une énergie moyenne supérieure à celle du palier précédent, alors on augmente la température ($T_j = T_{j-1}/c_{j-1}$) et on ralentit le refroidissement ($c_j = c_{j-1}^{r}$). Si aucun de ces deux cas ne se produit, on diminue seulement la température ($T_j = T_{j-1}c_{j-1}$) comme dans le recuit simulé traditionnel. Dans tous les cas, on ajuste les valeurs de c_j pour qu'elles restent bornées entre 0.96 et 0.996. Cela empêche que le paramètre de refroidissement s'approche trop de 1 et que l'algorithme prenne alors un temps excessif à s'exécuter.

Cette adaptation du paramètre de refroidissement découle de l'observation selon laquelle il est préférable que le recuit simulé explore plus longuement certaines températures qu'on pourrait qualifier de critiques et qu'il passe plus rapidement d'autres températures, généralement au début ou à la fin, qui produisent des résultats moins intéressants. Ainsi, si la plupart des solutions d'un palier sont moins bonnes que celles du palier précédent, c'est soit qu'on est dans une situation de marche aléatoire à température élevée ou dans un optimum local à température basse. On peut donc accélérer l'algorithme pour atteindre plus rapidement les températures intéressantes dans le premier cas ou la fin de l'algorithme dans le deuxième cas. D'un autre côté, si une forte majorité des solutions d'un palier sont meilleures que celles du palier précédent, l'algorithme est probablement en train de se diriger trop rapidement vers un optimum local. En augmentant la température et en ralentissant le refroidissement, on s'assure que l'algorithme continue d'explorer plus amplement l'espace de recherche en évitant de rester coincé dans un optimum local (Ortner, 2004; Perrin *et al.*, 2005).

Une conséquence du caractère adaptatif de cet algorithme est qu'il n'est plus possible de prédire le nombre de paliers et donc le nombre d'itérations qu'effectuera l'algorithme avant de s'arrêter. Il est possible que le nombre d'itérations soit réduit si le refroidissement est surtout accéléré. De la même manière, nous avons observé que la température peut tendre à osciller autour d'une température critique au gré des augmentations et diminutions de température : le nombre d'itérations est alors grandement augmenté. Pour borner le nombre d'itérations, nous avons donc modifié le recuit simulé adaptatif de (Ortner, 2004; Perrin *et al.*, 2005) pour qu'il se comporte comme un recuit simulé traditionnel une fois que le nombre k de paliers est dépassé.

8.4.4.5 Recherche tabou

Pour qu'une heuristique de recherche locale puisse échapper à un optimum local, il faut qu'elle accepte parfois une dégradation de la solution courante. On pourrait naïvement modifier l'heuristique de descente de telle sorte que la solution courante à l'itération $k+1$ est la meilleure solution voisine de la solution courante à l'itération k. Ainsi, même si cette heuristique tombait sur un optimum local à l'itération k, rien ne l'empêcherait d'explorer une nouvelle solution à l'itération $k+1$ qui soit moins bonne que l'optimum local. Cependant, cette heuristique retournerait fort probablement au même optimum local à l'itération $k+2$ et tournerait ainsi en rond.

Une telle heuristique fonctionnerait si on modifiait le voisinage de la solution courante à l'itération $k+1$ pour que l'optimum local de l'itération k n'en fasse plus partie. C'est le principe de la recherche tabou, qui utilise une mémoire à court terme, aussi appelée liste tabou, pour éviter de tourner en rond et de revenir sur des solutions récemment explorées. En interdisant les mouvements qui vont à l'encontre des mouvements récemment effectués, la recherche tabou peut ainsi échapper à un optimum local et poursuivre l'exploration de l'espace de recherche (Glover et Laguna, 1997).

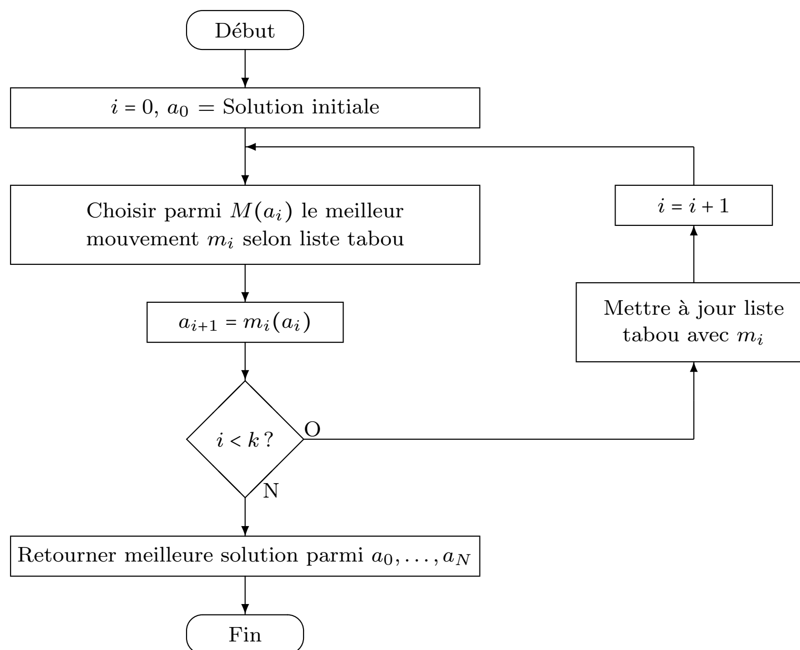

Figure 8.6 Fonctionnement général de l'algorithme de recherche tabou

La figure 8.6 illustre le fonctionnement général de la recherche tabou pour k itérations. La fonction $M : A \to \wp[A \to A]$ est définie, pour tout $a \in A$, comme l'ensemble des mouvements qui permet d'obtenir l'ensemble $N(a)$ des solutions voisines à a. Pour tout $a \in A$, l'ensemble

$M(a)$ se divise en deux sous-ensembles disjoints selon la liste tabou : l'ensemble $M_t(a)$ des mouvements tabous et l'ensemble $M_n(a)$ des mouvements non-tabous. Pour une fonction objective $o : A \rightarrow \mathbb{R}$ et une itération i lors de la recherche tabou, on choisit généralement comme mouvement m_i le meilleur mouvement non-tabou, soit un mouvement $m \in M_n(a_i)$ tel que $o(m(a_i)) \leq o(x(a_i))$ pour tout $x \in M_n(a_i)$, et ce même si m_i dégrade la solution courante avec $o(m_i(a_i)) > o(a_i)$. Deux exceptions peuvent se produire. D'abord, un mouvement tabou $m \in M_t(a_i)$ peut répondre au critère d'aspiration : si la solution produite suite au mouvement m est meilleure que la meilleure solution explorée jusqu'à date, alors ce mouvement peut être choisi comme m_i même si il est tabou. Ensuite, si tous les mouvements sont tabous, soit $M_n(a_i) = \varnothing$, alors le meilleur mouvement tabou est choisi. On effectue ensuite le mouvement m_i choisi sur la solution courante, puis on passe à la prochaine itération après avoir mis à jour la liste tabou avec ce mouvement.

La principale décision à prendre lors de la mise en oeuvre d'un algorithme de recherche tabou consiste à déterminer le fonctionnement de la liste tabou. La liste tabou la plus simple conserve seulement en mémoire les r dernières solutions visitées, où r est un paramètre entier. Si une solution a été visitée dans les r dernières itérations, alors tout mouvement qui mène à cette solution est tabou. Une variante plus sophistiquée de la recherche tabou se base sur les attributs des solutions : la liste tabou contient alors les attributs ajoutés à la solution lors des r_{OUT} dernières itérations et ceux retirés à la solution lors des r_{IN} dernières itérations. Lorsqu'un attribut est ajouté à la solution courante, tout mouvement qui retirerait cet attribut devient tabou pour les r_{OUT} prochaines itérations et, lorsqu'un attribut est retiré, tout mouvement qui l'ajouterait devient tabou pour les r_{IN} prochaines itérations. Étant donné qu'il y a généralement beaucoup plus d'attributs qui ne sont pas présents dans une solution donnée qu'il y a en de présents, il est plus restrictif de rendre tabou le retrait d'un attribut présent que l'ajout d'un attribut absent. C'est pourquoi on choisit généralement $r_{OUT} < r_{IN}$ pour équilibrer l'impact de la liste tabou sur les ajouts et les retraits d'attributs.

Les attributs retenus pour les problèmes d'exploration architecturale présentés à la section 8.2 sont basés sur la définition de l'espace de recherche exposée à la section 8.1. Ainsi, les attributs décrits au tableau 8.2 se rapportent au partitionnement logiciel/matériel, à l'allocation des processeurs, à l'assignation des tâches aux processeurs, à l'allocation des bus et à l'assignation des composants aux bus qui prévalent dans une solution donnée.

Par exemple, si on effectue sur une solution donnée un mouvement qui retire d'un bus b_1 l'implémentation matérielle du module x_0 pour la remplacer par l'implémentation logicielle du même module sur un processeur p assigné à un autre bus b_2, alors :

1. On retire, pour chaque autre composant, module ou tâche x_1 assigné au bus b_1 ou à un processeur assigné au bus b_1, l'attribut sb_{x_0,x_1} ;

2. On ajoute l'attribut sw_{x_0} ;
3. On ajoute l'attribut $sp_{x_0,t}$ pour chaque autre tâche t sur le processeur p ;
4. On ajoute l'attribut sb_{x_0,x_2} pour chaque composant, module ou tâche x_2 assigné au bus b_2 ou à un processeur assigné au bus b_2.

Tableau 8.2 Définitions des attributs pour la recherche tabou

Nom	**Description**
sw_x	Cet attribut est présent lorsque le module x est implémenté en logiciel.
ap_p	Cet attribut est présent lorsque le processeur p est alloué.
sp_{t_1,t_2}	Cet attribut est présent lorsque les tâches t_1 et t_2 sont assignées au même processeur.
ab_b	Cet attribut est présent lorsque le bus b est alloué.
sb_{x_1,x_2}	Cet attribut est présent lorsque les composants, tâches ou modules x_1 et x_2 sont assignés au même bus soit directement, soit indirectement (si x_1 ou x_2 est une tâche assignée sur un processeur assigné à ce bus).

8.4.4.6 Recherche tabou réactive

Une fois que le fonctionnement de la liste tabou est fixé, il reste à en déterminer la longueur, soit les paramètres r_{IN} et r_{OUT}. Si la valeur de ces paramètres est trop petite, alors la recherche tabou risque de rester prise dans un cycle et de ne pas pouvoir explorer plusieurs régions de l'espace de recherche. Par contre, si la valeur de ces paramètres est trop grande, alors la recherche tabou risque de passer à côté d'un optimum local qui pourrait être l'optimum global ou près de l'être. On peut tenter, par essai et erreur, de trouver les meilleures valeurs de r_{IN} et r_{OUT} pour une instance d'un problème. Cependant, ces valeurs ne seront pas nécessairement les mêmes pour toutes les instances du problème. De plus, même pour une instance donnée, il est possible qu'il faille diminuer ou augmenter la longueur de la liste tabou au cours de la recherche tabou.

La recherche tabou réactive (Battiti et Tecchiolli, 1994, 1995) s'attaque à ces problèmes via l'ajustement dynamique de la longueur de la liste tabou, tel qu'illustré à la figure 8.7. Ainsi, cet algorithme a une mémoire à long terme qui contient l'identité de la solution courante pour chacune des itérations précédentes. Lorsqu'une solution est visitée de nouveau, l'algorithme détecte ainsi l'existence d'une répétition et, si la longueur de la répétition (le nombre d'itérations séparant ces deux visites) est inférieure à un seuil l^*, alors la longueur de la liste tabou est augmentée. Inversement, si aucune répétition n'est détectée pendant un nombre d'itérations égal au double de la moyenne mobile de la longueur des 10 dernières répétitions inférieures à l^*, alors la longueur de la liste tabou est diminuée. Cette longueur est également diminuée si tous les voisins de la solution courante sont tabous.

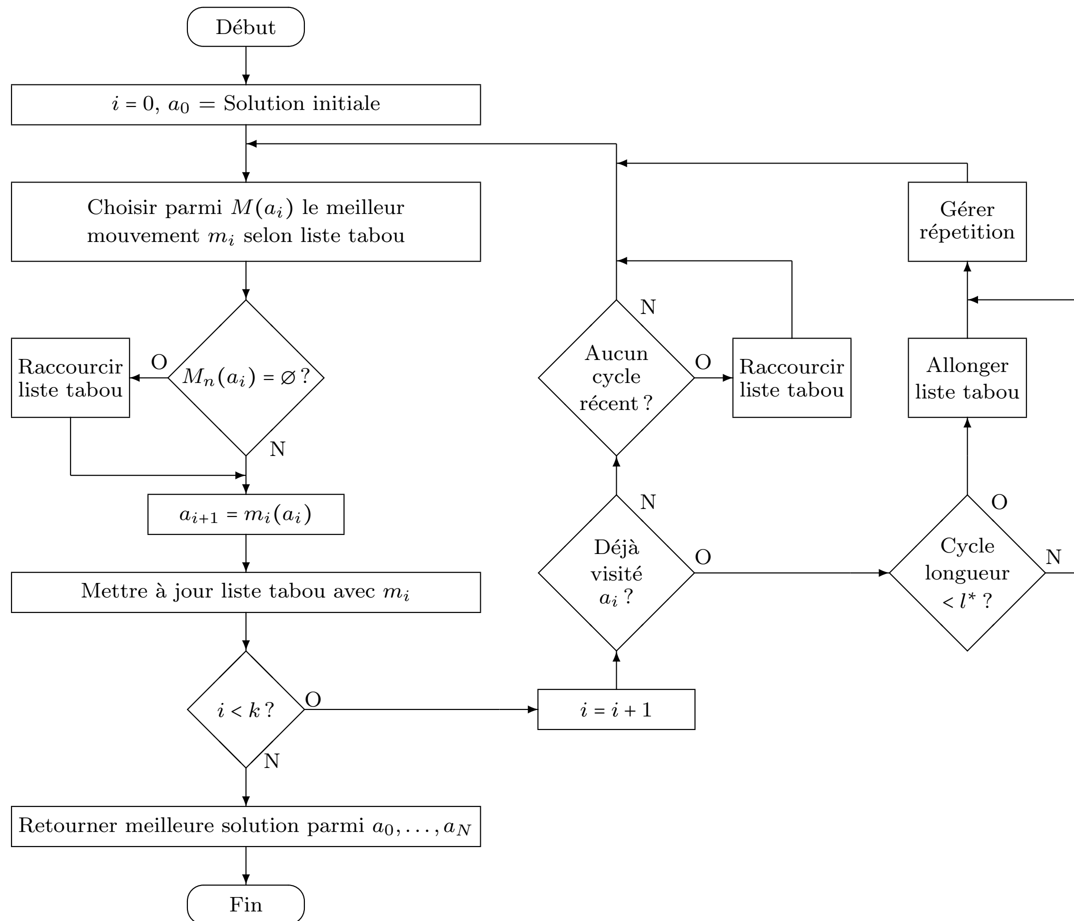

Figure 8.7 Fonctionnement général de l'algorithme de recherche tabou réactive

De plus, la recherche tabou réactive a recours à un mécanisme de diversification si elle constate que l'exploration reste confinée dans une seule région de l'espace de recherche. Cela peut se produire si la recherche effectue un cycle, mais aussi si la recherche tabou tourne autour d'un certain nombre de solutions attrayantes sans nécessairement faire un cycle. En effet, visiter la même solution à deux itérations différentes n'implique pas nécessairement un cycle, étant donné que la liste de mouvements tabous et donc le voisinage non tabou ne seront pas nécessairement identiques à ces deux itérations. Ainsi, si on détecte qu'une solution se répète un nombre de fois supérieur à un seuil s, alors cette solution est considérée comme une solution fréquente. Si le nombre de solutions fréquentes devient supérieur à un seuil c_{MAX}, alors l'algorithme détecte que la recherche est probablement confinée à une région de l'espace de recherche. Pour permettre l'exploration d'une nouvelle région, l'algorithme effectue alors une marche aléatoire avec un nombre d'itérations basé sur la moyenne mobile de la longueur des 10 dernières répétitions inférieures à l^*. La liste tabou est mise à jour avec les mouvements effectués lors de la marche aléatoire. La mémoire à long terme est réinitialisée

suite à la marche aléatoire (on conserve néanmoins la meilleure solution visitée jusqu'à date). Comme dans (Battiti et Tecchiolli, 1994, 1995), on fixe $s = 3$ et $c_{MAX} = 3$.

Bien que l'algorithme de (Battiti et Tecchiolli, 1994, 1995) permette l'ajustement dynamique de la longueur de la liste tabou, il fait apparaître un nouveau problème : la nécessité de fixer la bonne valeur du seuil l^*. En effet, si cette valeur est trop petite, la longueur de la liste tabou restera toujours petite et la recherche tabou réactive tolèrera des cycles relativement courts. Si cette valeur est trop grande, la recherche tabou tendra à fréquemment augmenter la longueur de la liste tabou et à rarement la diminuer, ce qui amènera la longueur de la liste tabou à toujours demeurer grande. À l'extrême, la longueur de la liste tabou peut devenir tellement grande qu'il devient fréquent que tous les voisins de la solution courante soient tabous et la liste tabou perd alors sa pertinence.

Nous avons amélioré la recherche tabou réactive en rendant la valeur du seuil l^* elle aussi dynamiquement ajustable. Cet ajustement se fait après chaque phase de diversification selon la fraction des itérations où tous les mouvements ont été tabous. Si cette fraction est supérieure à un seuil f_H, alors la valeur du seuil l^* est diminuée pour éviter que la longueur de la liste tabou devienne trop souvent trop élevée. Si cette fraction est inférieure à un deuxième seuil f_L, alors la valeur du seuil l^* est augmentée car il s'agit d'un signe que la longueur de la liste tabou tend à rester courte. Cette modification amène le problème de fixer les bonnes valeurs de f_L et f_H. Cela est néanmoins une amélioration, car le nombre d'itérations entre deux répétitions est une métrique qui ne tient compte ni de la structure de la liste tabou, ni de la définition du voisinage. La bonne valeur de l^* dépend ainsi de la structure de la liste tabou, de la définition du voisinage et donc de l'instance du problème. À l'inverse, la fraction des itérations où tous les mouvements sont tabous est une métrique qui tient compte à la fois de la structure de la liste tabou et de la définition du voisinage. Il devient donc possible de trouver des bonnes valeurs pour f_L et f_H qui s'appliquent à plusieurs structures de liste tabou, définitions de voisinage et instances de problème. Nous avons observé que des valeurs de $f_L = 0.01$ et $f_H = 0.05$ donnent de bons résultats.

L'autre amélioration apportée à l'algorithme de (Battiti et Tecchiolli, 1994, 1995) est que la liste tabou utilisée ici est plus sophistiquée. En effet, la liste tabou de (Battiti et Tecchiolli, 1994, 1995) considère simplement tabou l'inverse des mouvements effectués au cours des r dernières itérations. La liste tabou utilisée ici a deux longueurs différentes r_{OUT} et r_{IN} pour rendre séparément tabou le retrait et l'ajout d'attributs à la solution. Cela implique que notre variante de la recherche tabou réactive doit être capable d'adapter séparément ces longueurs r_{OUT} et r_{IN}. Ces longueurs sont simultanément augmentées ou diminuées en les multipliant par des facteurs identiques, sauf dans le cas où tous les mouvements sont tabous. On vérifie alors si c'est le retrait ou l'ajout d'attributs tabous qui rend tabous le plus grand nombre de

mouvements du voisinage. Dans le premier cas, seul r_{OUT} est diminué et seul r_{IN} est diminué dans le deuxième cas.

CHAPITRE 9

RÉSULTATS ET DISCUSSION

Les différentes étapes de la méthodologie présentée dans cette thèse sont appliquées à trois études de cas : le système de guidage d'un rover (astromobile) simplifié, un décodeur d'images JPEG avec détection de la peau et un codeur/décodeur pour la couche physique du standard de communications sans fil WiMAX. On présente d'abord ces trois applications, puis on indique comment elles sont modélisées à l'aide du modèle de calcul des réseaux de processus temps-réel. On présente ensuite des résultats sur la synthèse matérielle et le raffinement des communications, sur le profilage au niveau système, sur la caractérisation et l'estimation ainsi que sur l'exploration architecturale pour ces études de cas. Les temps nécessaires à l'exécution des simulations, des estimations, des synthèses et des algorithmes ont été mesurés sur un poste de travail Windows XP Professionnel avec un processeur Intel Core 2 Duo E6650 de 2.33 GHz, un disque dur de 7200 RPM et 2 GB de RAM.

9.1 Présentation des études de cas

9.1.1 Système de guidage d'un rover

La spécification pour SPACE du système de guidage d'un rover a été présentée pour la première fois dans (Filion *et al.*, 2007). La mission du rover est de suivre une piste sur le sol. Le rover est équipé de deux moteurs installés sur chacune des deux roues avant ainsi que d'une caméra à l'avant qui prend périodiquement des images du terrain devant le rover. Le système de guidage traite les images prises par cette caméra et contrôle la direction et la vitesse du rover en ajustant la vitesse des moteurs gauche et droit.

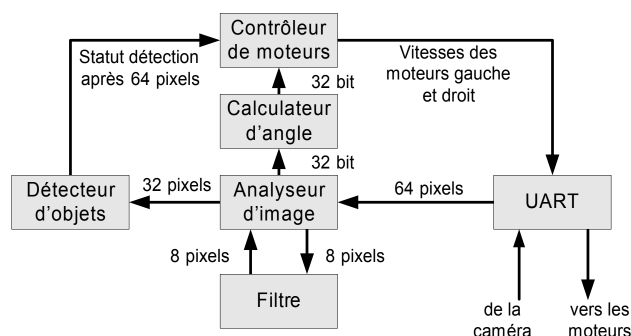

Figure 9.1 Schéma de la spécification exécutable du système de guidage du rover

Tel qu'illustré à la figure 9.1, le système de guidage est composé de cinq modules et d'un UART pour les communications avec la caméra et les moteurs selon le protocole RS232 (EIA, 1969). Ces cinq modules sont l'analyseur d'image, le filtre, le calculateur d'angle, le contrôleur des moteurs et le détecteur d'objets. L'analyseur d'image lit d'abord une image de 8x8 pixels de la caméra et l'envoie ensuite rangée par rangée au travers du filtre afin d'enlever le bruit de l'image. L'analyseur d'image traite ensuite l'image filtrée afin de déterminer la nouvelle direction que doit prendre le rover pour continuer à suivre la piste. Cette valeur est alors envoyée au calculateur d'angle qui transforme cette direction en un angle. Le contrôleur des moteurs reçoit cet angle et, à partir de la vitesse et de la direction actuelles du rover, ajuste les vitesses des moteurs gauche et droit. Pendant ce temps, le détecteur d'objet traite l'image et cherche des obstacles se trouvant à proximité de la piste. Le contrôleur des moteurs est informé lorsqu'un objet est détecté et il ralentit la vitesse du rover lorsque ces objets se rapprochent.

Afin de permettre la simulation du système de guidage sans disposer d'un rover physique se déplaçant dans un environnement réel, une application Windows simule les caméras et les moteurs du rover dans un environnement virtuel. L'analyseur d'image et le contrôleur de moteurs communiquent alors avec cette application Windows via un UART sur un port RS232 comme s'il s'agissait d'un rover réel. Le rover se déplace ainsi dans cet environnement virtuel selon les commandes qui lui sont envoyées par le système de guidage.

9.1.2 Décodeur JPEG avec détection de la peau

La spécification d'un décodeur JPEG avec SPACE a d'abord été présentée dans (Chevalier *et al.*, 2006) et l'ajout de la détection de la peau a été présenté dans (Migliorini, 2008). Tel qu'illustré à la figure 9.2, le décodeur JPEG avec détection de la peau est composé de six modules et trois mémoires partagées. Les six modules sont l'extracteur, le décodeur Huffman, la transformée inverse en cosinus discrète (IDCT), le quantificateur inverse, le convertisseur d'espace colorimétrique YCbCr vers RGB (Y2R) et le détecteur de peau. Les trois mémoires partagées sont la mémoire JPEG, la mémoire RGB et la mémoire en tons de gris.

Le décodeur JPEG avec détection de la peau procède en deux étapes principales pour chaque image JPEG. D'abord, l'image JPEG est décodée afin de restituer une image décompressée dans l'espace colorimétrique RGB (rouge vert bleu), qui est sauvegardée dans la mémoire RGB de manière similaire au format d'image BMP (Miano, 1999) avec 24 bits par pixel (8 bits par couleur). La détection de la peau s'effectue ensuite sur cette image RGB pixel par pixel et on obtient une image en tons de gris dans laquelle les pixels représentent la luminance de la peau dans l'image originale. Les pixels qui ne correspondent pas à de la peau deviennent blancs. Cette image en tons de gris est sauvegardée dans la mémoire en tons de

gris dans un format similaire au format BMP à 8 bits par pixel. Cette image de peau pourrait être ensuite utilisée pour la détection des visages (Peer et Solina, 1999). L'annexe F présente cette application plus en détails.

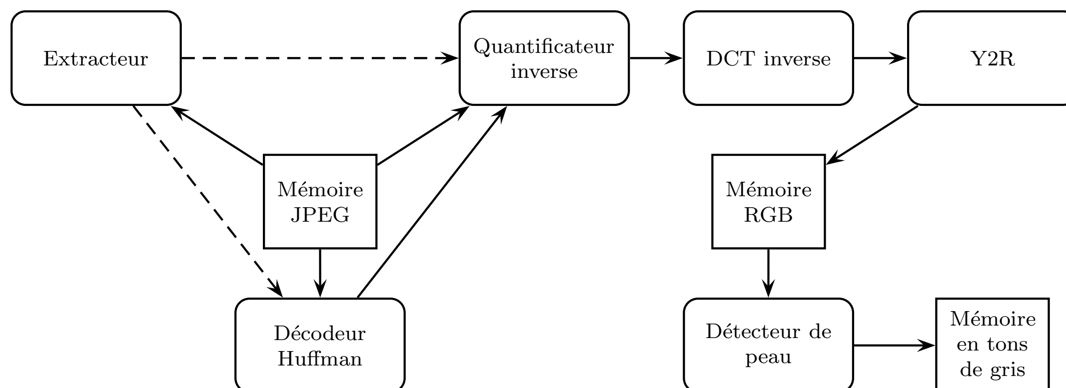

Figure 9.2 Schéma des modules et périphériques du décodeur JPEG

9.1.3 Encodeur/décodeur WIMAX

La spécification d'un codeur/décodeur pour la couche physique du standard de communications sans fil WiMAX a d'abord été présentée dans (Bah, 2010). Tel qu'illustré à la figure 9.3, cette application comprend les chaînes de codage et de décodage pour le WiMAX. Ainsi, un paquet de données peut être transmis sur un canal de communications en encodant ce paquet selon la chaîne de codage du WiMAX, aussi connu sous le nom de standard IEEE 802.16-2004 (IEEE, 2004). Cette chaîne de codage peut être paramétrée selon différents taux de transferts, types de modulation et nombre de sous-canaux. Un module paramétreur se charge de propager les paramètres pertinents aux différents module de la chaîne de codage. Un paquet peut également être reçu à partir d'un canal de communications en le décodant avec la chaîne de décodage WiMAX.

Pour simplifier la simulation de la spécification exécutable, on ne modélise pas la conversion analogique du signal et on relie directement entre elles les chaînes de codage et de décodage : le signal transmis par la chaîne de codage devient le signal reçu par la chaîne de décodage. Cela permet à la chaîne de codage de servir de banc d'essai pour la chaîne de décodage et vice versa. Bien sûr, dans une implémentation finale, le signal transmis par l'encodeur WiMAX serait reçu et décodé par un récepteur WiMAX distant et le signal reçu par le décodeur WiMAX aurait également été transmis par un transmetteur WiMAX distant. Cette application est présentée plus en détails à l'annexe F.

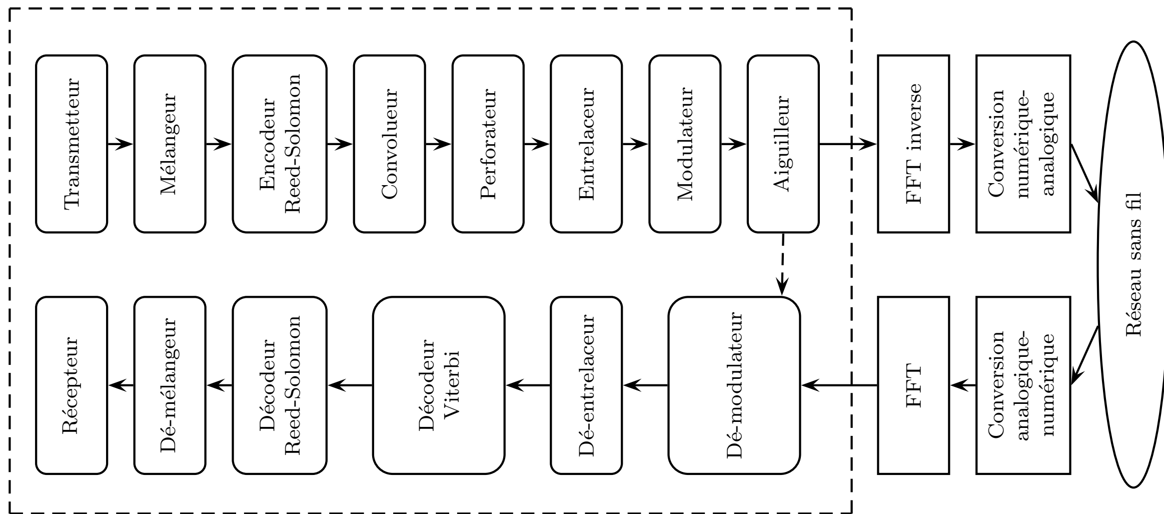

Figure 9.3 Schéma des modules de l'encodeur/décodeur WiMAX

9.2 Modéles RTPN

Les trois applications utilisées comme étude de cas respectent le modèle de calcul des RTPN présenté au chapitre 4. Ainsi, l'encodeur/décodeur WiMAX et le décodeur JPEG avec détection de la peau sont modélisés directement comme des réseaux de processus Kahn (KPN) : on fait correspondre un processus à chaque module de l'application et un canal Kahn à chaque dépendance de données entre les modules selon la modélisation des communications SPACE présentées à la section 4.3.5. Les mémoires partagées du décodeur JPEG avec détection de la peau sont chacune modélisées avec un ensemble de processus avec un canal de lecture 1-REG et un canal d'écriture 1-REG, tel que décrit à la section 4.3.1. Le décodeur JPEG avec détection de la peau a été codé de telle manière que l'accès aux mémoires partagées respecte la sémantique des KPN : il est donc équivalent à un KPN et, pour accélérer les simulations, on ne vérifie pas les temps d'écriture et de lecture à chaque case mémoire pour déterminer s'ils respectent bel et bien la sémantique des KPN.

Le système de guidage du rover est quant à lui modélisé selon le RTPN illustré à la figure 9.4. Tel que décrit à la section 4.3.3, on associe un canal 16-BREG à l'entrée qui provient de l'UART et un canal 16-REG à la sortie envoyée à l'UART avec une contrainte temps-réel. Le système de guidage doit mettre à jour la vitesse des moteurs au moins une fois par seconde pour que le rover se maintienne sur la piste : l'environnement lit donc la sortie envoyée aux moteurs via l'UART à chaque seconde. La caméra envoie également à chaque seconde une image au système de guidage pour analyse. Étant donné que ces images sont envoyées à l'UART à une vitesse de 33600 bauds (ou 33600 symboles par seconde) et que 10 symboles sont envoyés pour chaque octet (1 bit de départ, 8 bits de données et 1 bit d'arrêt), le canal d'entrée

16-BREG de l'UART reçoit un message d'un octet à chaque 0.298 millisecondes pendant la transmission d'une image. Lors de la simulation du rover, ces contraintes imposées par l'environnement sont utilisées pour extraire, selon les procédures décrites aux sections 4.2.2 et 7.2, les séquences de bits caractérisant l'ordonnancement des accès aux canaux 16-(B)REG. Cela permet d'évaluer à quel point une implémentation du rover respecte ces contraintes : si ces séquences de bits sont toutes à 1, alors cette implémentation respecte toutes les contraintes (aucun octet n'est écrasé dans l'UART et les moteurs reçoivent toutes les mises à jour à temps). Dans ce cas-ci, la spécification exécutable du rover est telle que toutes ses séquences de bits contiennent exclusivement des 1 et la métrique V pour les différentes architectures du rover est donc calculée selon le nombre de 0 que contiennent les séquences de bits de cette architecture.

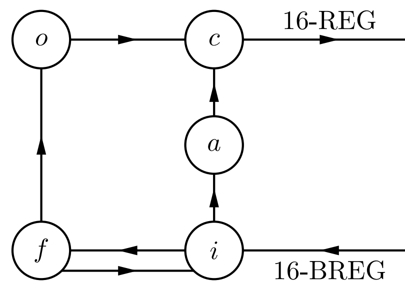

Figure 9.4 Modèle RTPN du rover avec l'analyseur d'image (i), le filtre (f), le calculateur d'angle (a), le contrôleur des moteurs (c) et le détecteur d'objets (o)

9.3 Synthèse matérielle et des communications

La méthode automatisée de raffinement des communications et de synthèse matérielle présentée au chapitre 5 a été appliquée aux trois études de cas. La spécification exécutable de chacune de ces applications constitue son modèle TLM pour les fins de cette méthode. Les communications des modules de l'application ont d'abord été raffinées pour utiliser un bus OPB. Finalement, les modules matériels de chaque application ont été synthétisés en une implémentation matérielle ciblant un FPGA Virtex-II XC2V2000-FF896 de Xilinx (Xilinx Inc., 2002).

9.3.1 Synthèse des modules matériels

Cette section présente des résultats sur les implémentations RTL des modules matériels produites par la méthode automatisée de raffinement des communications et de synthèse matérielle pour les différentes études du cas. Ainsi, la quantité de ressources matérielles

utilisées par chaque implémentation RTL d'un module matériel est évaluée en réalisant la synthèse logique de celle-ci à l'aide du logiciel Embedded Development Kit (EDK) fourni par Xilinx. Étant donné que l'architecture cible est un FPGA de Xilinx, les ressources matérielles sont exprimées en termes de *slices* du FPGA. (Dans un FPGA Xilinx, une *slice* comprend deux LUT et deux bascules.) Les autres ressources matérielles pouvant être utilisées par les modules RTL sont des mémoires BRAM et des multiplieurs.

Pour le décodeur JPEG avec détection de la peau, des implémentations RTL ont été générées pour chacun des modules de l'application, soit la conversion YUV2RGB, la DCT inverse, la quantification inverse, le décodeur Huffman, la détection de visage et l'extracteur. Le tableau 9.1 présente les ressources matérielles consommées par l'implémentation RTL générée pour chaque module du décodeur JPEG.

Tableau 9.1 Ressources matérielles utilisées par les modules du décodeur JPEG

Module	Slices	Mémoires BRAM	Multiplieurs
Conversion YUV2RGB	743	0	4
DCT inverse	8659	0	66
Quantification inverse	870	1	1
Décodeur Huffman	2548	3	0
Détection de visage	463	0	0
Extracteur	888	0	0

Pour le système de guidage du rover, des implémentations RTL ont été générées pour trois modules matériels, soit le calculateur d'angle, le filtre et le détecteur d'objet. La troisième colonne du tableau 9.2 présente les ressources matérielles consommées par l'implémentation RTL générée pour chaque module du rover. Aucun de ces modules du rover n'utilise de mémoire BRAM ou de multiplieur. Pour chacun des modules matériels du rover, une seconde implémentation RTL a été codée à la main par un concepteur matériel expérimenté qui a appliqué un flot manuel équivalent au flot automatisé présenté dans le chapitre 5 (Moss *et al.*, 2008). Ainsi, cette implémentation RTL des modules matériels utilise le même protocole de communications que celle générée par le flot automatisé et ces deux versions sont donc interchangeables. La synthèse logique de ces implémentations RTL codées à la main a également été réalisée à l'aide du logiciel EDK de Xilinx et la deuxième colonne du tableau 9.2 présente les ressources matérielles consommées pour chacun des modules du rover. Bien qu'un des modules générés (le calculateur d'angle) nécessite approximativement autant de ressources matérielles que sa version codée à la main, les deux autres modules générés (le filtre et le détecteur d'objet) utilisent significativement plus de ressources matérielles (de 50% à 136%) que leurs versions codées à la main. La latence des deux versions RTL de chaque module

du rover a été mesurée et les résultats montrent que les modules matériels générés ont la même latence que les modules codés à la main (Moss *et al.*, 2008). Ainsi, dans les deux cas, le calculateur d'angle a une latence de 14 cycles (incluant les communications), le filtre a une latence de 36 cycles et le détecteur d'objet a une latence de 27 cycles.

Tableau 9.2 Ressources matérielles utilisées par les modules du rover

	Slices		
Module RTL	Codé à la main	Généré	Ratio
Calculateur d'angle	33	35	1.06
Filtre	40	60	1.50
Détecteur d'objet	22	52	2.36

Pour le codec (encodeur/décodeur) WiMAX, des implémentations RTL ont été générées pour les modules suivants : le mélangeur, l'encodeur Reed-Solomon, le décodeur Reed-Solomon, le convolueur l'entrelaceur, le dé-entrelaceur et le décodeur Viterbi. Ainsi, la colonne de droite du tableau 9.3 présente les ressources matérielles consommées par l'implémentation RTL générée pour ces modules du codec WiMAX. Une seconde implémentation RTL sert également de point de comparaison pour ces modules matériels. Ainsi, pour l'encodeur Reed-Solomon, le décodeur Reed-Solomon, l'entrelaceur, le dé-entrelaceur et le décodeur Viterbi, cette seconde implémentation est fournie avec l'outil CORE Generator de Xilinx (Xilinx Inc., 2003) et on y a ajouté un adaptateur VHDL pour la gestion du protocole de communications de SPACE. Pour le mélangeur et le convolueur, cette seconde implémentation RTL a été codée à la main (Bah, 2010). La colonne de gauche du tableau 9.3 présente les ressources matérielles consommées pour cette seconde implémentation RTL des modules du codec WiMAX après une synthèse logique réalisée avec EDK. (Une implémentation RTL du dé-mélangeur, du perforateur, de l'aiguilleur, du modulateur et du dé-modulateur ont également été générées, mais ces modules ne sont pas inclus dans le tableau car ils n'ont pas une deuxième implémentation directement comparable.)

Dans certains cas, la quantité de *slices* utilisée par la version générée par le flot automatisé est sensiblement équivalente à celle utilisée par la version codée à la main (inférieure dans le cas de la convolution, une augmentation de 13% pour l'encodeur Reed-Solomon et de 14% pour le décodeur Viterbi). Pour d'autres modules, la version générée par le flot automatisé demande considérablement plus de *slices*, allant d'une augmentation de 42% pour le décodeur Reed-Solomon à un facteur de 6 pour l'entrelaceur.

Cette différence dans les quantités de ressources matérielles pour les modules du système de guidage du rover et du WiMAX s'explique par le fait que les versions codées à la main

Tableau 9.3 Ressources matérielles utilisées par les modules du codec WiMAX

Module RTL	Slices		Ratio
	Codé à la main	Généré	
Mélangeur	69	110	1.59
Encodeur Reed-Solomon	956	1085	1.13
Décodeur Reed Solomon	4306	6104	1.42
Convolueur	186	126	0.68
Entrelaceur	144	859	5.97
Dé-entrelaceur	130	374	2.88
Décodeur Viterbi	1080	1235	1.14

sont optimisées pour les caractéristiques spécifiques au module tandis que la génération automatique se base sur des opérations génériques de raffinement. En particulier, la séparation des communications et des calculs dans une méthodologie de haut niveau et l'utilisation d'un flot automatisé rendent possiblement plus difficile la génération d'une implémentation optimale pour les modules (tels que le filtre et le détecteur d'objet) qui transfèrent et traitent des structures de données de taille supérieure à la largeur du bus. Des tests subséquents ont montré qu'un couplage plus étroit des transacteurs et des calculs peut donner une implémentation nécessitant moins de ressources matérielles. Ainsi, si l'algorithme de détection d'objet du module détecteur d'objet est inséré directement dans la fonction de désérialisation de son transacteur, alors la synthèse comportementale génère une implémentation qui utilise 29 *slices*, comparativement à 22 *slices* pour l'implémentation manuelle et 52 *slices* pour l'implémentation générée sans cette modification.

Ces résultats selon lesquels la version RTL d'un module générée à l'aide d'un outil de synthèse comportementale demande plus de ressources matérielles qu'une version RTL codée à la main corrobore les résultats de travaux antérieurs portant sur les outils de synthèse comportementale. En effet, selon (Sarkar *et al.*, 2009) qui compare trois outils de synthèse comportementale à un processus de codage manuel, le codage manuel donne une implémentation RTL initiale qui consomme relativement peu de ressources matérielles, mais beaucoup de temps doit être consacré au débogage de la fonctionnalité de cette implémentation RTL. Inversement, la synthèse comportementale produit une implémentation RTL initiale qui a une fonctionnalité correcte, mais il faut ensuite modifier le code à haut niveau et itérer le processus de synthèse comportementale afin d'optimiser la quantité de ressources matérielles requises. Dans (Sarkar *et al.*, 2009), une implémentation RTL initiale générée par un outil de synthèse comportementale demande de 2 à 2.5 fois plus de ressources matérielles qu'une implémentation RTL codée à la main. Après l'optimisation du code pour la synthèse compor-

tementale, celle-ci demande de 10% à 25% plus de ressources matérielles que l'implémentation RTL codée à la main.

9.3.2 Simulation à différents niveaux

Le système de guidage du rover, le décodeur JPEG avec détection de la peau et le codec WiMAX ont été simulés à trois niveaux différents : au niveau de la spécification exécutable (niveau TLM), après le raffinement des communications vers un bus OPB (niveau OPB) et après la synthèse des modules matériels (niveau RTL) mais avant la synthèse de la plate-forme. On a ainsi vérifié que le comportement de chacune des applications demeurait conforme à la spécification après chaque étape de raffinement et de synthèse. On distingue ici le temps pris par une simulation, soit son WCT (Wall Clock Time), du temps simulé à l'intérieur de la simulation, soit son temps SystemC. Les WCT et les temps SystemC des simulations des différents niveaux sont comparés pour chacune des applications.

Les tableaux 9.4, 9.5 et 9.6 présentent ces résultats respectivement pour l'étude de cas du rover, du décodeur JPEG et du codec WiMAX. Au niveau RTL, on utilise pour chacun des modules matériels présentés à la section 9.3.1 l'implémentation RTL SystemC générée par le flot automatisé de raffinement des communications et de synthèse matérielle. Les modules qui n'ont pas d'implémentation RTL demeurent à un niveau comportemental.

Tableau 9.4 Simulation à différents niveaux du système de guidage du rover

Niveau	WCT (s)	Temps SystemC (s)
TLM	1.8	0.0617
OPB	3.5	0.0617
RTL	10.6	0.0627

Tableau 9.5 Simulation à différents niveaux du décodeur JPEG

Niveau	WCT (s)	Temps SystemC (s)
TLM	0.4	0.0033
OPB	3.3	0.0329
RTL	314.4	0.0609

On observe que le WCT de la simulation augmente à mesure que le niveau de raffinement augmente : le niveau RTL prend plus de temps à simuler que le niveau OPB, qui prend lui-même plus de temps que le niveau TLM. La différence est particulièrement prononcée entre le niveau OPB et le niveau RTL : le niveau OPB est 3 fois plus rapide que le niveau RTL dans le

Tableau 9.6 Simulation à différents niveaux de l'encodeur/décodeur WiMAX

Niveau	WCT (s)	Temps SystemC (s)
TLM	0.3	0.00011
OPB	0.4	0.00089
RTL	63.6	0.01061

cas du rover, 95 fois plus rapide pour le décodeur JPEG et 159 fois plus rapide pour le codec WiMAX. Cette augmentation de la vitesse de simulation confirme un des arguments en faveur des méthodologies de conception à plus haut niveau. La relativement faible augmentation de la vitesse dans le cas du rover est dû au fait que seuls 3 des 5 modules du rover ont une implémentation RTL. Par contre, la simulation au niveau RTL a son utilité étant donné que l'inclusion des implémentations RTL des modules matériels permet de modéliser plus précisément la performance du système comme le montrent les différences entre les temps SystemC (internes à la simulation) obtenus au niveau RTL et à plus haut niveau. Les faibles variations observées pour le rover s'expliquent par le fait que c'est l'attente de la réception des données par l'UART qui domine la performance du système dans ce cas-ci. Le chapitre 7 a présenté une méthode d'estimation qui vise à combiner la vitesse de la simulation à haut niveau avec la précision de la simulation au niveau RTL.

Des architectures logicielles/matérielles du décodeur JPEG ont également été simulées à la fois au niveau OPB, où tous les modules matériels sont à un niveau comportemental, et au niveau RTL, où les modules matériels ont leur implémentation RTL. Pour ces architectures, le niveau OPB est de 8.5 à 21 fois plus rapide que le niveau RTL alors que l'erreur sur le temps SystemC simulé au niveau OPB varie entre 0.7% et 8.5% par rapport au niveau RTL. L'ajout d'un ou plusieurs simulateurs de jeu d'instructions (ISS) ralentit considérablement la simulation aux niveaux OPB et RTL : le WCT varie entre 93 et 351 secondes au niveau OPB et entre 1626 et 3887 secondes au niveau RTL (Bois *et al.*, 2010). Par contre, étant donné que la performance logicielle domine alors la performance du système au complet, le niveau OPB, qui simule ce logiciel, demeure dans ce cas relativement précis comparativement au niveau RTL.

9.3.3 Synthèse sur FPGA

Le rover a été implémenté sur une carte multimédia de Xilinx qui comprend un FPGA Virtex-II XC2V2000-FF896 (Xilinx Inc., 2002). Le système de guidage du rover a été implémenté dans le FPGA avec une horloge de 50 MHz. Le port RS232 a été configuré à 33600 bauds et a été utilisé pour les communications entre le système de guidage et l'application

Windows qui simule les caméras et les moteurs du rover dans un environnement virtuel.

L'architecture cible testée comprend deux bus OPB (IBM Corp., 1999) et deux processeurs MicroBlaze (Xilinx Inc., 2005). Comme le montre la figure 9.5, l'architecture cible testée dans cette section est composée de trois modules matériels (le calculateur d'angle, le filtre et le détecteur d'objet) alors que les deux autres modules (le contrôleur et l'analyseur d'image) sont raffinés en des modules logiciels qui s'exécutent séparément sur les deux processeurs MicroBlaze. Pour éviter d'encombrer la figure, on ne montre ni les adaptateurs de bus OPB utilisés par les modules matériels, ni les minuteries et les mémoires utilisées par les processeurs.

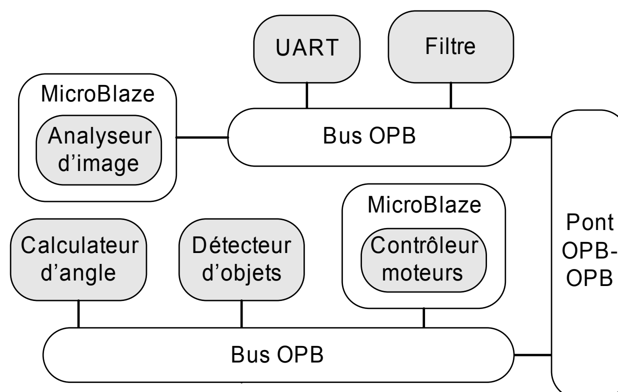

Figure 9.5 L'architecture cible pour le système de guidage

Deux versions de l'architecture cible ont été synthétisées et exécutées sur le FPGA cible : une première utilisant les modules matériels générés par notre méthode automatisée et une seconde utilisant les modules VHDL codés à la main, tel que décrit à la section 9.3.1. Les deux versions utilisent la même architecture de communication et le même logiciel embarqué. Dans les deux cas, le système de guidage a réussi à garder le rover sur la piste tout en respectant les contraintes de ressources matérielles et de fréquence du FPGA.

Comme le montre le tableau 9.7, l'architecture qui utilise les modules générés automatiquement a la même latence par image et utilise une quantité légèrement supérieure (par moins de 1%) de ressources matérielles comparativement à l'architecture qui utilise les modules codés à la main. Même si les modules matériels générés avaient une latence plus élevée que ceux codés à la main, la différence dans la latence du système aurait été petite étant donné que la plus grande partie de la latence de l'application vient de l'exécution du logiciel, ce qui est le cas de la plupart des systèmes embarqués. Aussi, la plupart des ressources matérielles utilisées par le système de guidage sont utilisées par les processeurs, les bus et les périphériques du système et non pas par les modules matériels. Dans de telles circonstances, le gain de productivité obtenu en automatisant le raffinement des communications et la synthèse

matérielle peut plus que compenser l'augmentation de la quantité de ressources matérielles. Étant donné qu'il est plus facile de faire des modifications dans le code au niveau TLM qu'au niveau RTL, un tel raffinement automatisé peut être utilisé pour un prototypage rapide et itératif du système. Cependant, il peut être nécessaire de faire certains raffinements manuels pour la version finale d'une application à très forte teneur matérielle et avec des contraintes serrées en termes de ressources matérielles.

Tableau 9.7 Ressources matérielles et latence pour l'architecture de la figure 9.5

Métrique	Matériel codé à la main	Matériel généré
Latence	$1,736 * 10^3$ cycles	$1,736 * 10^3$ cycles
Ressources matérielles	6,396 slices	6,456 slices

9.4 Profilage au niveau système

On présente dans cette section des résultats sur l'impact du profilage au niveau système sur la performance des simulations dans la plate-forme virtuelle SPACE.

9.4.1 Architectures testées

Les résultats sur le profilage au niveau système sont présentés pour trois architectures différentes pour chacune des études de cas. Ainsi les deux premières architectures testées pour le système de guidage du rover sont les architectures Rover-1MB-a (R_{1a}) et Rover-1MB-b (R_{1b}) qui contiennent chacune un bus OPB et un processeur MicroBlaze. Dans Rover-1MB-a, les modules de l'analyseur d'image, du calculateur d'angle et du contrôleur des moteurs sont implémentés en logiciel sur le processeur MicroBlaze alors que les modules du filtre et du détecteur d'objets sont implémentés en matériel sur le bus OPB. L'architecture Rover-1MB-b diffère de Rover-1MB-a par son partitionnement logiciel/matériel : le détecteur d'objets y est plutôt implémenté en logiciel. Finalement, la troisième architecture Rover-2MB (R_2) est illustrée à la figure 9.5.

Les trois architectures du décodeur JPEG avec détecteur de peau contiennent chacune deux bus. Les modules Y2R et détecteur de peau ainsi que la mémoire en tons de gris se trouvent sur le premier bus alors que les modules extracteur, décodeur Huffman, IDCT et quantificateur inverse ainsi que la mémoire JPEG se trouvent sur le deuxième bus. La mémoire RGB est quant à elle connectée aux deux bus. Les trois architectures diffèrent entre elles par leur partitionnement logiciel/matériel. Ainsi, dans la première architecture JPEG-HW

(J_0), tous les modules sont implémentés en matériel. Dans la deuxième architecture JPEG-1MB-a (J_{1a}), il y a un processeur sur lequel les modules extracteur et décodeur Huffman sont implémentés en logiciel alors que les autres sont implémentés en matériel. La troisième architecture JPEG-2MB-a (J_{2a}) contient 2 processeurs : le premier processeur est identique à celui de JPEG-1MB alors que le deuxième processeur exécute les tâches logicielles correspondant aux modules Y2R et détecteur de peau.

Les architectures du WiMAX suivent le même principe que celles du JPEG. Ainsi, les trois architectures ont chacune deux bus. On trouve sur le premier bus les modules de la chaîne de codage WiMAX (transmetteur, paramétreur, mélangeur, encodeur Reed-Solomon, convolueur, perforateur, entrelaceur, modulateur et aiguilleur) et sur le deuxième bus les modules de la chaîne de décodage WiMAX (dé-modulateur, dé-entrelaceur, décodeur Viterbi, décodeur Reed-Solomon, dé-mélangeur, récepteur). Dans la première architecture WiMAX-HW (W_0), tous les modules sont implémentés en matériel. Pour obtenir la deuxième architecture WiMAX-1MB (W_1), on ajoute un premier processeur et on y transfère en tant que tâches logicielles les modules perforateur et entrelaceur. La troisième architecture WiMAX-2MB (W_2) s'obtient en ajoutant un deuxième processeur et en y assignant les modules dé-mélangeur et récepteur comme tâches logicielles.

9.4.2 Impact du profilage

Chacune des architectures présentées à la section 9.4.1 a été simulée avec et sans le profilage au niveau système présenté au chapitre 6. Cela permet de mesurer quel est le temps de simulation additionnel qui est attribuable à l'instrumentation de la plate-forme virtuelle et à l'écriture des fichiers d'enregistrements du profilage. De plus, ces architectures ont été profilées au niveau système avec deux implémentations différentes du profilage non-intrusif du code logiciel : d'abord avec le profileur d'ISS MicroBlaze présenté à la section 6.1.2.1 et ensuite avec le profileur d'ISS GDB présenté à la section 6.1.2.2. On distingue encore ici le temps pris par une simulation (son WCT) du temps simulé à l'intérieur de la simulation (son temps SystemC). Ainsi, les tableaux 9.8, 9.9 et 9.10 présentent le WCT et le temps SystemC pour les simulations des architectures respectives du système de guidage du rover, du décodeur JPEG avec détection du peau et de l'encodeur/décodeur WiMAX. Les trois architectures du système de guidage du rover ont été simulées avec le même banc d'essai qui consiste en un ensemble d'images prises par la caméra du rover sur une distance correspondant à 1/16 de tour de piste. L'architecture JPEG-HW du décodeur JPEG avec détection de peau a été simulée avec un banc d'essai composé de 8 images de 128×128 pixels au format JPEG alors que les architectures JPEG-1MB-a et JPEG-2MB-a ont été simulées avec un banc d'essai de 1/8 d'une image JPEG de 128×128 pixels. Les trois architectures du codec WiMAX ont été

simulées pour trois trames WiMAX.

Tableau 9.8 Simulation avec et sans profilage des architectures du rover

	WCT (s)			
Architecture	Sans profilage	Prof. MicroBlaze	Prof. GDB	Temps SystemC (s)
Rover-1MB-a	34.2	36.0	527	0.60400
Rover-1MB-b	38.4	39.6	534	0.66698
Rover-2MB	58.6	59.4	520	0.57332

Tableau 9.9 Simulation avec et sans profilage des architectures du décodeur JPEG

	WCT (s)			
Architecture	Sans profilage	Prof. MicroBlaze	Prof. GDB	Temps SystemC (s)
JPEG-HW	14.9	20.3	20.3	0.11198
JPEG-1MB-a	16.2	17.0	270	0.14721
JPEG-2MB-a	114.2	115.4	2059	0.57764

Tableau 9.10 Simulation avec et sans profilage des architectures du codec WiMAX

	WCT (s)			
Architecture	Sans profilage	Prof. MicroBlaze	Prof. GDB	Temps SystemC (s)
WiMAX-HW	1.6	1.7	1.7	0.00336
WiMAX-1MB	6.0	6.3	70.8	0.09857
WiMAX-2MB	11.5	11.8	97.1	0.11200

On tire plusieurs constats de ces résultats. D'abord, l'ajout d'un processeur à une architecture (et donc l'ajout d'un ISS à la simulation) ralentit considérablement la simulation. Ainsi, la simulation sans profilage d'une architecture avec un ISS est 3.8 fois plus lente que la simulation d'une architecture sans ISS dans le cas de l'encodeur/décodeur WiMAX et 69.6 fois plus lente dans le cas du décodeur JPEG avec détection de peau (en tenant compte du fait que le stimuli fourni à JPEG-HW est 64 fois plus long que celui fourni à JPEG-1MB-a). Cela corrobore les résultats présentés dans (Chevalier *et al.*, 2006; Bois *et al.*, 2010).

Nos tests confirment que le temps total de la simulation, dans le référentiel du temps interne à la simulation, n'est pas modifié par le profilage au niveau système. Cela signifie que le profilage au niveau système ne perturbe pas la performance des simulations dans la plate-

forme virtuelle. Par contre, le profilage au niveau système augmente le WCT de la simulation. Cela signifie que l'utilisateur doit attendre plus longtemps avant que la simulation se termine.

On observe ainsi que le profilage au niveau système avec profileur d'ISS MicroBlaze a un impact sur le WCT des simulations des architectures ayant au moins un processeur, mais que cet impact est faible. Ainsi, la valeur absolue de l'augmentation du WCT varie de 0.3 à 1.8 secondes pour ces architectures alors que sa valeur relative varie entre 1.05% et 5.26% du temps de simulation sans profilage. Par contre, l'utilisation du profileur d'ISS GDB ralentit grandement la vitesse des simulations de ces architectures. Ainsi, les simulations de ces architectures avec un profilage d'ISS GDB prennent de 8.4 à 18.0 plus de temps que les simulations sans profilage. Ce ralentissement important est causé par les communications interprocessus en texte ASCII entre le profileur d'ISS GDB et le débogueur croisé GDB ainsi qu'entre ce débogueur et le serveur GDB de l'ISS. Il a également été observé que le débogueur croisé GDB pour le MicroBlaze semble effectuer des communications superflues avec le serveur GDB. Ainsi, pour ajouter ou retirer un point d'arrêt, le débogueur GDB demande d'abord au serveur GDB de retirer chacun des points d'arrêt existants, puis d'ajouter de nouveau chacun des points d'arrêts (sauf le point d'arrêt réellement enlevé ou avec en plus le point d'arrêt réellement ajouté, selon le cas). En comparaison, le profileur d'ISS MicroBlaze manipule directement les structures de données C/C++ de l'ISS et n'a donc pas recours à des communications interprocessus en texte ASCII. De plus, ces manipulations s'effectuent de manière parcimonieuse. C'est ce qui explique la bonne performance du profilage au niveau système avec profilage d'ISS MicroBlaze

Pour les architectures n'ayant aucun processeur, aucune différence n'est observée selon l'implémentation du profileur d'ISS étant donné qu'il n'y a alors aucun ISS ou logiciel embarqué à profiler. On observe également que la valeur relative de l'augmentation du WCT due au profilage varie entre 6.25% et 36.24% pour ces architectures. Ces valeurs relatives sont plus élevées que celles observées pour le profilage au niveau système (avec profileur d'ISS MicroBlaze) des architectures ayant au moins un processeur. Cela reflète essentiellement le fait qu'une simulation SystemC sans ISS est intrinsèquement bien plus rapide qu'une simulation avec ISS et que le WCT causé par le profilage représente donc une plus grande fraction du temps total de simulation dans le premier cas. À l'opposé, la fraction du WCT causée par le profilage tend à diminuer quand passe d'un à deux processeurs.

Ces résultats démontrent qu'il est possible d'obtenir pour un coût minime un profilage non-intrusif au niveau système des simulations d'architectures logicielles/matérielles. Cela nécessite l'implémentation d'un profileur d'ISS dédié à l'ISS en question, tel que le profileur d'ISS MicroBlaze. Le profileur d'ISS GDB générique utilisant le débogueur GDB croisé et le serveur GDB ne permet pas d'obtenir des performances de simulation acceptables. La

principale utilité de ce profileur générique serait donc d'aider le développement des profileurs dédiés aux différents ISS. Une solution intermédiaire serait d'implémenter un profileur d'ISS qui communique directement et parcimonieusement avec le serveur GDB sans passer par le débogueur croisé GDB. Un tel profileur d'ISS serait spécifique à un processeur donné, mais aurait l'avantage de ne pas nécessiter l'accès au code source, aux structures de données ou aux fonctions de l'ISS.

9.5 Caractérisation et estimation

Les méthodes d'estimation basées sur la caractérisation présentée au chapitre 7 ont été évaluées selon deux critères : la précision et la vitesse. Ces méthodes d'estimation sont comparées aux méthodes de mesure exactes des métriques qu'elles cherchent à estimer, soit la synthèse logique pour la quantité de ressources matérielles et la simulation SystemC pour le temps d'exécution. On s'attend à ce qu'une méthode d'estimation permette d'obtenir une valeur approximative d'une métrique dans un temps bien plus court que celui nécessaire à la mesure d'une valeur exacte. Dans les résultats de cette section, on compare le temps (WCT) nécessaire à une synthèse logique ou à la simulation SystemC d'une architecture donnée avec celui nécessaire à l'estimation de ces métriques pour cette même architecture. Les temps présentés pour l'estimation n'incluent pas le temps nécessaire à la caractérisation initiale. En effet, ce temps de caractérisation initiale est indépendant du nombre d'architectures qui seront évaluées par la suite. Le coût marginal (sur l'évaluation d'une architecture supplémentaire) de cette caractérisation est donc nul et le coût initial peut être amorti sur l'évaluation de milliers d'architectures dans un algorithme d'exploration architecturale tel que ceux présentés au chapitre 8. De plus, le coût initial de la caractérisation de la plate-forme peut être amorti sur l'évaluation des architectures des différentes applications ciblant cette plate-forme. À l'opposé, les méthodes de mesure reprennent à zéro l'évaluation de chaque architecture.

9.5.1 Estimation de la quantité de ressources matérielles

Les architectures testées dans cette section sont les mêmes architectures que celles présentées à la section 9.4.1 pour le système de guidage du rover et pour l'encodeur/décodeur WiMAX. On réutilise également l'architecture J_0 du décodeur JPEG avec détection de peau ainsi que deux nouvelles architectures : J_{1b} et J_{2b}. L'architecture J_{1b} est similaire à J_{1a}, mais seul le module IDCT est assigné au seul processeur. L'architecture J_{2b} est obtenue à partir de l'architecture J_{1b} en assignant le module extracteur au premier processeur et en ajoutant sur le même bus un deuxième processeur sur lesquels s'exécutent les modules quantificateur inverse et décodeur Huffman.

La méthode d'estimation de la quantité de ressources matérielles présentée à la section 7.3.4 a été appliquée à chacune de ces architectures. Une synthèse logique de chacune de ces architectures a ensuite été réalisée à l'aide du logiciel EDK de Xilinx. Finalement, pour obtenir une mesure encore plus précise de la quantité des ressources matérielles, un placement de ces architectures sur le FPGA cible a été réalisé avec EDK. Toutes les architectures ont été évaluées avec une cible FPGA Xilinx Virtex 4 VFX60FF1152-11.

Les figures 9.6, 9.7, 9.8 et 9.9 comparent respectivement le nombre de bascules, de LUTs, de mémoires BRAM et de multiplieurs pour chaque architecture selon la méthode d'estimation puis selon les mesures prises suite à la synthèse logique et au placement avec EDK.

On observe que l'estimation a une grande précision relativement aux mesures prises suite à la synthèse logique. Ainsi, l'estimation prédit exactement le nombre de multiplieurs utilisés. L'erreur d'estimation sur la quantité de bascules est très faible : elle varie entre 0.52% et 2.04% et est en moyenne de 1.45%. L'erreur d'estimation sur la quantité de LUTs est faible : elle va de 1.45% à 6.29% et est en moyenne de 3.22%. Finalement, l'erreur d'estimation sur la quantité de mémoires BRAM est plus importante, mais tout de même modérée et acceptable : elle varie entre 0.00% et 14.29% et est en moyenne de 4.82%. L'erreur d'estimation sur la quantité de mémoires BRAMs reflète principalement l'erreur d'estimation sur la quantité de mémoire logicielle requise.

L'estimation a une précision moindre, mais néanmoins acceptable, par rapport aux mesures prises suite au placement. Le nombre de multiplieurs et de mémoires BRAM mesurés suite au placement est le même que suite à la synthèse logique. L'estimation prédit donc encore une fois exactement le nombre de multiplieurs utilisés et le nombre de mémoires BRAMs est estimé avec une erreur moyenne de 4.82%. Par contre, le nombre de bascules et de LUTs mesurés suite au placement est significativement inférieur aux nombres mesurés suite à la synthèse logique, ce qui cause des erreurs d'estimation plus importantes. Ainsi, l'erreur d'estimation sur la quantité de bascules varie entre 1.36% et 20.48% et est en moyenne de 11.72% alors que l'erreur d'estimation sur la quantité de LUTs varie entre 2.80% et 14.00% et est en moyenne de 8.93%.

La différence entre les quantités de ressources matérielles mesurées suite à la synthèse logique et suite au placement s'explique par les optimisations globales effectuées par EDK lors du placement. Prenons l'exemple d'un adaptateur de bus. La synthèse logique de cet adaptateur (notamment réalisée lors de la caractérisation) tiendra compte de toute la logique nécessaire à l'utilisation de l'ensemble des fonctionnalités du protocole du bus. Cependant, si le module de cet adaptateur n'exerce qu'une partie des fonctionnalités de ce protocole, alors seule une partie de la logique de cet adaptateur est exercée et le reste de la logique peut être éliminé. C'est cette optimisation qui est réalisée lors du placement. Les résultats

présentés aux tableaux 9.6 à 9.9 représentent donc la limite de la précision qu'il est possible d'atteindre avec une caractérisation basée sur la synthèse logique dans les outils de Xilinx. Une caractérisation plus précise devrait être basée sur le placement de différents ensembles de composants matériels et demanderait un temps bien plus long simplement dû au fait que le nombre d'ensemble de composants est bien plus élevé que le nombre de composants.

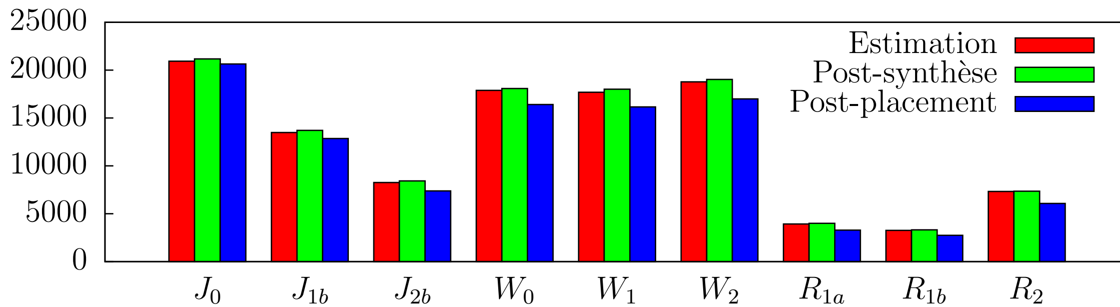

Figure 9.6 Comparaison du nombre estimé et mesuré de bascules

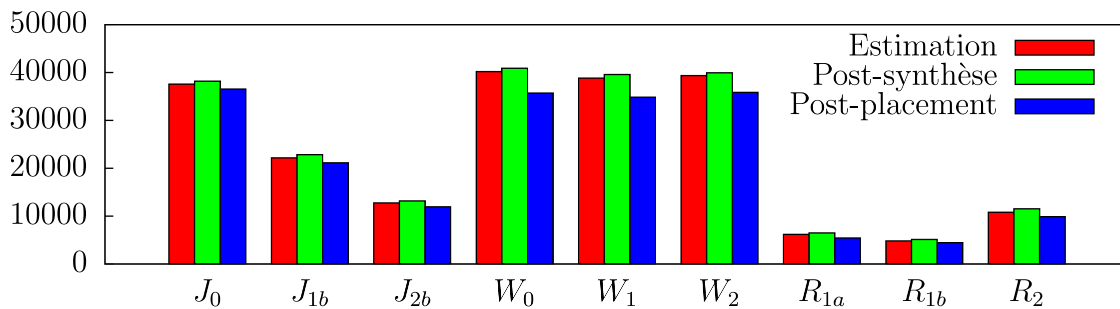

Figure 9.7 Comparaison du nombre estimé et mesuré de LUTs

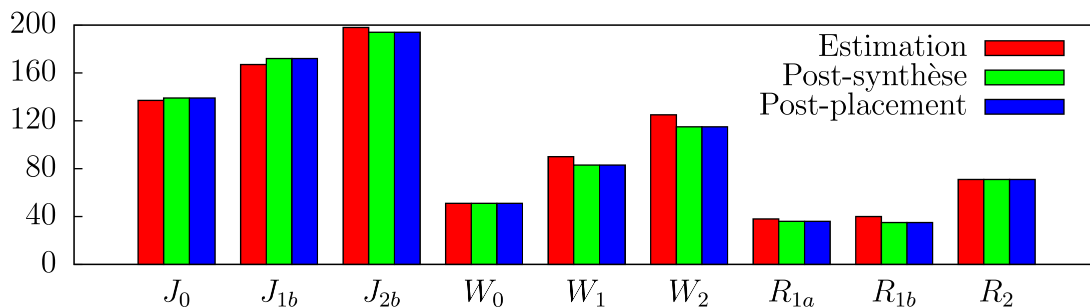

Figure 9.8 Comparaison du nombre estimé et mesuré de mémoires BRAMs

La figure 9.10 présente le WCT nécessaire à l'estimation, à la synthèse logique et au placement de chacune des architectures. Le temps nécessaire au placement inclut le temps pris par la synthèse logique préalable. Cela met en lumière un avantage indéniable de l'estimation, soit sa grande vitesse. Ainsi, le temps pris par l'estimation varie entre 0.53 et 17.48 millisecondes,

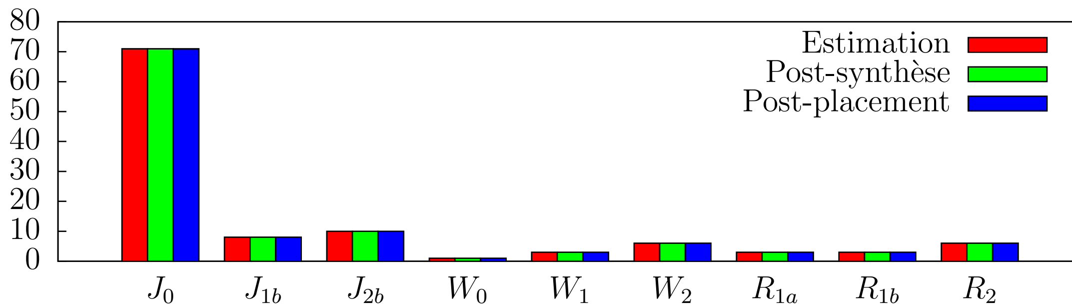

Figure 9.9 Comparaison du nombre estimé et mesuré de multiplieurs

pour une moyenne de 2.97 millisecondes. Le temps pris par la synthèse logique se situe quant à lui entre 360 et 3953 secondes (moyenne de 2146 secondes) alors que le temps pris par le placement va de 476 à 4110 secondes (moyenne de 2627 secondes). L'estimation est ainsi de 226105 à 3750009 fois plus rapide que la synthèse logique (moyenne de 1428109 fois plus rapide) et de 235085 à 3906049 fois plus rapide que le placement (moyenne de 1787808 fois plus rapide). Il va de soi que si le temps nécessaire à une synthèse logique ou un placement est de l'ordre d'une heure, alors il est peu pratique d'utiliser ces méthodes pour évaluer chacune des milliers d'architectures rencontrées par un algorithme d'exploration architecturale. Seule l'estimation peut fournir la vitesse d'évaluation nécessaire.

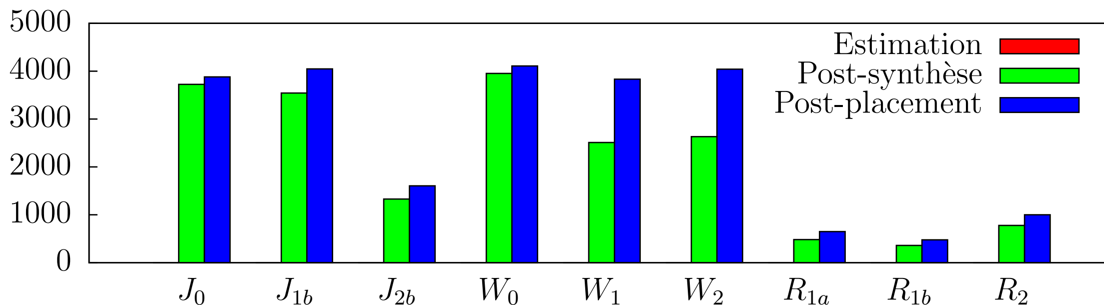

Figure 9.10 Comparaison du temps pris par l'estimation, la synthèse logique et le placement

9.5.2 Estimation du temps d'exécution

La méthode d'estimation du temps d'exécution présentée à la section 7.2.4 a été appliquée à différentes architectures du système de guidage du rover, du décodeur JPEG avec détection de peau et de l'encodeur/décodeur WiMAX afin de déterminer leur temps d'exécution. Un stimulus d'un tour de piste a été utilisé pour les architectures du rover, un stimulus de deux images JPEG de 128×128 pixels a été fourni aux architectures du décodeur JPEG et le stimulus des architectures du codec WiMAX est de trois trames WiMAX. Chacune de ces architectures a également été simulée au niveau Simtek de SPACE suite à une synthèse

du système. Cette simulation s'effectue donc avec une implémentation SystemC RTL pour les modules assignés en matériel, avec une implémentation logicielle exécutée instruction par instruction sur des ISS pour les modules assignés en logiciel, ainsi qu'avec des modèles SystemC BCA des bus, processus, mémoires et périphériques.

Les figures 9.11, 9.12 et 9.13 comparent, selon le référentiel du temps interne à la simulation, le temps d'exécution estimé et simulé pour les architectures respectives du système de guidage du rover, du décodeur JPEG avec détection de peau et de l'encodeur/décodeur WiMAX. Les architectures retenues ne contiennent pas d'interblocages : on évite ainsi que la simulation bloque prématurément au premier interblocage et que le temps d'exécution mesuré par la simulation soit alors faussé. Les 54 architectures évaluées pour le rover ont de 1 à 3 processeurs et de 1 à 4 bus. L'erreur d'estimation par rapport à la simulation va de 0.12% à 3.86% pour ces architectures et est en moyenne de 2.60%. Les 31 architectures du décodeur JPEG ont de 0 à 3 processeurs et de 1 à 3 bus et l'erreur d'estimation y varie entre 1.86% et 7.38%, pour une moyenne de 4.42%. Finalement, les 32 architectures du codec WiMAX ont de 0 à 2 processeurs et de 1 à 4 bus et l'erreur d'estimation y varie entre 0.07% et 4.13%, pour une moyenne de 1.46%. L'estimation a donc une bonne précision puisque l'erreur d'estimation est inférieure à 10% pour chacune des 119 architectures évaluées et que l'erreur d'estimation moyenne pour celles-ci est de 2.77%.

Le nombre de violations fonctionnelles des 54 architectures du système de guidage du rover ont également été estimées selon la méthode d'estimation présentée à la section 7.2.4. Ce nombre de violations fonctionnelles a également été mesuré suite à une simulation au niveau Simtek de SPACE de ces architectures. Des violations fonctionnelles se produisent dans ces architectures lorsque les séquences de bits caractérisant l'ordonnancement du RTPN du rover (présenté à la section 9.2) contiennent des 0 (pixels écrasés dans l'UART ou violation d'une contrainte temps-réel). Contrairement à un interblocage, une telle violation fonctionnelle ne bloque pas la simulation prématurément et les valeurs estimées peuvent alors être comparées aux valeurs mesurées. Ainsi, pour chacune des 10 premières architectures du système de guidage du rover, l'estimation détecte 110350 violations fonctionnelles (qui sont dues à une lecture trop lente des données dans le UART d'entrée). La simulation Simtek détecte quant à elle 110238 violations fonctionnelles pour chacune de ces architectures. L'erreur d'estimation est donc de seulement 0.10% pour ces architectures. Autant l'estimation et que la simulation Simtek considèrent que les 44 autres architectures du système de guidage du rover ne produisent aucune violation fonctionnelle. Étant donné que le décodeur JPEG avec détection de peau et l'encodeur/décodeur WiMAX sont tous deux essentiellement spécifiés comme des KPN et que tous leurs ordonnancements sont donc fonctionnellement équivalents, l'estimation et la simulation Simtek détectent exactement 0 erreurs pour chacune de leurs 63

architectures.

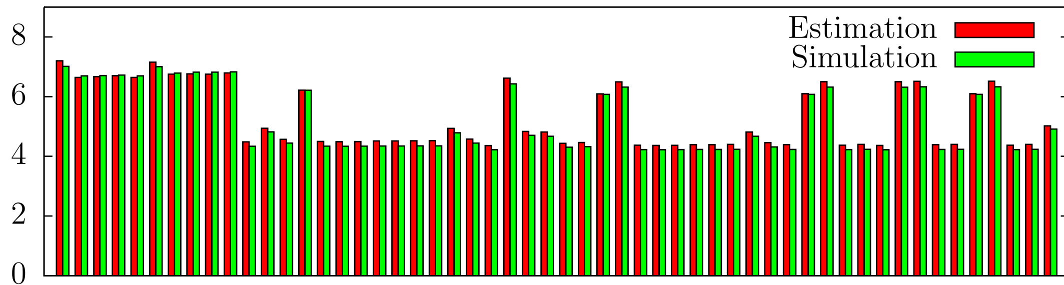

Figure 9.11 Comparaison du temps d'exécution estimé et simulé pour 54 architectures du système de guidage du rover

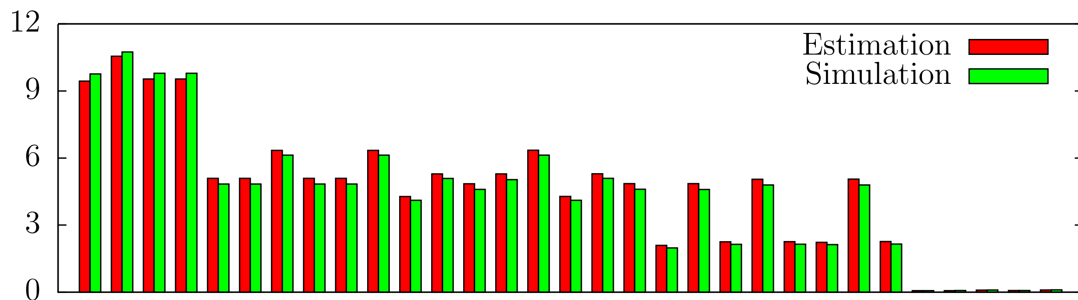

Figure 9.12 Comparaison du temps d'exécution estimé et simulé pour 31 architectures du décodeur JPEG avec détection de peau

Les tableaux 9.11, 9.12 et 9.13 comparent le temps nécessaire (WCT) à l'estimation et à la mesure par simulation Simtek des architectures respectives du système de guidage du rover, du décodeur JPEG avec détection de peau et de l'encodeur/décodeur WiMAX. Ces tableaux présentent le WCT moyen, minimum et maximum pour l'évaluation des architectures de chacune des études de cas. On observe ainsi que le temps pris par une estimation est de l'ordre des secondes alors que le temps pris par une simulation est de l'ordre des milliers de secondes. L'estimation permet donc d'accélérer grandement l'évaluation des architectures. Ces tableaux présentent également l'accélération moyenne, minimale et maximale qui a été obtenue en ayant recours à l'estimation plutôt qu'à la simulation pour les architectures de chacune des études de cas. Ainsi, cette accélération a été de 393 dans le pire cas et de 47358 dans le meilleur cas.

9.6 Exploration architecturale

Les algorithmes d'exploration architecturale présentés au chapitre 8 ont été appliqués aux études de cas du système de guidage du rover, du décodeur JPEG avec détection de peau

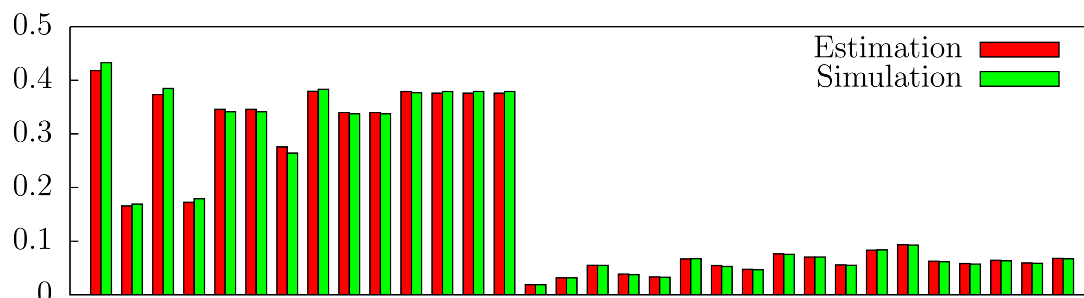

Figure 9.13 Comparaison du temps d'exécution estimé et simulé pour 32 architectures de l'encodeur/décodeur WiMAX

Tableau 9.11 WCT nécessaire à l'estimation et à la mesure par simulation du temps d'exécution des architectures du système de guidage du rover

	WCT de l'estimation (s)	WCT de la simulation (s)	Accélération de l'estimation
Moyenne	1.589	1548	991
Minimum	1.192	745	393
Maximum	2.541	2421	1566

Tableau 9.12 WCT nécessaire à l'estimation et à la mesure par simulation du temps d'exécution des architectures du décodeur JPEG

	WCT de l'estimation (s)	WCT de la simulation (s)	Accélération de l'estimation
Moyenne	1.161	6891	5986
Minimum	0.901	581	395
Maximum	2.267	21268	17853

Tableau 9.13 WCT nécessaire à l'estimation et à la mesure par simulation du temps d'exécution des architectures de l'encodeur/décodeur WiMAX

	WCT de l'estimation (s)	WCT de la simulation (s)	Accélération de l'estimation
Moyenne	0.044	996	23073
Minimum	0.032	182	5592
Maximum	0.082	1954	47358

et de l'encodeur/décodeur WiMAX. Dans tous les cas, ces algorithmes visent à minimiser la valeur de la fonction objective définie à la section 8.3, qui est la mesure de qualité des solutions trouvées par les algorithmes d'exploration architecturale. Plus cette valeur est petite, meilleure est la solution trouvée par l'algorithme. Le calcul de la valeur de cette fonction objective se base sur différentes métriques et contraintes. L'évaluation des métriques associées à chaque architecture s'effectue avec les méthodes d'estimation présentées au chapitre 7. Pour l'évaluation de la métrique de temps d'exécution, on utilise les mêmes bancs d'essai qu'à la section 9.5.2. On configure également le simulateur de performance présenté à la section 7.2.4 avec un facteur d'accélération (décrit à la section 7.2.3.4) de 100 pour la performance du RTOS et de l'API logicielle. En attendant que les optimisations suggérées pour le logiciel embarqué à la section 10.2.5 soient réalisées, cela assure que la performance actuelle du RTOS et de l'API logicielle de SPACE ne viennent pas dominer le temps d'exécution des architectures et que les algorithmes d'exploration architecturale puissent être comparés entre eux alors qu'ils explorent une plus large gamme d'architectures logiciel/matériel.

Comme on vise à ce que la solution retenue n'ait aucune violation fonctionnelle, la contrainte V_{MAX} du nombre de violations fonctionnelles tolérées est égale à 0 pour tous les exemples présentés dans cette section. Les contraintes $A_{j,MAX}$ sur les quantités de ressources matérielles de type j dépendent de la plate-forme ciblée par l'exploration architecturale. Le tableau 9.14 présente les ressources matérielles disponibles sur les plates-formes cible des exemples de cette section, qui sont tous des FPGA Virtex de Xilinx. Les contraintes T_{MAX} de temps d'exécution sont précisées séparément pour chaque exemple.

Les algorithmes d'exploration architecturale ont été appliqués à l'exemple du décodeur JPEG avec détection de peau selon les six configurations de plates-formes cible et de contraintes T_{MAX} données au tableau 9.15. Pour chacune des plates-formes cible, on suppose que 25% des bascules, des LUTs et des multiplieurs ainsi que 80% des mémoires BRAM sont disponibles pour l'application du JPEG. Il est à noter que si il y a suffisamment de ressources matérielles pour implémenter tous les modules en matériel, ou si il n'y a pas suffisamment de mémoire BRAM pour implémenter des modules en logiciel, alors le partitionnement logiciel/matériel trivial trouvé par les algorithmes d'exploration architecturale est de mettre l'ensemble des modules en matériel. De manière analogue, les six configurations de plate-forme cibles et de T_{MAX} énumérées au tableau 9.16 ont été utilisées pour les explorations architecturales de l'encodeur/décodeur WiMAX. Le WiMAX cible ces plates-formes en supposant que 35% des bascules et des LUTs et que 100% des mémoires BRAM et des multiplieurs sont disponibles pour l'application du WiMAX. Finalement, le système de guidage du rover a été testé en ciblant une plate-forme XC2V2000 avec une disponibilité de 100% des mémoires BRAM et des multiplieurs et selon les quatre configurations du tableau 9.17 pour ce qui est

de la contrainte T_{MAX} et de la disponibilité des bascules et des LUTs.

Les algorithmes d'exploration architecturale sont comparés entre eux non seulement selon la valeur de la fonction objective de leur meilleure solution, mais également selon leur temps d'exploration, qui désigne le temps pris (WCT) pour exécuter l'algorithme d'exploration architecturale. Plus ce temps est petit, plus rapide est l'algorithme et meilleur il est selon ce critère. Ce temps n'inclut pas le temps nécessaire à la caractérisation de l'application ou de la plate-forme, tel que présenté au chapitre 7. En effet, cette caractérisation a déjà été faite avant l'exploration architecturale et elle est la même pour tous les algorithmes évalués.

Tableau 9.14 FPGA Virtex ciblés par les tests d'exploration architecturale

Modèle	Package	Vitesse	Bascules	LUTs	BRAMs	Multiplieurs
XC2V2000	FF896	-6	21504	21504	56	56
XC4VFX60	FF672	-10	50560	50560	232	128
XC4VFX100	FF1152	-10	84352	84352	376	160
XC4VFX140	FF1517	-10	126336	126336	552	192

Tableau 9.15 Plate-forme cibles et contraintes T_{MAX} pour le décodeur JPEG avec détection de peau

Configuration	Plate-forme	T_{MAX}
J_a	XC4VFX60	0.75 s
J_b	XC4VFX60	1.00 s
J_c	XC4VFX100	0.75 s
J_d	XC4VFX100	1.00 s
J_e	XC4VFX140	0.75 s
J_f	XC4VFX140	1.00 s

Tableau 9.16 Plate-forme cibles et contraintes T_{MAX} pour l'encodeur/décodeur WiMAX

Configuration	Plate-forme	T_{MAX}
W_a	XC4VFX60	0.2 s
W_b	XC4VFX60	0.3 s
W_c	XC4VFX100	0.2 s
W_d	XC4VFX100	0.3 s
W_e	XC4VFX140	0.2 s
W_f	XC4VFX140	0.3 s

Tableau 9.17 Disponibilité des bascules et LUTs et contraintes T_{MAX} pour le système de guidage du rover

Configuration	Disponibilités des bascules et LUTs	T_{MAX}
R_a	25%	1.5 s
R_b	25%	2.0 s
R_c	100%	1.5 s
R_d	100%	2.0 s

9.6.1 Espace de recherche et analyse combinatoire

L'exploration architecturale du système de guidage du rover et l'encodeur/décodeur WiMAX s'effectue parmi l'ensemble de solutions généré par la fonction de définition de l'espace de recherche f_1 présentée à la section 8.1. Ainsi, cette exploration architecturale consiste à trouver un partitionnement logiciel/matériel des modules de la spécification exécutable, un nombre de processeurs à allouer, une assignation des tâches logicielles aux processeurs, un nombre de bus à allouer et une assignation des composants matériels aux bus. L'exploration architecturale du décodeur JPEG avec détection de peau s'effectue quant à elle selon la fonction de définition de l'espace de recherche f_2 présentée à la section 8.1. La particularité de f_2 tient au fait que certains composants matériels (les mémoires RAM) peuvent être simultanément assignés à deux bus différents au lieu d'être obligatoirement assigné à un seul et même bus.

Le système de guidage du rover comprend trois modules (le filtre, le calculateur d'angle et le détecteur d'objet) qui peuvent être implémentés en logiciel ou en matériel, deux modules (l'analyseur d'image et le contrôleur des moteurs) qui doivent être implémentés en logiciel et un composant matériel (l'UART). En appliquant la formule C.14 de la section C.1.5, on obtient que l'espace de recherche généré par f_1 pour le rover comprend $E(3,2,1) = 2336$ solutions.

L'encodeur/décodeur WiMAX contient quant à lui 15 modules qui peuvent tous être implémentés en logiciel ou en matériel. On applique donc la formule C.6 de la section C.1.4 pour obtenir que l'espace de recherche engendré par f_1 pour l'encodeur/décodeur WiMAX est composé de $E(15) = 546114702621611 \approx 5.4611 \times 10^{14}$ solutions.

Le décodeur JPEG est composé de six modules qui peuvent être implémentés en logiciel et en matériel et de trois composants matériels qui peuvent chacun être assignés à deux bus différents. Si l'exploration architecturale s'effectuait selon f_1 (sans permettre l'assignation d'un composant matériel à deux bus), alors le nombre de solutions serait obtenu par la formule C.14 et on aurait alors $E(6,0,3) = 852219$ solutions. Étant donné que l'exploration

architecturale s'effectue selon f_2, c'est plutôt la formule C.29 qui s'applique et, selon le tableau C.1, l'espace de recherche contient donc $E_2(6,3) \geq 21034003$ solutions. Permettre à chacune des trois mémoires RAMs d'être assignée à deux bus différents multiplie ainsi par au moins 24.68 la taille de l'espace de recherche.

Si on suppose que le temps moyen nécessaire à l'évaluation d'une architecture sera égal aux temps moyen présentés (selon l'étude de cas) aux tableaux 9.11 à 9.13, alors l'évaluation de toute les solutions du système de guidage du rover demande 3712 secondes, celle de toutes les solutions du décodeur JPEG avec détection de peau demande 285 jours et celle de toutes les solutions de l'encodeur/décodeur WiMAX prend 767910 ans. Ainsi, la rapidité des méthodes d'estimation rend possible une exploration architecturale exhaustive pour le système de guidage du rover, mais la grande complexité combinatoire des problèmes d'exploration architecturale fait en sorte que des heuristiques approximatives doivent être utilisées pour le décodeur JPEG et le codec WiMAX.

9.6.2 Comparaisons des algorithmes de recherche locale

Les algorithmes de recherche locale présentés à la section 8.4.4, soit la marche aléatoire, la descente, le recuit simulé adaptatif et la recherche tabou réactive, ont été appliqués au système de guidage du rover, au décodeur JPEG avec détection de peau et à l'encodeur/décodeur WiMAX. Ces algorithmes ont été appliqués séparément pour les problèmes de la maximisation de la performance et de la minimisation de la quantité de ressources matérielles, et ce pour chacune des configurations de contraintes énumérées aux tableaux 9.15 à 9.17. Les solutions initiales fournies à ces algorithmes de recherche locale ont été générées avec l'algorithme glouton à biais logiciel (présenté à la section 8.4.3). L'algorithme exhaustif du parcours en profondeur présenté à la section 8.4.2 a également été appliqué au système de guidage du rover afin de trouver la solution optimale aux problèmes d'exploration architecture qui portent sur cette application.

Les figures 9.14 et 9.15 comparent respectivement la valeur objective de la meilleure solution et le WCT des algorithmes de recherche locale appliqués au problème de la maximisation de la performance. Ces algorithmes sont configurés pour 1000 itérations dans le cas de l'encodeur/décodeur WiMAX, pour 250 itérations dans le cas du décodeur JPEG avec détection de peau et pour 10 itérations dans le cas du système de guidage du rover. Les figures 9.16 et 9.17 comparent de manière similaire l'application des algorithmes de recherche locale au problème de la minimisation de la quantité de ressources matérielles. Les figures 9.14 et 9.16 tronquent la valeur de la fonction objective à 1.4. Si l'exploration architecturale effectuée par un algorithme dépasse cette valeur, alors il suffit, pour les fins de l'interprétation des résultats, de souligner que cet algorithme a échoué dans ce cas et la valeur exacte de la fonction

objective importe alors peu. En effet, si la valeur de la fonction objective est supérieure à 1 pour une architecture donnée, cela signifie que cette architecture ne respecte pas au moins une des contraintes du problème.

On observe dans tous les cas que la descente est beaucoup plus rapide que la marche aléatoire, le recuit simulé ou la recherche tabou (son WCT varie entre 4 et 101 secondes et est à peine visible sur les figures). Cela s'explique par le fait que la descente se termine dès qu'un minimum local est atteint. Aussi, le WCT du recuit simulé adaptatif est généralement plus élevé que celui de la recherche tabou réactive. En effet, le WCT du recuit simulé adaptatif augmente lorsque celui-ci ralentit le refroidissement ou augmente la température. Le recuit simulé adaptatif obtient des résultats mitigés : bien qu'il réussisse fréquemment à obtenir de meilleures solution que la marche aléatoire ou la descente, il arrive fréquemment qu'il ne réussisse pas à obtenir une solution adéquate pour l'encodeur/décodeur WiMAX. Quant à la recherche tabou réactive, elle obtient dans tous les cas des résultats au moins aussi bons que la marche aléatoire, la descente et le recuit simulé adaptatif, et il arrive fréquemment qu'elle obtienne la meilleure solution parmi ces algorithmes. La recherche tabou réactive a réussi à obtenir une solution acceptable dans toutes ces configurations. On observe également que la recherche tabou réactive est le seul de ces algorithmes qui ait trouvé une solution optimale pour toutes les configurations du rover : elle trouve les mêmes solutions que celles trouvées par le parcours en profondeur pour le rover, et ce en seulement 10 itérations. Par contre, cela ne garantit pas que la recherche tabou peut trouver des solutions optimales pour des applications plus complexes en un nombre restreint d'itérations.

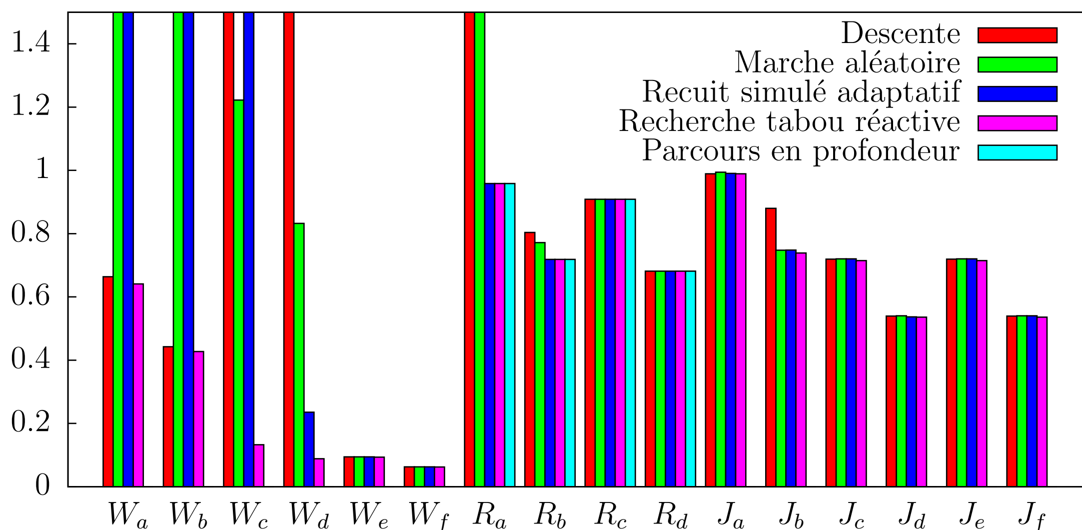

Figure 9.14 Comparaison des valeurs objectives obtenues par les algorithmes de recherche locale pour la maximisation de la performance pour différentes configurations

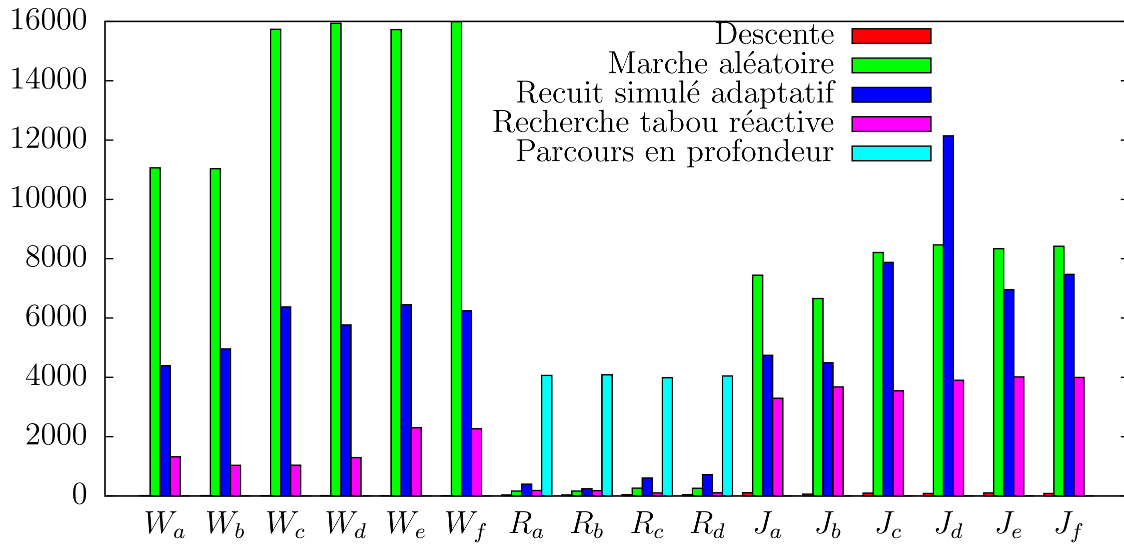

Figure 9.15 Comparaison du WCT (en secondes) pour les algorithmes de recherche locale pour la maximisation de la performance pour différentes configurations

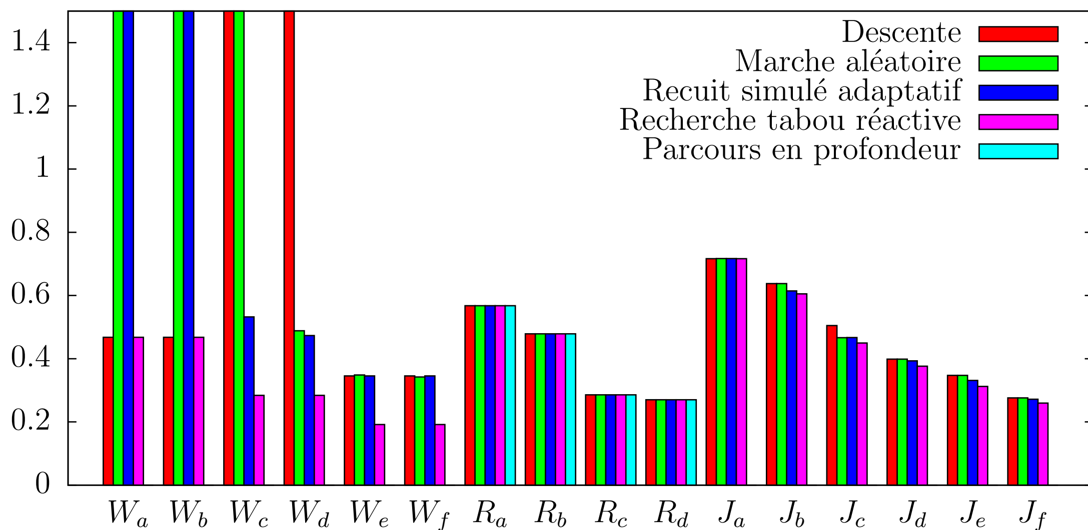

Figure 9.16 Comparaison des valeurs objectives obtenues par les algorithmes de recherche locale pour la minimisation de la quantité de ressources matérielles pour différentes configurations

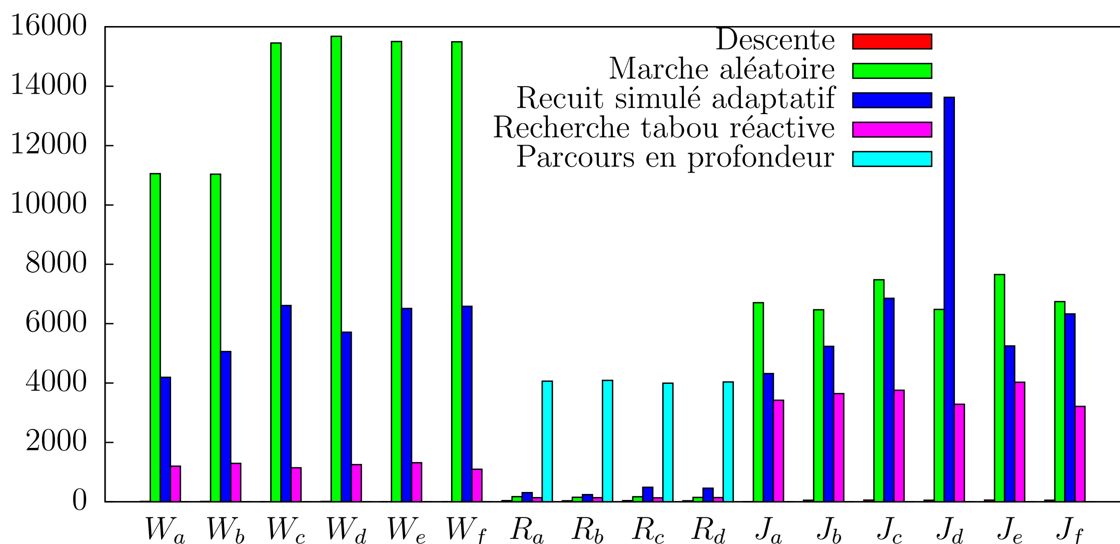

Figure 9.17 Comparaison du WCT (en secondes) pour les algorithmes de recherche locale pour la minimisation de la quantité de ressources matérielles pour différentes configurations

Les figures 9.18 et 9.19 comparent l'application du recuit simulé et de la recherche tabou au décodeur JPEG lorsque ces algorithmes sont configurés pour 250 ou 1000 itérations. Cela illustre la convergence plus rapide de la recherche tabou réactive par rapport au recuit simulé adaptatif. Ainsi, dans tous les cas sauf un, la meilleure solution trouvée par la recherche tabou réactive après 250 itérations est la même qu'après 1000 itérations. Par contre, d'augmenter le nombre d'itérations à 1000 permet au recuit simulé de passer plus de temps à explorer l'espace de recherche à des températures plus basses et le recuit simulé adaptatif arrive à obtenir des solutions meilleurs ou aussi bonnes que la descente pour le décodeur JPEG. Par contre, la recherche tabou réactive obtient encore des solutions meilleures ou égales à celles du recuit simulé adaptatif après 1000 itérations. En fait, dans tous les cas sauf un, cela reste vrai même si la recherche tabou réactive est configurée pour seulement 250 itérations et que le recuit simulé adaptatif est configuré pour 1000 itérations. Étant donné que la recherche tabou est déjà plus rapide que le recuit simulé pour un nombre d'itérations égal, il est donc préférable d'utiliser la recherche tabou réactive pour trouver la meilleure solution possible dans un espace de recherche trop grand pour faire l'objet d'une recherche exhaustive. La descente peut également trouver dans certains cas des solutions intéressantes en un temps très court, alors elle peut être effectuée dans un premier temps avant de déterminer si une exploration architecture plus poussée avec la recherche tabou est nécessaire.

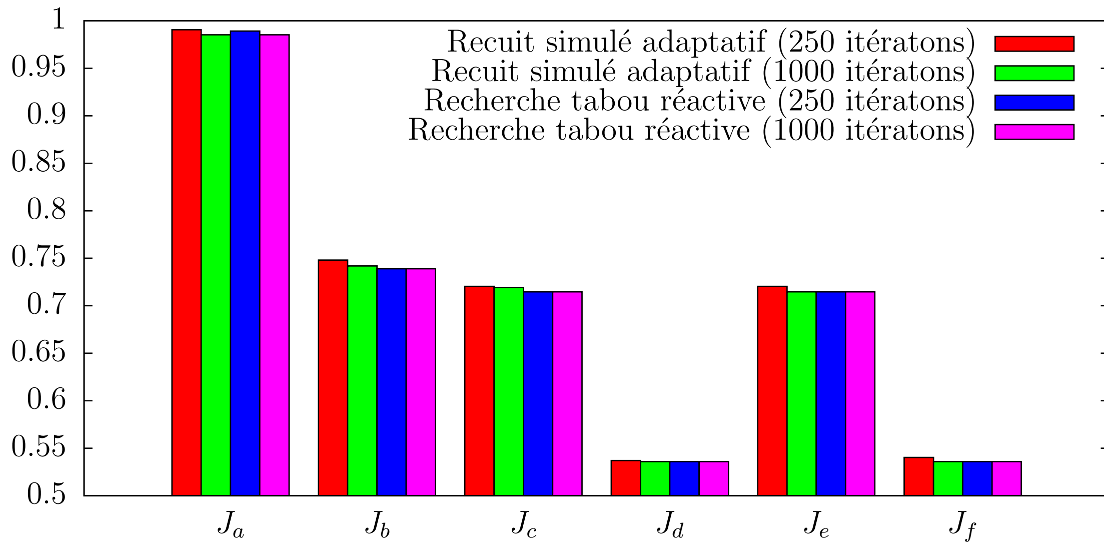

Figure 9.18 Comparaison des valeurs objectives selon le nombre d'itérations pour les algorithmes de recherche locale pour la maximisation de la performance du décodeur JPEG sur différentes plates-formes cibles

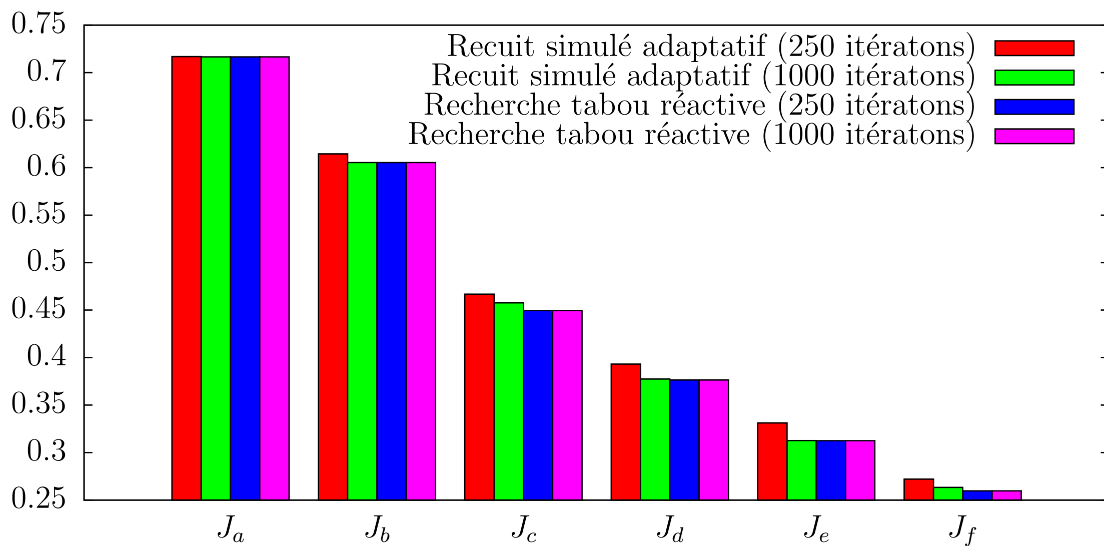

Figure 9.19 Comparaison des valeurs objectives selon le nombre d'itérations pour les algorithmes de recherche locale pour la minimisation de la quantité de ressources matérielles du décodeur JPEG sur différentes plates-formes cibles

9.6.3 Caractéristiques des solutions obtenues

Les caractéristiques des solutions obtenues dépendent à la fois des caractéristiques de l'application, de la plate-forme cible, des contraintes et de l'objectif à atteindre. Ainsi, si l'objectif est la maximisation de la performance et que les contraintes de quantité de ressources matérielles sont peu sévères, alors les algorithmes d'exploration architecturale auront tendance à implémenter tous les modules en matériel et à les assigner sur plusieurs bus afin d'accélérer les communications. Ainsi, on obtiendra alors l'architecture J_0 présentée à la section 9.4.1 pour l'exemple du décodeur JPEG avec détection de peau. À l'inverse, si l'objectif est la minimisation de la quantité de ressources matérielles et que les contraintes de performance sont peu sévères, alors la solution obtenue comprendra typiquement un bus et un processeur sur lequel tous les modules sont assignés en tant que tâches logicielles.

Les solutions obtenues tombent entre ces deux extrêmes lorsque les contraintes deviennent plus sévères. Par exemple, dans le cas de la maximisation de la performance du décodeur JPEG avec détection de peau, le resserrement des contraintes de quantité de ressources matérielles (tel qu'aux configurations J_e et J_f) force le module IDCT à être implémenté en logiciel sur un premier processeur. On obtient alors l'architecture J_1 présentée à la section 9.5.1. Il peut sembler contre-intuitif d'implémenter ce module en logiciel, mais ce module consomme beaucoup de ressources matérielles lorsqu'il est implémenté en matériel et l'algorithme détermine qu'il est préférable, pour respecter les contraintes de ressources, d'implémenter ce seul module en logiciel plutôt que d'avoir à le faire pour plusieurs autres modules. Un resserrement encore plus important des contraintes de ressources matérielles (tel qu'aux configurations J_a et J_b) force l'assignation du module extracteur en tant que tâche logicielle sur le premier processeur et l'allocation d'un deuxième processeur sur lequel s'exécutent les modules quantificateur inverse et décodeur Huffman. On obtient alors l'architecture J_2 de la section 9.5.1. On peut se demander pourquoi un troisième processeur et un quatrième processeur ne sont pas alloués pour exécuter ces quatre modules logiciels. La raison est qu'un processeur consomme une quantité significative de ressources matérielles et que cela entraînerait une violation des contraintes de ressources matérielles. En effet, tel qu'illustré aux figures 9.6 à 9.9, l'allocation d'un processeur peut entraîner une hausse ou une baisse de la quantité de ressources matérielles utilisées : il faut considérer les ressources matérielles utilisées par le processeur supplémentaire et celles économisées par le déplacement d'un ou plusieurs modules du matériel vers le logiciel.

On observe un processus inverse dans le problème de la minimisation de la quantité de ressources matérielles, plus la contrainte de performance devient sévère, plus l'algorithme doit allouer des processeurs, allouer des bus et déplacer des modules du logiciel au matériel pour respecter les contraintes de performance.

En bref, les algorithmes de recherche locale proposés, et plus particulièrement la recherche tabou réactive, permettent d'explorer dans un temps raisonnable des milliers d'architectures potentielles et de trouver des bonnes solutions architecturales pour diverses applications embarquées ciblant différentes plates-formes et selon des contraintes et des critères paramétrables de performance et de quantité de ressources matérielles.

CHAPITRE 10

CONCLUSION

10.1 Synthèse des travaux

Nous avons présenté une méthodologie de niveau système pour la conception, l'exploration architecturale et la synthèse des systèmes embarqués basée sur la technologie Space Codesign™ et sa plate-forme virtuelle SPACE. Cette méthodologie combine un modèle de calcul expressif, une méthode de synthèse matérielle automatisée des modules d'une spécification SystemC, un profilage non-intrusif au niveau système, une méthode de caractérisation automatisée de l'application et du système d'exploitation temps-réel (RTOS), ainsi que des heuristiques pour une formulation unifiée du problème d'exploration architecturale. La méthodologie a été appliquée à trois études de cas : un système de guidage d'un astromobile, un décodeur JPEG avec détection de peau et un encodeur/décodeur WiMAX.

Ainsi, nous avons montré comment ce nouveau modèle de calcul, les réseaux de processus temps-réel (RTPN), permet de modéliser des aspects importants du traitement temps-réel tels que la scrutation, les senseurs échantillonnés, les périphériques d'entrée/sortie et les contraintes temps-réel. La sémantique dénotationnelle des RTPN a également été définie, ce qui a permis de développer une méthode qui vérifie si le raffinement d'une spécification exécutable SystemC vers une implémentation concrète est fonctionnellement correct.

Nous avons présenté une méthode qui automatise le raffinement des communications transactionnelles vers des protocoles précis au cycle et à la broche près ainsi que la génération des blocs matériels pour les modules de l'application. Nous avons également montré comment cette méthode permet, conjointement avec une méthode de génération de code embarqué incluant un RTOS, de générer une implémentation de l'application, qui peut ensuite être simulée avec la plate-forme virtuelle ou synthétisée et exécutée sur la cible finale. Cette méthode de synthèse matérielle a été appliquée avec succès à plusieurs modules, certains aussi complexes qu'un décodeur Reed-Solomon ou Huffman.

Nous avons montré comment un logiciel peut être profilé de manière non-intrusive sur un ISS afin d'extraire le temps à l'entrée et à la sortie de chaque appel de fonction de même que la valeur des arguments passés en paramètre et de la valeur de retour. Ce profilage du logiciel a été utilisé pour développer un profilage au niveau système qui permet d'extraire non-intrusivement des données sur la performance des modules logiciels et matériels, des processeurs, du RTOS, des bus et des mémoires à partir d'une simulation du système. Les

résultats montrent que ce profilage non-intrusif a un impact minime sur la vitesse de simulation lorsque l'architecture contient au moins un processeur et lorsque le profilage logiciel est directement intégré avec l'ISS.

Nous avons présenté une méthode automatisée pour la caractérisation du RTOS, de la fonctionnalité de l'application et des implémentations logicielles et matérielles de ses modules. Un simulateur de performance a été présenté pour l'estimation précise et rapide, selon ces caractérisations, de la performance d'un ensemble d'architectures pour l'application, et ce en tenant compte de la contention sur les bus et de l'ordonnancement des tâches sur les processeurs. Cette méthode d'estimation a permis d'évaluer le temps d'exécution d'un ensemble d'architectures avec une précision de 8% et une vitesse de 400 à 48000 fois plus rapide qu'une simulation complète. Cette caractérisation a également été appliquée à une méthode d'estimation précise et rapide des besoins en ressources matérielles, qui a permis d'estimer la quantité de ressources matérielles d'un ensemble d'architectures avec une précision de 20% et avec une vitesse 200000 fois plus rapide que la synthèse logique avec placement.

Nous avons formulé le problème d'exploration architecturale comme une combinaison du partitionnement logiciel/matériel, de l'allocation des processeurs, de l'assignation des tâches aux processeurs et du choix d'une topologie de communication. Des heuristiques de recherche locale, soit la marche aléatoire, le recuit simulé adaptatif et la recherche tabou réactive, ont été définies pour la résolution de ce problème. La recherche tabou réactive a systématiquement obtenu de meilleurs résultats que la marche aléatoire et le recuit simulé adaptatif. Dans les cas où il a été possible de trouver une solution optimale avec un parcours en profondeur, la recherche tabou réactive a également pu trouver cette solution optimale.

10.2 Limites et améliorations futures

Cette section décrit les limites des méthodes présentées ainsi que les travaux de recherche futurs qui permettraient de dépasser ces limites.

10.2.1 Équivalence des ordonnancements

Dans le modèle de calcul des réseaux de processus temps-réel (RTPN) présenté au chapitre 4, on définit deux ordonnancements d'un RTPN comme équivalents si et seulement si ceux-ci produisent dans les deux cas le même ensemble de séquences de jetons pour l'ensemble des ports (d'écriture ou de lecture) des canaux (soit si et seulement si les deux ordonnancements produisent les mêmes séquences de bits définies à la section 4.2.2). Cependant, cette exigence d'une égalité sur les ports de chaque canal $c \in C$ peut être trop restrictive. En effet, il est possible qu'on soit seulement intéressé aux séquences de jetons de certains canaux, par exemple

les canaux de sortie $O \subseteq C$ du système. Dans ce cas, on pourrait affaiblir la condition d'équivalence pour que deux ordonnancements de ce RTPN soient équivalents si et seulement si ils produisent les mêmes séquences de jetons sur l'ensemble des ports de $O \subseteq C$. Il s'agit d'une distinction similaire à celle qui existe entre un système déterministe, dont toutes les opérations et donc les sorties sont déterministes, et un système faiblement non-déterministe, dont les opérations peuvent être non-déterministes mais dont les sorties demeurent déterministes (Sondergaard et Sestoft, 1992).

Le processus p dont le code est fourni à la figure 10.1 en est un exemple trivial. Supposons que son canal d'entrée est un canal POLL i et son canal de sortie un canal Kahn o. Selon l'ordonnancement de p et du processus qui écrit dans i, il est possible que p effectue des lectures non-bloquantes sur i alors que i est vide et donc que des jetons de valeur absente soient ajoutés à la séquence de jetons au port de lecture de i. Par contre, tous ces jetons supplémentaires sont ignorés par p étant donné la boucle d'attente active qui implémente effectivement une lecture bloquante sur i. Dans ce cas, la séquence de jetons sur o ne dépend pas de l'ordonnancement de p et du processus qui écrit sur i. Deux ordonnancements peuvent donc ne pas être équivalents selon la définition de la section 4.2.2 (avec des jetons de valeur absente à des endroits différents sur le port de lecture de i) tout en étant équivalents selon une définition qui considère seulement le canal o.

Il est à noter qu'établir cette équivalence par rapport à o a nécessité une inspection du code du processus p. Cela contraste avec la définition de la section 4.2.2 qui exige seulement que le processus p soit continu au sens de Scott, ce qui est garanti si il est modélisé dans le langage de (Kahn, 1974) qui est similaire à celui de SPACE. Une avenue de recherche future serait de définir une méthode d'analyse statique du code des processus P d'un RTPN qui indique sous quelles conditions deux ordonnancements d'un RTPN sont équivalents pour un ensemble de canaux $O \subseteq C$. Il est probable que le problème consistant à trouver les conditions qui sont à la fois nécessaires et suffisantes ne soit pas décidable. Il faudrait ensuite que la méthode de caractérisation présentée à la section 7.2 puisse être configurée selon les conditions suffisantes extraites par l'analyse statique, afin que la spécification et une implémentation d'une application soient considérées fonctionnellement équivalentes si ces conditions sont respectées.

10.2.2 Consommation de puissance et d'énergie

La méthodologie proposée évalue les différentes architectures d'une application selon des critères de performance et de coût matériel. D'autres critères sont cependant importants pour la conception des systèmes embarqués, soit la consommation de puissance et d'énergie (Wolf *et al.*, 2008). Une amélioration future de la méthodologie proposée serait donc d'y intégrer

```
p(in dataIn, out dataOut) {
    while(1) {
        do { x=dataIn.read(); } while(x == EMPTY_TOKEN);
        y=f(x);
        dataOut.write(y);
    }
}
```

Figure 10.1 Un processus RTPN qui utilise une lecture non-bloquante comme une lecture bloquante

des métriques de consommation de puissance et d'énergie.

Cette intégration impliquerait de définir une métrique de consommation de puissance moyenne $P : A \to \mathbb{R}$ et une métrique de consommation d'énergie $E : A \to \mathbb{R}$ et on aurait, par définition, $E(a) = P(a)T(a)$ pour toute architecture a. Si on dispose d'un moyen d'évaluer $P(a)$ ou $E(a)$, il est alors aisé d'ajouter des contraintes de puissance ou d'énergie aux problèmes d'exploration architecturale présentés à la section 8.2 ou de définir des nouveaux problèmes qui consistent à minimiser la consommation de puissance ou d'énergie. Les fonctions objectives pour l'évaluation des architectures seraient alors directement ajustées selon la formule générale de la section 8.3 et tous les algorithmes d'exploration architecturale présentés à la section 8.4 pourraient être directement appliqués avec ces nouvelles fonctions objectives.

Le principal défi serait donc de développer une méthode suffisamment précise et rapide pour l'évaluation des métriques $E(a)$ ou $P(a)$. Des travaux sont présentement en cours pour intégrer une méthode d'estimation de la consommation de puissance et d'énergie dans la plate-forme virtuelle SPACE (Rogers-Vallée *et al.*, 2010). Cette méthode consiste en la définition d'un modèle de consommation de puissance et d'énergie pour les processeurs, les bus et les mémoires et en la caractérisation de ces modèles pour un FPGA Xilinx par un ensemble de simulations au niveau RTL avec ModelSim (Mentor Graphics Corp., 2008) et XPower (Wenande et Chidester, 2001). Ces modèles peuvent ensuite estimer la puissance moyenne et l'énergie consommée selon les informations fournies par une simulation de niveau transactionnel. Par exemple, la simulation transactionnelle doit fournir au modèle associé à un processeur l'adresse (PC) et le type de chaque instruction exécutée alors que le modèle associé à une mémoire doit recevoir l'adresse et la donnée de chaque accès à la mémoire (Rogers-Vallée *et al.*, 2010). Afin de permettre l'évaluation de différents partitionnements logiciel/matériel, il serait nécessaire que cette méthode permette de définir et de caractériser automatiquement des modèles de consommation de puissance et d'énergie pour les implémentations matérielles des modules générées selon la méthode présentée au chapitre 5.

Cependant, même après cette caractérisation, l'utilisation de ces modèles de consommation demande une simulation complète approximativement précise au cycle près, où chaque instruction et chaque accès mémoire sont simulés séparément. Pour accélérer l'évaluation d'un grand nombre d'architectures, il serait donc nécessaire d'effectuer une deuxième ronde de caractérisation pour permettre l'estimation de la performance et de l'énergie au niveau du simulateur de performance présenté à la section 7.2.4. Cette deuxième ronde de caractérisation associerait par exemple une consommation de puissance et d'énergie aux différentes exécutions des segments de programme des modules et aux opérations du RTOS, tel que les changements de contexte. Les simulations réalisées pour caractériser le temps d'exécution des segments de programme des modules, tel que présenté à la section 7.2.2, pourraient ainsi également servir à caractériser leur consommation de puissance et d'énergie.

10.2.3 Temps de caractérisation des modules logiciels

Tel que décrit à la section 7.2.2, la caractérisation du temps d'exécution logiciel des modules nécessite, pour chacun de ceux-ci, une simulation SPACE à niveau d'abstraction mixte qui inclut un ISS exécutant l'implémentation logicielle du module. Une telle simulation est relativement lente et le nombre de simulations nécessaires à la caractérisation croît linéairement avec le nombre n de modules dans l'application. Cette caractérisation est complètement automatisée et l'utilisateur peut donc effectuer d'autres activités pendant ce temps : elle ne se trouve donc généralement pas sur le chemin critique du projet. Par contre, il est possible que l'utilisateur veuille effectuer une co-exploration de l'algorithme et de l'architecture de l'application (Lee *et al.*, 2009), ce qui impliquerait que l'exploration architecturale soit lancée pour une spécification exécutable qui contient une première version de l'algorithme, que celui-ci soit modifié en fonction des résultats de l'exploration architecturale, que celle-ci soit lancée de nouveau et ainsi de suite. Chaque modification de la spécification exécutable implique une nouvelle caractérisation des modules logiciels et il serait alors nécessaire de l'accélérer.

Le temps nécessaire à la caractérisation des modules logiciels pourrait être diminué en ayant recours aux méthodes présentées à la section 2.4.2.1. Ainsi, ces méthodes pourraient être utilisées pour générer un modèle transactionnel précisément temporisé d'un module logiciel à l'aide d'une analyse statique des blocs de base du module logiciel. La simulation utilisée pour caractériser les segments de programme du module logiciel n'aurait alors plus besoin d'exécuter celui-ci instruction par instruction sur un ISS, mais pourrait simplement faire appel à son modèle transactionnel temporisé. Cela amènerait une nette accélération de ces simulations et donc de la caractérisation. Si des métriques de consommation de puissance et d'énergie étaient ajoutées à la méthodologie, tel que discuté à la section 10.2.2, alors une méthode similaire permettrait de caractériser les modules logiciels selon ces métriques.

La puissance et l'énergie associée à chaque instruction seraient caractérisées à l'aide de la méthode de (Rogers-Vallée *et al.*, 2010), puis la puissance et l'énergie de chaque bloc de base seraient évaluées par une analyse statique comme celle de (Tiwari *et al.*, 1996), et finalement la puissance et l'énergie de chaque exécution d'un segment de programme seraient caractérisées par des simulations transactionnelles tel que discuté à la section 10.2.2.

10.2.4 Généralisation de l'exploration architecturale

L'exploration architecturale présentée au chapitre 8 s'effectue en supposant que chaque module de l'application a au plus une implémentation logicielle et une implémentation matérielle et que les processeurs et les bus alloués sont homogènes. Il serait possible de généraliser l'exploration architecturale en permettant que chaque module ait un nombre arbitraire d'implémentations, que les processeurs soient hétérogènes et que des topologies de communication arbitraires soient explorées.

Ainsi, la méthode de synthèse matérielle présentée à la section 5 pourrait être modifiée pour générer plusieurs implémentations matérielles d'un même module en utilisant différentes options lors de la synthèse comportementale. Cette étape pourrait même faire l'objet d'une exploration micro-architecturale afin de trouver un ensemble Pareto-optimal d'implémentations matérielles d'un module (Chtourou et Hammami, 2005). De la même manière, la méthode de synthèse logicielle présentée à la section 3.2.5.1 pourrait cibler différents processeurs ou différentes configurations d'un même processeur. Pour un processeur extensible tel que le Xtensa (Gonzalez, 2000), cela pourrait même impliquer de générer automatiquement des nouvelles instructions selon le code du module (Goodwin et Petkov, 2003; Sun *et al.*, 2003).

La caractérisation du temps d'exécution et du coût matériel de chaque implémentation matérielle d'un module se ferait alors selon les méthodes présentées respectivement aux sections 7.2.2 et 7.3.1. La recherche locale présentée à la section 8.4.4 devrait alors être étendue pour inclure un mouvement qui consiste à remplacer une implémentation matérielle d'un module par une autre implémentation matérielle du même module. De la même manière, le temps d'exécution et la mémoire logicielle de chaque implémentation logicielle d'un module seraient caractérisés selon les méthodes présentées respectivement aux sections 7.2.2 et 7.3.2. Le coût matériel de chaque processeur (ou chaque configuration d'un processeur) serait caractérisé selon la méthode présentée à la section 7.3.3. La recherche locale présentée à la section 8.4.4 serait alors étendue pour y ajouter un mouvement qui remplace un processeur par un autre processeur (ou une autre configuration du même processeur) et modifie en conséquence les implémentations logicielles des modules assignés à ce processeur.

Une configuration de processeur qui demanderait un traitement particulier serait l'ajout

d'une mémoire cache. En effet, la méthode de caractérisation présentée à la section 7.2.2 suppose que, bien qu'un changement de contexte ou une interruption puissent retarder l'exécution d'un segment de programme en suspendant le module auquel il appartient, le temps pris par le module pour exécuter un chemin donné de ce segment, lorsqu'il détient le processeur, est constant. Cette supposition se justifie pour un processeur sans cache alors que le temps d'exécution d'un code est déterminé essentiellement par des facteurs locaux. Par exemple, même si un changement de contexte vient modifier le contenu du pipeline du processeur au milieu de l'exécution d'un segment, cela aura un impact seulement sur un nombre d'instructions au pire égal à la taille du pipeline (typiquement 5 ou moins pour les processeurs embarqués). Par contre, les effets d'un changement de contexte sur le taux de défauts (*miss*) de cache, et donc sur le temps d'exécution, peuvent se faire sentir même plusieurs milliers d'instructions après le changement de contexte (Mogul et Borg, 1991). La caractérisation présentée à la section 7.2.2 tiendrait donc seulement compte des défauts de cache intrinsèques au module logiciel sans tenir compte de ceux causés par le changement de contexte. Pour en tenir compte, il serait possible d'extraire, lors de la caractérisation des modules logiciels, les traces des adresses mémoire auxquelles chaque module logiciel accède. Le simulateur de performance pourrait alors être configuré avec, pour chaque processeur, un modèle analytique de cache (Agarwal *et al.*, 1989; Liu *et al.*, 2008) qui serait construit à partir des traces des modules assignés à ce processeur et qui pourrait estimer les défauts de cache dus aux changements de contexte.

Finalement, il serait possible d'ajouter à l'exploration architecturale d'autres liens de communications que des bus tel que des liens point à point ou des réseaux sur puces (NoC). Il faudrait alors caractériser leur délai (section 7.2.3.3) de même que leur coût matériel (section 7.3.3) et les ajouter au simulateur de performance (section 7.2.4). La recherche locale présentée à la section 8.4.4 devrait également être modifiée pour y inclure des mouvements qui ajoutent ou retirent des liens point à point ou des NoC et qui y assignent ou déplacent les composants matériels. L'inclusion de ces mécanismes de communication permettrait d'explorer une plus grande variété de topologie de communication.

10.2.5 Performance du logiciel embarqué

Présentement, la synthèse logicielle réalisée pour SPACE associe une tâche logicielle à chaque module logiciel assigné à un processeur et ces différentes tâches sont ordonnancées dynamiquement par un RTOS. Tel que présenté à la section 7.2.3.4, les délais associés aux changements de contexte et aux communications entre tâches logicielles sont élevés, se chiffrant dans les centaines voire les milliers de cycles pour chaque changement de contexte ou communication. Cela crée une tension au niveau de la granularité de la spécification exécutable : plus elle est fine, mieux elle peut exprimer le parallélisme de tâche, mais plus il risque

d'y avoir une multitude de tâches logicielles ordonnancées dynamiquement sur les processeurs.

Une solution à ce problème serait d'implémenter plusieurs modules logiciels (assignés à un même processeur) en une seule tâche logicielle qui est exécutée séquentiellement. Suite à cette fusion des modules logiciels, les changements de contexte entre ceux-ci seraient éliminés et les communications entre eux deviendraient alors des accès à des variables locales de la tâche plutôt que des appels aux mécanismes de communication et de synchronisation du RTOS. Cela amènerait un gain de performance considérable. Pour ce faire, des méthodes d'ordonnancement quasi-statique (Cortadella *et al.*, 2005; Gu *et al.*, 2010) pourraient être appliquées afin de générer, à partir de m modules assignés à un processeur, t tâches logicielles équivalentes tel que $t \leq m$. Étant donné la généralité de notre modèle de calcul, il n'est pas possible de garantir qu'une seule tâche logicielle puisse implémenter l'ensemble des modules assignés à un processeur.

Dans le cas où $t = 1$, le RTOS pourrait être éliminé et le processeur pourrait simplement exécuter une boucle infinie contenant le code de la tâche, ce qui permettrait des gains autant au niveau de la performance que de la mémoire requise par le logiciel embarqué. Dans le cas où $t > 1$, il serait nécessaire de conserver un ordonnanceur dynamique pour les t tâches. Si on accepte d'avoir un ordonnancement coopératif des tâches logicielles (plutôt que l'ordonnancement préemptif présentement utilisé après la synthèse logicielle avec SPACE), alors il serait possible d'utiliser les méthodes de (Nacul et Givargis, 2006; Cho et Choi, 2009) pour découper les tâches en des segments non-préemptibles, qui seraient alors combinés avec l'ordonnanceur dynamique pour former un programme monolithique. Celui-ci s'exécuterait de manière plus performante qu'avec un RTOS conventionnel.

Ces optimisations diminueraient le temps d'exécution des architectures qui ont des modules assignés en logiciel et pourraient donc faire en sorte que l'exploration architecturale trouve des solutions de meilleure qualité. Il serait alors nécessaire de s'assurer que les méthodes de caractérisation et d'estimation, et donc la fonction objective utilisée par l'exploration architecturale, tiennent compte des nouvelles optimisations réalisées lors de la synthèse logicielle.

10.2.6 Évaluation industrielle de la méthodologie

Un travail futur pertinent serait d'évaluer la méthodologie dans un contexte industriel afin de déterminer l'impact qu'a l'utilisation de la méthodologie sur la durée des projets de conception de systèmes embarqués et sur la qualité des résultats obtenus. Il s'agit essentiellement d'un problème de recherche en génie logiciel pour lequel l'établissement d'un protocole expérimental réaliste et rigoureux est un défi important (Basili *et al.*, 1986; Wohlin *et al.*, 2000; Kitchenham, 2007). Idéalement, deux équipes de travail en tous points comparables

réaliseraient chacune de manière indépendante un important projet de développement de systèmes embarqués, une équipe utilisant la méthodologie et l'autre ne l'utilisant pas, et ce sans que l'une ou l'autre équipe n'ait un biais pour ou contre la méthodologie. Ce protocole expérimental serait par contre difficilement réalisable étant donné la grande quantité de ressources humaines (l'équipe travaillant en double) qui devraient y être consacrées. Un protocole moins rigoureux mais plus réaliste serait qu'une équipe de travail utilise la méthodologie pour réaliser un projet industriel de conception de systèmes embarqués et qu'on compare la durée et la qualité des résultats de ce projet à ceux d'un autre projet similaire précédemment réalisé par la même équipe de travail sans utiliser la méthodologie.

En bref, cette thèse a présenté de nouvelles méthodes et une nouvelle méthodologie de niveau système ayant pour but d'accélérer la conception, l'exploration architecturale et l'implémentation des systèmes embarqués. Cet objectif est d'une grande importance pour les compagnies qui visent à commercialiser des systèmes embarqués et doivent avoir un temps de mise en marché le plus court possible. Il est pertinent de poursuivre les recherches dans ce domaine afin que de telles méthodologies de niveau système soient adoptées par l'industrie pour réaliser plus efficacement leurs projets de développement de systèmes embarqués.

RÉFÉRENCES

ABDI, S., SHIN, D. et GAJSKI, D. (2003). Automatic communication refinement for system level design. *40th Design Automation Conference*. IEEE, Anaheim, CA, 300–5.

ABRAMSKY, S. et JUNG, A. (1994). Domain theory. S. Abramsky, D. M. Gabbay et T. S. E. Maibaum, éditeurs, *Handbook of Logic in Computer Science*, Clarendon Press, Oxford, UK, vol. 3. 1–168.

AGARWAL, A., HOROWITZ, M. et HENNESSY, J. (1989). An analytical cache model. *ACM Transactions on Computer Systems*, 7, 184–215.

AHMED, U. et KHAN, G. N. (2004). Embedded system partitioning with flexible granularity by using a variant of tabu search. *Canadian Conference on Electrical and Computer Engineering 2004*. IEEE, Niagara Falls, Ont., Canada, vol. 4, 2073–6.

AHN, Y., HAN, K., LEE, G., SONG, H., YOO, J., CHOI, K. et FENG, X. (2008). SoCDAL : System-on-chip design accelerator. *ACM Transactions on Design Automation of Electronic Systems*, 13, 38 pp.

ALLEN, F. E. (1970). Control flow analysis. *ACM SIGPLAN Notices*, 5, 1–19.

ALTERA CORP. (2004). Nios Embedded Processor. http ://www.altera.com/products/ip/processors/nios.

ARATO, P., JUHASZ, S., MANN, Z. A., ORBAN, A. et PAPP, D. (2003). Hardware-software partitioning in embedded system design. *2003 IEEE International Symposium on Intelligent Signal Processing*. IEEE, Budapest, Hungary, 197–202.

ARATO, P., MANN, Z. A. et ORBAN, A. (2005). Algorithmic aspects of hardware/software partitioning. *ACM Transactions on Design Automation of Electronic Systems*, 10, 136–56.

ARM LTD. (2010). ARM Processors. http ://www.arm.com/products/processors/.

AUSTIN, T., LARSON, E. et ERNST, D. (2002). SimpleScalar : an infrastructure for computer system modeling. *Computer*, 35, 59–67.

AXELSSON, J. (1997). Architecture synthesis and partitioning of real-time systems : a comparison of three heuristic search strategies. *Proceedings of 5th International Workshop on Hardware/Software Co Design. Codes/CASHE '97*. IEEE Comput. Soc. Press, Braunschweig, Germany, 161–5.

BAGHDADI, A., ZERGAINOH, N. E., CESARIO, W. O. et JERRAYA, A. A. (2002). Combining a performance estimation methodology with a hardware/software codesign flow

supporting multiprocessor systems. *IEEE Transactions on Software Engineering*, 28, 822–31.

BAH, F.-L. (2010). *Conception au niveau système d'une application de protocole sans fil.* Mémoire de maîtrise, École Polytechnique de Montréal.

BAILEY, B., MARTIN, G. et PIZIALI, A. (2007). *ESL Design and Verification : A Prescription for Electronic System-Level Methodology.* Morgan Kaufmann, San Francisco, CA.

BANERJEE, S. et DUTT, N. (2004). Efficient search space exploration for HW-SW partitioning. *International Conference on Hardware/Software Codesign and Systems Synthesis.* ACM, Stockholm, Sweden, 122–7.

BARR, M. et MASSA, A. J. (2006). *Programming embedded systems with C and GNU development tools.* O'Reilly Media, Sebastopol, CA, seconde édition.

BASILI, V., SELBY, R. et HUTCHENS, D. (1986). Experimentation in software engineering. *IEEE Transactions on Software Engineering*, SE-12, 733–43.

BATTITI, R. et TECCHIOLLI, G. (1994). The reactive tabu search. *ORSA Journal on Computing*, 6, 126–40.

BATTITI, R. et TECCHIOLLI, G. (1995). Training neural nets with the reactive tabu search. *IEEE Transactions on Neural Networks*, 6, 1185–200.

BELTRAME, G., FOSSATI, L. et SCIUTO, D. (2009). ReSP : a nonintrusive transaction-level reflective MPSoC simulation platform for design space exploration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28, 1857–69.

BENINI, L., BERTOZZI, D., BRUNI, D., DRAGO, N., FUMMI, F. et PONCINO, M. (2003). SystemC cosimulation and emulation of multiprocessor SoC designs. *Computer*, 36, 53–9.

BERNARDI, O., NOY, M. et WELSH, D. (2007). On the growth rate of minor-closed classes of graphs. http ://arxiv.org/abs/0710.2995v1.

BERNHART, F. R. (1999). Catalan, Motzkin, and Riordan numbers. *Discrete Mathematics*, 204, 73–112.

BILSEN, G., ENGELS, M., LAUWEREINS, R. et PEPERSTRAETE, J. (1996). Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44, 397–408.

BLICKLE, T., TEICH, J. et THIELE, L. (1998). System-level synthesis using evolutionary algorithms. *Design Automation for Embedded Systems*, 3, 23–58.

BOIS, G., MOSS, L., FILION, L. et FONTAINE, S. (2010). Codesign experiences based on a virtual platform. B. Bailey et G. Martin, éditeurs, *ESL Models and their Application :*

Electronic System Level Design and Verification in Practice, Springer, New York, NY. 273–308.

BOLLAERT, T. (2008). Catapult synthesis : A practical introduction to interactive C synthesis. P. Coussy et A. Morawiec, éditeurs, *High-Level Synthesis : From Algorithm to Digital Circuit*, Springer, New York, NY. 29–52.

BROCK, J. D. et ACKERMAN, W. B. (1981). Scenarios : a model of non-determinate computation. *Formalization of Programming Concepts, an International Colloquium*. Springer-Verlag, Peniscola, Spain, 252–9.

BUCK, J. T. (1993). *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. Thèse de doctorat, University of California, Berkeley.

BUCK, J. T., HA, S., LEE, E. A. et MESSERSCHMITT, D. G. (1994). Ptolemy : A framework for simulating and prototyping heterogeneous systems. *International Journal in Computer Simulation*, 4, 155–182.

CADENCE DESIGN SYSTEMS, INC. (2008). Cadence C-to-Silicon compiler delivers on the promise of high-level synthesis. Rapport technique, Cadence Design Systems, Inc.

CAI, L. et GAJSKI, D. (2003). Transaction level modeling : an overview. *First IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis*. ACM, Newport Beach, CA, 19–24.

CAI, L., GERSTLAUER, A. et GAJSKI, D. (2004). Retargetable profiling for rapid, early system-level design space exploration. *Proceedings 2004. Design Automation Conference*. New York, NY, USA, 281 – 6.

CEVA, INC. (2004). CEVA-Teak. http ://www.ceva-dsp.com/products/cores/ceva-teak.php.

CHARALAMBIDES, C. A. (2005). *Combinatorial Methods in Discrete Distributions*. Wiley Series in Probability and Statistics. Wiley-Interscience, Hoboken, NJ.

CHATHA, K. S. et VEMURI, R. (2002). Hardware-software partitioning and pipelined scheduling of transformative applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10, 193–208.

CHEVALIER, J., DE NANCLAS, M., FILION, L., BENNY, O., RONDONNEAU, M., BOIS, G. et ABOULHAMID, M. (2006). A SystemC refinement methodology for embedded software. *IEEE Design & Test of Computers*, 23, 148–58.

CHIEN, R. (1964). Cyclic decoding procedures for Bose-Chaudhuri-Hocquenghem codes. *IEEE Transactions on Information Theory*, IT-10, 357–363.

CHO, Y. et CHOI, K. (2009). Code decomposition and recomposition for enhancing embedded software performance. *Proceedings of the ASP-DAC 2009. 14th Asia and South Pacific Design Automation Conference*. Piscataway, NJ, USA, 624–9.

CHTOUROU, S. et HAMMAMI, O. (2005). SystemC space exploration of behavioral synthesis options on area, performance and power consumption. *Proceedings. The 17th ICM 2005. 2005 International Conference on Microelectronics*. Piscataway, NJ, USA, 67–71.

CLAYBERG, E. et RUBEL, D. (2006). *Eclipse : Building Commercial-Quality Plug-ins*. Addison-Wesley Professional, Boston, MA.

CMP MEDIA LLC (2006). 2006 State of Embedded Market Survey. http ://www.embedded.com/columns/survey.

COMTET, L. (1974). *Advanced combinatorics : the art of finite and infinite expansions*. Reidel Publishing Company, Boston, MA.

CORTADELLA, J., KONDRATYEV, A., LAVAGNO, L., PASSERONE, C. et WATANABE, Y. (2005). Quasi-static scheduling of independent tasks for reactive systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24, 1492–514.

COUSSY, P., GAJSKI, D., MEREDITH, M. et TAKACH, A. (2009). An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26, 8–17.

COWARE INC. (2010). CoWare Platform Architect. http ://www.coware.com/products/platformarchitect.php.

DE BRUIJN, N. G. (1981). *Asymptotic Methods in Analysis*. Dover Publications, New York, troisième édition.

DE KOCK, E. A., SMITS, W. J. M., VAN DER WOLF, P., BRUNEL, J. Y., KRUIJTZER, W. M., LIEVERSE, P., VISSERS, K. A. et ESSINK, G. (2000). YAPI : Application modeling for signal processing systems. *37th conference on Design automation*. ACM Press, Los Angeles, California, United States, 402–05.

DENIZIAK, S. et GORSKI, A. (2008). Hardware/software co-synthesis of distributed embedded systems using genetic programming. *ICES '08 : Proceedings of the 8th international conference on Evolvable Systems : From Biology to Hardware*. Springer-Verlag, Berlin, Heidelberg, 83–93.

DERRICK, E., BALCI, O. et NANCE, R. (1989). A comparison of selected conceptual frameworks for simulation modeling. *1989 Winter Simulation Conference Proceedings*. San Diego, CA, USA, 711–18.

DHODHI, M., AHMAD, I. et STORER, R. (1995). SHEMUS : synthesis of heterogeneous multiprocessor systems. *Microprocessors and Microsystems*, 19, 311–19.

DICK, R. P. et JHA, N. K. (1998). MOGAC : a multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17, 920–35.

DOMER, R., GERSTLAUER, A., PENG, J., SHIN, D., CAI, L., YU, H., ABDI, S. et GAJSKI, D. D. (2008). System-on-chip environment : a SpecC-based framework for heterogeneous MPSoC design. *EURASIP Journal on Embedded Systems*, 2008, 1–13.

DORIGO, M., MANIEZZO, V. et COLORNI, A. (1996). Ant system : optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man and Cybernetics, Part B (Cybernetics)*, 26, 29–41.

ELECTRONICS INDUSTRIES ASSOCIATION (1969). EIA standard RS-232-C interface between data terminal equipment and data communication equipment employing serial data interchange.

ELES, P., PENG, Z., KUCHCHINSKI, K. et DOBOLI, A. (1997). System level hardware/software partitioning based on simulated annealing and tabu search. *Design Automation for Embedded Systems*, 2, 5–32.

ERBAS, C., CERAV-ERBAS, S. et PIMENTEL, A. D. (2006). Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. *IEEE Transactions on Evolutionary Computation*, 10, 358–74.

FAIZ, A. (2007). *Méthodes de raffinement des communications pour passer d'une plateforme SystemC à un système reprogrammable*. Mémoire de maîtrise, École Polytechnique de Montréal.

FALK, J., HAUBELT, C. et TEICH, J. (2006). Efficient representation and simulation of model-based designs in SystemC. *Proc. FDL0́6, Forum on Design Languages 2006*. Darmstadt, Germany, 129–134.

FERNANDEZ-BACA, D. (1989). Allocating modules to processors in a distributed system. *IEEE Transactions on Software Engineering*, 15, 1427–1436.

FILION, L., CANTIN, M.-A., MOSS, L., ABOULHAMID, M. et BOIS, G. (2007). Space Codesign : A SystemC framework for fast exploration of hardware/software systems. *Design & Verification Conference and Exhibition*. San Jose, CA, 8 pp.

FINC, M. et ZEMVA, A. (2005). Profiling soft-core processor applications for hardware/-software partitioning. *Journal of Systems Architecture*, 51, 315–29.

FLAJOLET, P. et SALVY, B. (1995). Computer algebra libraries for combinatorial structures. *Journal of Symbolic Computation*, 20, 653–71.

FLYNN, D. (1997). AMBA : enabling reusable on-chip designs. *IEEE Micro*, 17, 20–27.

FORNEY, JR., G. (1965). On decoding BCH codes. *IEEE Transactions on Information Theory*, IT-11, 549–557.

FREE SOFTWARE FOUNDATION (2010). The GNU project. http ://www.gnu.org.

FUMMI, F., LOGHI, M., PONCINO, M. et PRAVADELLI, G. (2009). A cosimulation methodology for HW/SW validation and performance estimation. *ACM Transactions on Design Automation of Electronic Systems*, 14, 32 pp.

GAISLER, J. (2003). The LEON-2 processor user's manual. Rapport technique, Gaisler Research.

GAJSKI, D. D., ZHU, J., DOMER, R., GERSTLAUER, A. et ZHAO, S. (2000). *SpecC : Specification Language and Methodology*. Kluwer Academic Publishers, Boston, MA.

GAREY, M. R. et JOHNSON, D. S. (1979). *Computers and Intractability : A Guide to the Theory of NP-Completeness*. WH Freeman and Company, San Francisco, CA.

GEILEN, M. et BASTEN, T. (2003). Requirements on the execution of Kahn process networks. *Proceedings of the 12th European Symposium on Programming. Lecture Notes in Computer Science Vol.2618*, Springer-Verlag, Warsaw, Poland. 319–34.

GEILEN, M. et BASTEN, T. (2004). Reactive process networks. *4th ACM international conference on Embedded software*. ACM Press, Pisa, Italy, 137–46.

GERIN, P., HAMAYUN, M. M. et PETROT, F. (2009). Native MPSoC co-simulation environment for software performance estimation. *Embedded Systems Week 2009 - 7th IEEE/ACM International Conference on Hardware/Software-Co-Design and System Synthesis, CODES+ISSS 2009*. Grenoble, France, 403–412.

GERSTLAUER, A., DONGWAN, S., JUNYU, P., DOMER, R. et GAJSKI, D. D. (2007). Automatic layer-based generation of system-on-chip bus communication models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 26, 1676–87.

GLOVER, F. W. et LAGUNA, M. (1997). *Tabu Search*. Kluwer Academic Publishers, London.

GOLDBERG, D. (1989). *Genetic Algorithms in Search and Optimization*. Addison-Wesley, Boston, MA.

GONZALEZ, R. (2000). Xtensa : a configurable and extensible processor. *IEEE Micro*, 20, 60–70.

GOODWIN, D. et PETKOV, D. (2003). Automatic generation of application specific processors. *CASES 2003 : International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM Press, San Jose, CA, United States, 137–147.

GRAHAM, S. L., KESSLER, P. B. et MCKUSICK, M. K. (1982). Gprof : A call graph execution profiler. *SIGPLAN '82 : Proceedings of the 1982 SIGPLAN symposium on Compiler construction*. ACM, New York, NY, USA, 120–126.

GREENSOCS LTD (2008). Karlsruhe SystemC parser suite. http ://www.greensocs.com/en/projects/KaSCPar.

GROTKER, T., LIAO, S., MARTIN, G. et SWAN, S. (2002). *System design with SystemC*. Kluwer Academic Publishers, Norwell, MA.

GU, R., JANNECK, J., RAULET, M. et BHATTACHARYYA, S. (2010). Exploiting statically schedulable regions in dataflow programs. *Journal of Signal Processing Systems*. http ://dx.doi.org/10.1007/s11265-009-0445-1.

HAMMING, R. (1950). Error detecting and error correcting codes. *Bell System Technical Journal*, 29, 147–160.

HAN, K.-H. et KIM, J.-H. (2002). Quantum-inspired evolutionary algorithm for a class of combinatorial optimization. *IEEE Transactions on Evolutionary Computation*, 6, 580–93.

HENDRICKS, K. et SINGHAL, V. (1997). Delays in new product introductions and the market value of the firm : the consequences of being late to the market. *Management Science*, 43, 422–36.

HENIA, R., HAMANN, A., JERSAK, M., RACU, R., RICHTER, K. et ERNST, R. (2005). System level performance analysis - the SymTA/S approach. *IEE Proceedings-Computers and Digital Techniques*, 152, 148–66.

HOGG, T. et HUBERMAN, B. A. (1985). Attractors on finite sets : the dissipative dynamics of computing structures. *Physical Review A (General Physics)*, 32, 2338–46.

HOUGH, R., KRISHNAMURTHY, P., CHAMBERLAIN, R. D., CYTRON, R. K., LOCKWOOD, J. et FRITTS, J. (2007). Empirical performance assessment using soft-core processors on reconfigurable hardware. *Proceedings of the 2007 Workshop on Experimental Computer Science*. San Diego, CA, United states, 12 pp.

HUANG, K., GRUNERT, D. et THIELE, L. (2007). Windowed FIFOs for FPGA-based multiprocessor systems. *IEEE 18th International Conference on Application-specific Systems, Architectures and Processors*. Montreal, Canada, 36–42.

HUBERT, H. et STABERNACK, B. (2009). Profiling-based hardware/software co-exploration for the design of video coding architectures. *IEEE Transactions on Circuits and Systems for Video Technology*, 19, 1680–91.

HUFFMAN, D. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 1098–1101.

HWANG, Y., ABDI, S. et GAJSKI, D. (2008). Cycle-approximate retargetable performance estimation at the transaction level. *DATE '08 : Proceedings of the conference on Design, automation and test in Europe*. ACM, New York, NY, USA, 3–8.

HWANG, Y., SCHIRNER, G. et ABDI, S. (2009). Automatic generation of cycle-approximate TLMs with timed RTOS model support. *Third IFIP TC 10 International Embedded Systems Symposium, IESS 2009*. Langenargen, Germany, 66–76.

IBM CORP. (1999). The CoreConnect bus architecture. Rapport technique, IBM Corp.

IBM MICROELECTRONICS DIVISION (1998). The PowerPC 405 core. Rapport technique, IBM Corp.

INFINEON TECHNOLOGIES AG (2000). TC10GP unified 32-bit microcontroller-DSP - user's manual. Rapport technique, Infineon Technologies AG.

INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS (2004). IEEE Std 802.16-2004 : IEEE standard for local and metropolitan area networks part 16 : Air interface for fixed broadband wireless access systems.

INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS (2005). IEEE Std 1666-2005 : IEEE standard SystemC language reference manual.

INTEL CORP. (2002). i960 Processors. http ://www.intel.com/design/i960/.

INTERNATIONAL BUSINESS STRATEGIES INC. (2004). Analysis of the relationship between EDA expenditures and competitive positioning of IC vendors for 2003. http ://www.edac.org/downloads/04_05_28_IBS_Report.pdf.

ISSHIKI, T., LI, D., KUNIEDA, H., ISOMURA, T. et SATOU, K. (2009). Trace-driven workload simulation method for multiprocessor system-on-chips. *2009 46th ACM/IEEE Design Automation Conference (DAC)*. Piscataway, NJ, USA, 232–7.

JADDOE, S., THOMPSON, M. et PIMENTEL, A. D. (2009). Signature-based calibration of analytical performance models for system-level design space exploration. *Transactions on High-Performance Embedded Architectures and Compilers*, 4, 18 pp.

JENG, J.-H. et TRUONG, T.-K. (1999). On decoding of both errors and erasures of a Reed-Solomon code using an inverse-free Berlekamp-Massey algorithm. *IEEE Transactions on Communications*, 47, 1488–1494.

JIGANG, W., CHANG, B. et SRIKANTHAN, T. (2009). A hybrid branch-and-bound strategy for hardware/software partitioning. *Proceedings of the 2009 8th IEEE/ACIS International Conference on Computer and Information Science, ICIS 2009*. Shanghai, China, 641–644.

JIGANG, W., SRIKANTHAN, T. et CHEN, G. (2010). Algorithmic aspects of hardware/software partitioning : 1D search algorithms. *IEEE Transactions on Computers*, 59, 532–544.

JIGANG, W., SRIKANTHAN, T. et JIAO, T. (2008). Algorithmic aspects for functional partitioning and scheduling in hardware/software co-design. *Design Automation for Embedded Systems*, 12, 345–375.

JUNG, H., YANG, H. et HA, S. (2008). Optimized RTL code generation from coarse-grain dataflow specification for fast HW/SW cosynthesis. *Journal of Signal Processing Systems*, 52, 13 – 34.

KAHN, G. (1974). The semantics of a simple language for parallel programming. *IFIP-74*. North-Holland, Proceedings of IFIP Congress 74, 471–75.

KALAVADE, A. et LEE, E. (1997). The extended partitioning problem : hardware/software mapping, scheduling, and implementation-bin selection. *Design Automation for Embedded Systems*, 2, 125–63.

KANGAS, T., KUKKALA, P., ORSILA, H., SALMINEN, E., HANNIKAINEN, M., HAMALAINEN, T. D., RIIHIMAKI, J. et KUUSILINNA, K. (2006). UML-based multiprocessor SoC design framework. *ACM Transactions on Embedded Computing Systems*, 5, 281–320.

KEINERT, J., STREUBUHR, M., SCHLICHTER, T., FALK, J., GLADIGAU, J., HAUBELT, C., TEICH, J. et MEREDITH, M. (2009). SystemCoDesigner : an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. *ACM Transactions on Design Automation of Electronic Systems*, 14, 23 pp.

KENNEDY, J. et EBERHART, R. (1995). Particle swarm optimization. *1995 IEEE International Conference on Neural Networks Proceedings*. New York, NY, USA, vol. 4, 1942–8.

KERNIGHAN, B. et LIN, S. (1970). An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49, 291–308.

KEUTZER, K., NEWTON, A. R., RABAEY, J. M. et SANGIOVANNI-VINCENTELLI, A. (2000). System-level design : orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19, 1523–43.

KHOT, S. (2006). Ruling out PTAS for graph min-bisection, dense k-subgraph, and bipartite clique. *SIAM Journal on Computing*, 36, 1025–1071.

KIM, S. et HA, S. (2006). Efficient exploration of bus-based system-on-chip architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14, 681–92.

KITCHENHAM, B. (2007). Empirical paradigm - the role of experiments [software engineering]. *Empirical Software Engineering Issues. Critical Assessment and Future Directions. International Workshop. Revised Papers (Lecture Notes in Computer Science Vol.4336).* Berlin, Germany, 25–32.

KLINGAUF, W. et GUNZEL, R. (2005). From TLM to FPGA : Rapid prototyping with SystemC and transaction level modeling. *IEEE International Conference on Field-Programmable Technology.* 285–86.

KNERR, B., HOLZER, M. et RUPP, M. (2008). RRES : a novel approach to the partitioning problem for a typical subset of system graphs. *EURASIP Journal on Embedded Systems*, 2008, 1–13.

KOVAC, J., PEER, P. et SOLINA, F. (2003). Human skin color clustering for face detection. *IEEE Region 8 EUROCON 2003. Computer as a Tool. Proceedings.* Piscataway, NJ, USA, vol. 2, 144–8.

KRASNER, J. (2003). Embedded software development issues and challenges. Rapport technique, Embedded Market Forecasters. http ://www.embeddedforecast.com/emf_esdi&c.pdf.

KWOK, Y.-K. et AHMAD, I. (1999). Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31, 406–471.

KWON, S., LEE, C., KIM, S., YI, Y. et HA, S. (2004). Fast design space exploration framework with an efficient performance estimation technique. *Proceedings of the 2004 2nd Workshop on Embedded Systems for Real-Time Multimedia.* Piscataway, NJ, USA, 27–32.

LABROSSE, J. J. (2002). *MicroC/OS II : The Real Time Kernel.* CMP Books, Gilroy, CA, seconde édition.

LAHIRI, K., RAGHUNATHAN, A. et DEY, S. (2001). System-level performance analysis for designing on-chip communication architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20, 768–83.

LAMPORT, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21, 558–65.

LATTNER, C. et ADVE, V. (2004). LLVM : a compilation framework for lifelong program analysis transformation. *International Symposium on Code Generation and Optimization.* Los Alamitos, CA, USA, 75–86.

LE BEUX, S., NICOLESCU, G., BOIS, G., BOUCHEBABA, Y., LANGEVIN, M. et PAULIN, P. (2009). Optimizing configuration and application mapping for MPSoC architectures. *Proceedings - 2009 NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2009.* San Francisco, CA, United states, 474–481.

LEE, C., KIM, S. et HA, S. (2010). A systematic design space exploration of MPSoC based on synchronous data flow specification. *Journal of Signal Processing Systems*, 58, 193–213.

LEE, E. A. (1999). Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7, 25–45.

LEE, E. A. (2006). The problem with threads. *Computer*, 29, 33–42.

LEE, E. A. et MESSERSCHMITT, D. G. (1987). Synchronous data flow. *Proceedings of the IEEE*, 75, 1235–45.

LEE, E. A. et SANGIOVANNI-VINCENTELLI, A. (1998). A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17, 1217–29.

LEE, G. G., CHEN, Y.-K., MATTAVELLI, M. et JANG, E. (2009). Algorithm/architecture co-exploration of visual computing on emergent platforms : overview and future prospects. *IEEE Transactions on Circuits and Systems for Video Technology*, 19, 1576–87.

LI, Y.-T. et MALIK, S. (1997). Performance analysis of embedded software using implicit path enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16, 1477–87.

LIU, F., GUO, F., SOLIHIN, Y., KIM, S. et EKER, A. (2008). Characterizing and modeling the behavior of context switch misses. *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*. Toronto, ON, Canada, 91–101.

LIU, X. et LEE, E. A. (2008). CPO semantics of timed interactive actor networks. *Theoretical Computer Science*, 409, 110–125.

LLOYD, S. (2002). Computational capacity of the universe. *Physical Review Letters*, 88, 237901/1–4.

LOEFFER, C., LIGTENBERG, A. et MOSCHYTZ, G. S. (1989). Practical fast 1-D DCT algorithms with 11 multiplications. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*. Glasgow, Scotland, vol. 2, 988–991.

LOPEZ-VALLEJO, M., GRAJAL, J. et LOPEZ, J. C. (2000). Constraint-driven system partitioning. *Proceedings of Meeting on Design Automation and Test in Europe*. IEEE Comput. Soc, Paris, France, 411–16.

LYNCH, N. A. et STARK, E. W. (1989). A proof of the Kahn principle for input/output automata. *Information and Computation*, 82, 81–92.

MADSEN, J., STIDSEN, T. K., KJAERULF, P. et MAHADEVAN, S. (2006). Multi-objective design space exploration of embedded system platforms. B. Kleinjohann, L. Kleinjohann, R. J. Machado, C. Pereira et P. S. Thiagarajan, éditeurs, *From Model-Driven Design*

to Resource Management for Distributed Embedded Systems, Springer, Boston, MA, vol. 225 de *IFIP International Federation for Information Processing*. 185–194.

MANN, Z. A. (2004). *Partitioning algorithms for hardware/software co-design.* Thèse de doctorat, Budapest University of Technology and Economics.

MANN, Z. A., ORBAN, A. et ARATO, P. (2007a). Finding optimal hardware/software partitions. *Formal Methods in System Design*, 31, 241–263.

MANN, Z. A., ORBAN, A. et FARKAS, V. (2007b). Evaluating the Kernighan-Lin heuristic for hardware/software partitioning. *International Journal of Applied Mathematics and Computer Science*, 17, 249–267.

MASSEY, J. L. (1969). Shift-register synthesis and BCH decoding. *IEEE Transactions on Information Theory*, IT-15, 122–7.

MCDOWELL, C. et HELMBOLD, D. (1989). Debugging concurrent programs. *Computing Surveys*, 21, 593–622.

MENTOR GRAPHICS CORP. (2005). Seamless hardware/software integration environment. http ://www.mentor.com/seamless.

MENTOR GRAPHICS CORP. (2008). ModelSim : Advanced simulation and debug. http ://www.mentor.com/modelsim.

MENTOR GRAPHICS CORP. (2009). Algorithmic C data types. http ://www.mentor.com/products/esl/high_level_synthesis/ac_datatypes.

MEREDITH, M. (2008). High-level SystemC synthesis with Forte's Cynthesizer. P. Coussy et A. Morawiec, éditeurs, *High-Level Synthesis : From Algorithm to Digital Circuit*, Springer, New York, NY. 75–97.

METROPOLIS, N., ROSENBLUTH, A., ROSENBLUTH, M., TELLER, A. et TELLER, E. (1953). Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21, 1087–1092.

MIANO, J. (1999). *Compressed image file formats : JPEG, PNG, GIF, XBM, BMP.* ACM Press/Addison-Wesley Publishing Co., New York, NY.

MIGLIORINI, C. (2008). *Exploration architecturale de communications-sur-puce au niveau système.* Mémoire de maîtrise, École Polytechnique de Montréal.

MOGUL, J. et BORG, A. (1991). The effect of context switches on cache performance. *SIGPLAN Notices*, 26, 75–84.

MOSS, L. et BOIS, G. (2009). On the Scott-continuity of tagged signal processes. Rapport technique EPM-RT-2009-01, École Polytechnique de Montréal.

MOSS, L., CANTIN, M.-A., BOIS, G. et ABOULHAMID, M. (2008). Automation of communication refinement and hardware synthesis within a system-level design methodology. *2008 19th IEEE/IFIP International Symposium on Rapid System Prototyping*. San Jose, CA, 75–81.

MOSS, L., DE NANCLAS, M., FILION, L., FONTAINE, S., BOIS, G. et ABOULHAMID, M. (2007). Seamless hardware/software performance co-monitoring in a codesign simulation environment with RTOS support. *10th Design, Automation and Test in Europe Conference and Exhibition*. IEEE, Nice, France, 876–881.

MULLER, H. (1993). *Simulating Computer Architectures*. Thèse de doctorat, University of Amsterdam.

NACUL, A. et GIVARGIS, T. (2006). Synthesis of time-constrained multitasking embedded software. *ACM Transactions on Design Automation of Electronic Systems*, 11, 822–847.

NEMHAUSER, G. L. et WOLSEY, L. A. (1988). *Integer and combinatorial optimization*. Wiley-Interscience, New York, NY, USA.

NIEMANN, R. et MARWEDEL, P. (1997). An algorithm for hardware/software partitioning using mixed integer linear programming. *Design Automation for Embedded Systems*, 2, 165–93.

NIKOLOV, H., STEFANOV, T. et DEPRETTERE, E. (2008a). Automated integration of dedicated hardwired IP cores in heterogeneous MPSoCs designed with ESPAM. *EURASIP Journal on Embedded Systems*, 2008, 15 pp.

NIKOLOV, H., STEFANOV, T. et DEPRETTERE, E. (2008b). Systematic and automated multiprocessor system design, programming, and implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 27, 542–55.

NIKOLOV, H., THOMPSON, M., STEFANOV, T., PIMENTEL, A., POLSTRA, S., BOSE, R., ZISSULESCU, C. et DEPRETTERE, E. (2008c). Daedalus : toward composable multimedia MP-SoC design. *2008 45th ACM/IEEE Design Automation Conference*. Piscataway, NJ, USA, 574–9.

OH, H. et HA, S. (2002). Hardware-software cosynthesis of multi-mode multi-task embedded systems with real-time constraints. *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002*. New York, NY, USA, 133–8.

ORSILA, H., KANGAS, T. et HAMALAINEN, T. (2005). Hybrid algorithm for mapping static task graphs on multiprocessor SoCs. *Proceedings 2005 International Symposium on System-on-Chip*. Piscataway, NJ, USA, 146–150.

ORSILA, H., KANGAS, T., SALMINEN, E., HAMALAINEN, T. et HANNIKAINEN, M. (2007). Automated memory-aware application distribution for multi-processor system-on-chips. *Journal of Systems Architecture*, 53, 795–815.

ORTNER, M. (2004). *Processus Ponctuels Marqués pour l'Extraction Automatique de Caricatures de Bâtiments à partir de Modèles Numériques d'Elévation.* Thèse de doctorat, Université de Nice-Sophia Antipolis.

PAGE, B. et KREUTZER, W. (2005). *The Java Simulation Handbook : Simulating Discrete Event Systems with UML and Java.* Shaker Verlag, Aachen, Allemagne.

PANANGADEN, P. et SHANBHOGUE, V. (1992). The expressive power of indeterminate dataflow primitives. *Information and Computation*, 98, 99–131.

PANANGADEN, P. et STARK, E. W. (1988). Computations, residuals, and the power of indeterminacy. T. Lepisto et A. Salomaa, éditeurs, *Proceedings of the 15th International Colloquium on Automata, Languages and Programming. Lecture Notes in Computer Science*, Springer-Verlag, London, vol. 317. 439–454.

PAPADIMITRIOU, C. H. et YANNAKAKIS, M. (1990). Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19, 322–328.

PARKS, T. M. (1995). *Bounded Scheduling of Process Networks.* Thèse de doctorat, University of California, Berkeley.

PEER, P. et SOLINA, F. (1999). An automatic human face detection method. N. Brandle, éditeur, *Proceedings of Computer Vision Winter Workshop*. 122–130.

PENNEBAKER, W. B. et MITCHELL, J. L. (1993). *JPEG still image data compression standard.* Van Nostrand Reinhold, New York, NY, troisième édition.

PENSON, K. A., BLASIAK, P., DUCHAMP, G., HORZELA, A. et SOLOMON, A. I. (2004). Hierarchical Dobinski-type relations via substitution and the moment problem. *Journal of Physics A (Mathematical and General)*, 37, 3475–87.

PERRIN, G., DESCOMBES, X. et ZERUBIA, J. (2005). Adaptive simulated annealing for energy minimization problem in a marked point process application. *Energy Minimization Methods in Computer Vision and Pattern Recognition. 5th International Workshop, EMMCVPR 2005. Proceedings (Lecture Notes in Computer Science Vol.3757)*. Berlin, Germany, 3–17.

PIMENTEL, A., ERBAS, C. et POLSTRA, S. (2006). A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*, 55, 99–112.

PIMENTEL, A. D., THOMPSON, M., POLSTRA, S. et ERBAS, C. (2008). Calibration of abstract performance models for system-level design space exploration. *Journal of Signal Processing Systems*, 50, 99–114.

POSADAS, H., HERRERA, F., SANCHEZ, P., VILLAR, E. et BLASCO, F. (2004). System-level performance analysis in SystemC. *Proceedings. Design, Automation and Test in Europe Conference and Exhibition*. IEEE Comput. Soc, Paris, France, vol. 1, 378–83.

PRAKASH, S. et PARKER, A. (1992). SOS : synthesis of application-specific heterogeneous multiprocessor systems. *Journal of Parallel and Distributed Computing*, 16, 338–51.

PRAKASH, S. et PARKER, A. (1994). Synthesis of application-specific multiprocessor systems including memory components. *Journal of VLSI Signal Processing*, 8, 97–116.

RAE, A. et PARAMESWARAN, S. (1998). Application-specific heterogeneous multiprocessor synthesis using differential-evolution. *Proceedings. 11th International Symposium on System Synthesis*. Los Alamitos, CA, USA, 83–8.

REED, I. S. et SOLOMON, G. (1960). Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8, 300–304.

RESANO, J. J., PEREZ, M. E., MOZOS, D., MECHA, H. et SEPTIEN, J. (2003). Analyzing communication overheads during hardware/software partitioning. *Microelectronics Journal*, 34, 1001–7.

RIIHIMAKI, J., SALMINEN, E., KUUSILINNA, K. et HAMALAINEN, T. (2002). Parameter optimization tool for enhancing on-chip network performance. *2002 IEEE International Symposium on Circuits and Systems*. Piscataway, NJ, USA, vol. 4, 61–4.

RIORDAN, J. (1958). *An Introduction to Combinatorial Analysis*. John Wiley and Sons, Inc., New York, NY.

ROGERS-VALLÉE, M., CANTIN, M.-A. et BOIS, G. (2010). IP characterization methodology for fast and accurate power consumption estimation at transactional level model. Soumis à 6th International Conference on Hardware-Software Codesign and System Synthesis.

ROSINGER, H.-P. (2004). Connecting customized IP to the MicroBlaze soft processor using the Fast Simplex Link (FSL) channel. Rapport technique, Xilinx, Inc.

ROWE, K. (2010). Time to market is a critical consideration. http ://www.embedded.com/columns/guest/223100896.

SAAD, E.-S., AWADALLA, M. et EL-DEEN, K. (2009). FPGA based software profiler for hardware-software codesign. *26th National Radio Science Conference (NRSC 2009)*. Piscataway, NJ, USA, 8 pp.

SARKAR, S., DABRAL, S., TIWARI, P. et MITRA, R. (2009). Lessons and experiences with high level synthesis. *IEEE Design & Test of Computers*, 26, 34–45.

SCHLICHTER, T., LUKASIEWYCZ, M., HAUBELT, C. et TEICH, J. (2006). Improving system level design space exploration by incorporating SAT-solvers into multi-objective evolutionary algorithms. *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*. Piscataway, NJ, USA, 6 pp.

SCHNERR, J., BRINGMANN, O., VIEHL, A. et ROSENSTIEL, W. (2008). High-performance timing simulation of embedded software. *2008 45th ACM/IEEE Design Automation Conference*. IEEE, Anaheim, CA, 290–5.

SHANKAR, K. et LYSECKY, R. (2009). Non-intrusive dynamic application profiling for multitasked applications. *2009 46th ACM/IEEE Design Automation Conference (DAC)*. Piscataway, NJ, USA, 130–5.

SHANNON, L. et CHOW, P. (2004). Using reconfigurability to achieve real-time profiling for hardware/software codesign. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays - FPGA*. Monterey, CA., United states, vol. 12, 190–199.

SHIN, D., GERSTLAUER, A., DOMER, R. et GAJSKI, D. (2008). An interactive design environment for C-based high-level synthesis of RTL processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16, 466–75.

SINHA, B. P. et BHATTACHARYA, B. B. (1985). On the numerical complexity of short-circuit faults in logic networks. *IEEE Transactions on Computers*, C-34, 186–190.

SINNEN, O. et SOUSA, L. A. (2004). On task scheduling accuracy : Evaluation methodology and results. *Journal of Supercomputing*, 27, 177–194.

SINNEN, O., SOUSA, L. A. et SANDNES, F. E. (2006). Toward a realistic task scheduling model. *IEEE Transactions on Parallel and Distributed Systems*, 17, 263–275.

SLOANE, N. J. A. (2010). The on-line encyclopedia of integer sequences. http ://www.research.att.com/ njas/sequences/.

SLOMKA, F., ALBERS, K. et HOFMANN, R. (2004). A multiobjective tabu search algorithm for the design space exploration of embedded systems. *IFIP 18th World Computer Congress, TC10 Working Conference on Distributed and Parallel Embedded Systems (DIPES 2004)*. 227–236.

SONDERGAARD, H. et SESTOFT, P. (1992). Non-determinism in functional languages. *Computer Journal*, 35, 514–23.

SPACE CODESIGN SYSTEMS INC. (2008). Space Codesign GenX user's guide. Rapport technique SCO-USR-005 v1.4.R01, Space Codesign Systems Inc.

SPRUNT, B. (2002). The basics of performance-monitoring hardware. *IEEE Micro*, 22, 64–71.

SRINIVASAN, R. (1995). RFC 1832 : XDR : External Data Representation standard. Rapport technique, Internet Engineering Task Force. RFC1832.

STALLMAN, R. M., PESCH, R. H. et SHEBS, S. (2002). *Debugging with GDB : The GNU Source-level Debugger*. Free Software Foundation, Boston, MA, 9th édition.

STONE, H. S. (1977). Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, SE-3, 85–93.

STREHL, K., THIELE, L., GRIES, M., ZIEGENBEIN, D., ERNST, R. et TEICH, J. (2001). FunState-an internal design representation for codesign. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9, 524–44.

STREUBUHR, M., HAUBELT, C. et TEICH, J. (2009). System level performance simulation for heterogenous multi-processor architectures. *1st HiPEAC Workshop on Rapid Simulation and Performance Evaluation : Methods and Tools, in conjunction with the 4th HiPEAC Conference*. Paphos, Cyprus.

SUN, F., RAVI, S., RAGHUNATHAN, A. et JHA, N. (2003). A scalable application-specific processor synthesis methodology. *ICCAD-2003. International Conference on Computer Aided Design*. Piscataway, NJ, USA, 283–90.

TEXAS INSTRUMENTS INC. (2010). Digital Signal Processors. http ://www.ti.com/home_p_dsp.

TIBBOEL, W., REYES, V., KLOMPSTRA, M. et ALDERS, D. (2007). System-level design flow based on a functional reference for HW and SW. *44th ACM/IEEE Design Automation Conference*. IEEE, San Diego, CA, 23–28.

TIWARI, V., MALIK, S., WOLFE, A. et LEE, M.-C. (1996). Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, 13, 223–38.

UEDA, K., SAKANUSHI, K., TAKEUCHI, Y. et IMAI, M. (2005). Architecture-level performance estimation method based on system-level profiling. *IEE Proceedings : Computers and Digital Techniques*, 152, 12–19.

VALLERIO, K. et JHA, N. (2003). Task graph extraction for embedded system synthesis. *Proceedings 16th International Conference on VLSI Design*. Los Alamitos, CA, USA, 480–6.

VAN DER WOLF, P., DE KOCK, E., HENRIKSSON, T., KRUIJTZER, W. et ESSINK, G. (2004). Design and programming of embedded multiprocessors : an interface-centric approach. *International Conference on Hardware/Software Codesign and Systems Synthesis*. ACM, Stockholm, Sweden, 206–17.

VAN DIJK, H. W., SIPS, H. J. et DEPRETTERE, E. F. (2003). Context-aware process networks. *IEEE International Conference on Application-Specific Systems, Architectures, and Processors*. IEEE Comput. Soc, The Hague, Netherlands, 6–16.

VAN LAARHOVEN, P. et AARTS, E. (1987). *Simulated Annealing : Theory and Applications*. D. Reidel, Boston, MA.

VAST SYSTEMS TECHNOLOGY CORP. (2008). Metrix : Monitoring and measuring embedded systems development. http ://www.vastsystems.com/solutions-visualization.html.

VITERBI, A. J. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, IT-13, 260–269.

VOLCKMANN, M., BALACCO, S. et ROMMEL, C. (2008). Virtual system prototyping/ simulation tools for software development and verification. *2008 Embedded Software Market Intelligence Service*, VDC Research Group, Natick, MA.

WANG, G., GONG, W. et KASTNER, R. (2003). A new approach for task level computational resource bi-partitioning. *Proceedings of the Fifteenth IASTED International Conference on Parallel and Distributed Computing and Systems*. Anaheim, CA, USA, vol. 1, 439–44.

WANG, G., GONG, W. et KASTNER, R. (2006). Application partitioning on programmable platforms using the ant colony optimization. *Journal of Embedded Computing*, 2, 119–136.

WEISBERG, S. (2005). *Applied linear regression*. Wiley-Interscience, Hoboken, NJ, troisième édition.

WENANDE, S. et CHIDESTER, R. (2001). Xilinx takes power analysis to new levels with XPower. *Xcell Journal*, 41, 26–27.

WIANGTONG, T., CHEUNG, P. Y. K. et LUK, W. (2002). Comparing three heuristic search methods for functional partitioning in hardware-software codesign. *Design Automation for Embedded Systems*, 6, 425–49.

WILD, T., FOAG, J., PAZOS, N. et BRUNNBAUER, W. (2003). Mapping and scheduling for architecture exploration of networking SoCs. *Proceedings 16th International Conference on VLSI Design concurrently with the 2nd International Conference on Embedded Systems Design*. Los Alamitos, CA, USA, 376–81.

WILF, H. S. (1994). *Generatingfunctionology*. Academic Press, San Diego, CA, seconde édition.

WOHLIN, C., RUNESON, P. et HOST, M. (2000). *Experimentation in software engineering : an introduction*. Kluwer Academic Publishers, Norwell, MA.

WOLF, F. et ERNST, R. (2001). Execution cost interval refinement in static software analysis. *Journal of Systems Architecture*, 47, 339–56.

WOLF, W. (1997). An architectural co-synthesis algorithm for distributed, embedded computing systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5, 218–29.

WOLF, W., JERRAYA, A. et MARTIN, G. (2008). Multiprocessor system-on-chip (MPSoC) technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27, 1701–13.

XILINX INC. (2002). *Virtex-II Pro Platform FPGA Handbook*. Xilinx Inc., San Jose, CA.

XILINX INC. (2003). CORE Generator guide. Rapport technique, Xilinx Inc.

XILINX INC. (2005). MicroBlaze processor reference guide. Rapport technique, Xilinx Inc.

XILINX INC. (2006). XST user guide. Rapport technique, Xilinx Inc.

YATES, R. K. (1993). Networks of real-time processes. *Proceedings of the 4th International Conference on Concurrency Theory*. Springer-Verlag, Hildesheim, Germany, 384–97.

YOO, J.-H., FENG, X., CHOI, K., YOUNG CHUNG, E. et CHOI, K.-M. (2006). Worst case execution time analysis for synthesized hardware. *Proceedings of the ASP-DAC 2006. Asia and South Pacific Design Automation Conference 2006*. Piscataway, NJ, USA, 905–910.

ZHANG, Z., FAN, Y., JIANG, W., HAN, G., YANG, C. et CONG, J. (2008). AutoPilot : A platform-based ESL synthesis system. P. Coussy et A. Morawiec, éditeurs, *High-Level Synthesis : From Algorithm to Digital Circuit*, Springer, New York, NY. 99–112.

ZITZLER, E., THIELE, L., LAUMANNS, M., FONSECA, C. et DA FONSECA, V. (2003). Performance assessment of multiobjective optimizers : an analysis and review. *IEEE Transactions on Evolutionary Computation*, 7, 117–32.

ANNEXE A

TRANSFORMATION D'UN RTPN EN DEPN

Le chapitre 4 a présenté le modèle de calcul des réseaux de processus temps-réel (RTPN) et a défini un ordonnancement d'un RTPN comme une fonction qui associe un temps donné à chaque évènement du RTPN. On indique ici comment un RTPN dont on a fixé l'ordonnancement peut être modélisé comme un réseau de processus à évènements discrets (DEPN). De manière équivalente, un DEPN qui représente une implémentation pleinement temporisée d'un RTPN peut fixer l'ordonnancement de celui-ci.

Les DEPN sont modélisés avec le méta-modèle formel de calcul appelé *tagged signal model* (Lee et Sangiovanni-Vincentelli, 1998; Lee, 1999). Ainsi, dans ce méta-modèle, on a un ensemble $T = \mathbb{R}$ qui représente une dimension temporelle, un ensemble de valeurs V et un évènement (jeton temporisé) e est un élément de l'ensemble $T \times V$: un évènement se produit à un instant donné avec une valeur donnée. Un signal à évènement discrets s est une suite d'évènements $((t_1, v_1), (t_2, v_2), \dots)$ qui respecte un ordre chronologique : on a donc que $i < j$ implique $t_i < t_j$ pour tout i, j. Un signal est donc une séquence de jetons temporisés. On désigne par S par l'ensemble des signaux à évènements discrets. Pour tout m, n, une fonction $f : S^m \to S^n$ est donc déterministe : si on connait le vecteur de signaux $\vec{x}$ fourni en entrée à f, alors on connait également le vecteur de signaux $\vec{y} = f(\vec{x})$ produit par f. Il est à noter que ce déterminisme porte non seulement sur la fonctionnalité (valeurs des jetons temporisés) mais également sur la performance (temps des évènements). Cette modélisation ne permet donc pas de faire varier le temps d'exécution des processus et c'est pourquoi elle n'a pas été retenue pour notre méthode d'exploration architecturale.

Un DEPN est déterministe si il est possible déterminer de manière unique son comportement, autant en terme de fonctionnalité que de performance, à partir des processus qui le composent et des connexions entre ceux-ci. Pour trouver dans quelles conditions un DEPN est déterministe, il faut faire appel au concept de causalité et au théorème Banach du point fixe (Yates, 1993; Lee, 1999). Ainsi, une fonction f est causale si, pour toute paire de signaux r et s identiques jusqu'au temps t, on a $f(r)$ et $f(s)$ identiques au moins jusqu'au temps t. En d'autres termes, l'entrée de la fonction jusqu'au temps t détermine la sortie au moins jusqu'au temps t. Une fonction f est delta-causale si il existe un $\Delta > 0$ tel que, pour toute paire de signaux r et s identiques jusqu'au temps t, on a $f(r)$ et $f(s)$ identiques au moins jusqu'au temps $t + \Delta$. C'est une autre manière de dire qu'une fonction delta-causale introduit un délai d'au moins Δ entre son entrée et sa sortie. Si chaque processus d'un DEPN est

delta-causal et déterministe, alors le comportement du DEPN existe et est unique : il est donc déterministe (Yates, 1993; Lee, 1999). De plus, il est possible de trouver la valeur des signaux du DEPN en les initialisant à une valeur initiale quelconque (par exemple le signal vide), puis en appliquant itérativement les fonctions correspondant aux processus DEPN. En d'autres termes, on peut trouver le comportement unique d'un tel DEPN en l'exécutant ou en le simulant. Cela signifie que la sémantique opérationnelle d'un tel DEPN (la manière dont il est exécuté) correspond à sa sémantique dénotationnelle (l'application qu'il modélise).

On montre ici comment un RTPN dont a fixé l'ordonnancement peut être modélisé en un DEPN et quelles conditions doit respecter un RTPN pour que les DEPN générés à partir de celui-ci soient déterministes.

A.1 Modélisation des canaux du RTPN

Tel qu'illustré à la figure A.1, un canal déterministe d'un RTPN peut être modélisé par un processus DEPN qui est fonction $f : S^3 \to S$ qui a comme entrées un signal x de données envoyées, un signal r de requêtes de lecture, un signal a d'arbitrage et comme sortie un signal y de données reçues. Les évènements sur les signaux x et r correspondent respectivement aux évènements w_e et r_b du modèle RTPN : une donnée arrive sur le signal x lorsque le processus source a terminé de l'écrire et un évènement arrive sur le signal r lorsque le processus destinataire demande de lire une donnée. Le canal peut alors transmettre une donnée au processus destinataire via le signal y si une telle donnée est disponible (évènement a du RTPN). Sinon, son comportement dépendra du type de canal : un canal avec lecture bloquante attendra une nouvelle donnée en x pour ensuite la transmettre via y au processus destinataire, un canal avec lecture non-bloquante transmettra plutôt une valeur absente sur le canal y. Les évènements du signal y correspondent aux évènements r_e du RTPN : la lecture se termine lorsque le canal donne sa réponse.

Le signal a d'un canal modélise le fait que les canaux d'un système ne sont pas nécessairement indépendants les uns des autres et que leur comportement et leur performance peut varier selon l'utilisation des autres canaux, par exemple si plusieurs canaux se partagent un même bus. Cet arbitre peut également servir à s'assurer que le DEPN impose l'ordonnancement fixé pour le RTPN. L'ajout d'un tel arbitre au modèle DEPN s'effectue sans perte de généralité, puisque rien n'exclut qu'il permette à plusieurs canaux de communiquer en parallèle. Le signal a représente donc des commandes pouvant être fournies par un arbitre. Si un RTPN contient n canaux, l'arbitre est modélisé comme un processus DEPN qui est une fonction $f : S^{2n} \to S^n$ qui prend en entrée deux vecteurs $\vec{x}$ et $\vec{r}$ regroupant les signaux x et r de chaque canal et qui produit en sortie un vecteur $\vec{a}$ regroupant les signaux a de chaque

canal. Un tel arbitre est donc également déterministe, bien que la modélisation exacte de sa fonction f puisse être complexe.

A.2 Modélisation des processus du RTPN

Un processus déterministe d'un RTPN avec m canaux d'entrées et n canaux de sorties devient dans le modèle DEPN une fonction $f : S^{m+1} \to S^{m+n}$. Cette fonction a comme entrées un signal b d'ordonnancement et un vecteur $\vec{y}$ de signaux de données regroupant les signaux de sortie y de chacun des m canaux d'entrées. Ses sorties sont un vecteur $\vec{r}$ regroupant les signaux de requête r des canaux d'entrée ainsi qu'un vecteur $\vec{z}$ regroupant les signaux d'entrées x de chacun des n canaux de sortie. Ainsi, un processus envoie des requêtes de lecture à ses canaux d'entrée via les signaux $\vec{r}$, reçoit les valeurs lues via les signaux $\vec{y}$ et produit des valeurs de sortie qu'il écrit à ses canaux de sortie via les signaux $\vec{z}$. Ainsi, un processus communique avec les autres processus seulement via les canaux de communications. La figure A.1 illustre un exemple de processus avec un canal d'entrée.

Le signal d'ordonnancement b fourni en entrée au processus sert à modéliser le cas où plusieurs processus se partagent un même processeur et où un ordonnanceur doit s'assurer qu'un seul de ces processus s'exécute à la fois. Un tel ordonnanceur peut également s'assurer que le DEPN impose l'ordonnancement fixé pour le RTPN. L'ajout d'un tel ordonnanceur au modèle DEPN s'effectue sans perte de généralité, puisque rien n'exclut qu'il permette à plusieurs processus de s'exécuter en parallèle. Cet ordonnanceur est modélisé comme une fonction qui prend en entrée trois matrices de signaux Y, R et Z qui regroupent les vecteurs de signaux $\vec{y}$, $\vec{r}$ et $\vec{z}$ associés à chacun des processus du système et produit en sortie un vecteur de signaux $\vec{b}$ regroupant les signaux b d'ordonnancement de chaque processus.

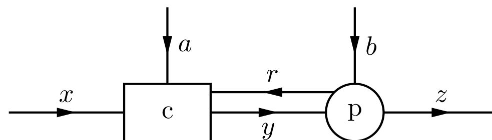

Figure A.1 Un processus p avec un canal c d'entrée, un signal a d'arbitrage, un signal b d'ordonnancement, un signal r de requête et des signaux x,y et z de données

A.3 Composition en un réseau de processus

À un RTPN donné dont a fixé l'ordonnancement, on associe un DEPN dont les processus sont les modélisations DEPN des processus RTPN, des canaux RTPN, de l'arbitre et de l'ordonnanceur présentées aux sections précédentes. Ces processus DEPN sont connectés

entre eux par l'ensemble de signaux illustré à la figure A.2. Ainsi, le RTPN a en entrée une matrice I de signaux qui regroupent l'ensemble des entrées externes du RTPN. Ces signaux I sont fournis en entrée au vecteur $\vec{c}$ des canaux du RTPN, à l'arbitre et à l'ordonnanceur. Les canaux, l'arbitre et l'ordonnanceur reçoivent également en entrée la matrice X de tous les signaux de données internes au RTPN produits par le vecteur $\vec{p}$ des processus du RTPN ainsi que la matrice R des signaux de requêtes de lecture de chaque processus. L'arbitre utilise ces informations pour produire un vecteur $\vec{a}$ de signaux d'arbitrage de chaque canal alors que les canaux produisent la matrice Y des signaux de données transmis aux processus et à l'ordonnanceur. L'ordonnanceur génère le vecteur $\vec{b}$ des signaux d'ordonnancement des processus. Les processus produisent une matrice O de signaux de sortie externes au RTPN et cette matrice est également fournie en entrée à l'ordonnanceur. La matrice formée Z formée par la concaténation des matrices X et O correspond à l'ensemble des signaux de sortie produits par les processus du système.

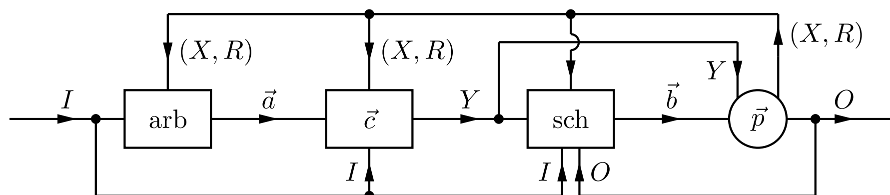

Figure A.2 Modèle général d'un RTPN en tant que DEPN

On cherche à savoir si, pour une matrice donnée de signaux d'entrée I, il est possible de déterminer la valeur de l'ensemble des autres signaux du DEPN. En d'autres termes, le DEPN est-il déterministe ? Il n'est pas suffisant que l'ensemble des éléments du DEPN soient des fonctions déterministes. En effet, prenons par exemple un DEPN tel que l'arbitre ne retarde jamais un canal, que l'ordonnanceur ne retarde jamais un processus et que les processus effectuent toujours une requête de lecture dès qu'une valeur est disponible dans un canal d'entrée. Supposons de plus que les canaux de ce DEPN sont tous tels que leur signal de sortie y est égale à leur signal d'entrée x et que les processus de ce DEPN sont tous tels que leurs signaux de sortie sont égaux à leur signal d'entrée. Si deux de ces processus et deux de ses canaux sont connectés en une boucle de rétroaction, alors on obtient que les signaux x_1, y_1, x_2, y_2 de cette boucle de rétroaction peuvent prendre n'importe quelle valeur tant que $x_1 = y_1 = x_2 = y_2$, tel qu'illustré à la figure A.3 (l'arbitre, l'ordonnanceur et les signaux de requête sont omis pour plus de clarté). Si on ajoute au DEPN une entrée i qui est ignorée, alors, pour chaque valeur donnée de i, le DEPN peut produire en sortie n'importe quelle valeur de o tant que $x_1 = y_1 = x_2 = y_2 = o$: ce DEPN est donc non-déterministe.

On sait qu'un DEPN est déterministe si tous ses processus sont déterministes et delta-

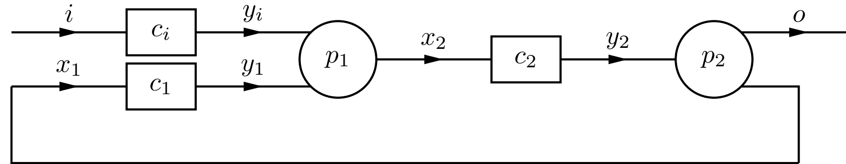

Figure A.3 Modèle d'un DEPN non déterministe avec $x_1 = y_1 = x_2 = y_2 = o$

causals. On montre de plus que ce théorème Banach du point fixe peut s'appliquer même si chaque élément du DEPN n'est pas delta-causal. On suppose d'abord que chaque élément du DEPN est causal et déterministe. Supposons également que chaque canal RTPN modélisé comme processus DEPN est delta-causal (cette hypothèse n'est pas respectée par le DEPN non-déterministe de la figure A.3). Chaque canal produit donc sa sortie y avec un délai d'au moins Δ par rapport à ses entrées x, r et a et le vecteur des canaux produit donc sa sortie Y avec un délai d'au moins Δ par rapport à ses entrées X, R et $\vec{a}$. L'arbitre, qui est causal, produit sa sortie $\vec{a}$ avec un délai supérieur ou égal à 0 par rapport à ses entrées X et R. Si on compose l'arbitre avec le vecteur de canaux $\vec{c}$, on obtient donc que l'élément résultant illustré à la figure A.4 produit sa sortie Y avec un délai d'au moins Δ par rapport à ses entrées I, X et R : cet élément composé est donc delta-causal. On démontre ensuite qu'il suffit de supposer que l'ordonnanceur est delta-causal pour que le DEPN soit déterministe.

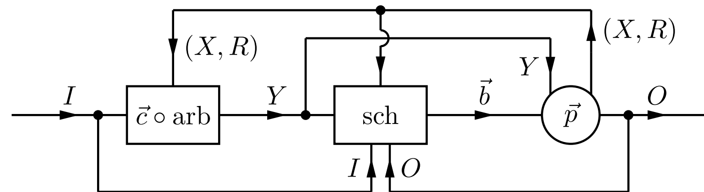

Figure A.4 Modèle du RTPN après composition de l'arbitre avec les canaux

L'hypothèse de delta-causalité de l'ordonnanceur est naturelle étant donné que, pour les processus représentant une tâche logicielle à ordonnancer sur un processeur, l'ordonnanceur est également exécuté en logiciel et a donc un temps de réaction non nul. Quant aux processus purement comportementaux ou représentant un composant matériel et n'ayant donc pas besoin d'être ordonnancés sur un processeur, l'ordonnanceur leur fournit une sortie constante indiquant que le processus peut toujours s'exécuter. La production d'un tel signal constant est indépendante des entrées de l'ordonnanceur et est donc trivialement delta-causale.

L'ordonnanceur produit sa sortie $\vec{b}$ avec un délai d'au moins Δ par rapport à ses entrées I, X, Y, R et O alors que le vecteur de processus $\vec{p}$ est causal et produit donc ses sorties X, R et O avec un délai d'au moins 0 par rapport à ses entrées Y et $\vec{b}$. Le délai d'au moins Δ sur la production de $\vec{b}$ par l'ordonnanceur se répercute alors sur la production de sorties de $\vec{p}$,

qui a $\vec{b}$ comme entrée. Il a été démontré dans (Lee, 1999) qu'une composition tel que celle de l'ordonnanceur et du vecteur de processus $\vec{p}$ est delta-causale : elle produit ses sorties X, R et O avec un délai d'au moins Δ par rapport à ses entrées I et Y. Après ces deux compositions, on obtient que le RTPN tel qu'illustré à la figure A.5 est composé de deux éléments qui sont chacun delta-causal, le RTPN est donc déterministe. Cela démontre qu'un DEPN généré à partir d'un RTPN est déterministe si l'ordonnanceur et chaque canal sont delta-causals et que l'arbitre et chaque processus sont causals.

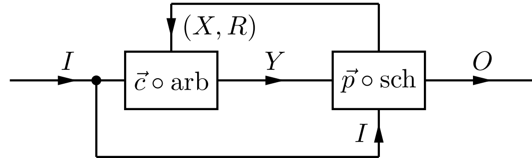

Figure A.5 Modèle du RTPN après composition de l'ordonnanceur avec les processus

Une preuve plus complexe se base sur la propriété de la Scott-continuité : si chaque processus d'un réseau de processus est Scott-continu, alors le réseau de processus est lui aussi Scott-continu et a un comportement déterministe. Cette propriété a été démontrée par (Liu et Lee, 2008) pour les DEPN. Il suffirait donc de démontrer que les canaux, l'arbitre et l'ordonnanceur d'un tel DEPN sont Scott-continus pour démontrer qu'un RTPN modélisé en DEPN est Scott-continu (et donc déterministe) si les processus qui le composent sont tous Scott-continus. On a ainsi démontré dans (Moss et Bois, 2009) que les modèles DEPN des canaux Kahn et POLL sont Scott-continus. L'avantage d'une preuve basée sur la Scott-continuité plutôt que sur la delta-causalité est que cette preuve couvre des systèmes continus ou des hybrides continu/discret qui ne sont pas nécessairement delta-causaux. Cependant, comme les systèmes embarqués étudiés dans cette thèse sont des systèmes informatiques numériques et donc discrets, la preuve basée sur la delta-causalité présentée ci-dessus suffit.

A.4 Simulation et exécution déterministe

Les résultats présentés dans cette section s'appliquent également à la simulation ou à l'exécution d'applications embarquées modélisées comme un ensemble de processus concurrents (tels des fils d'exécution SystemC (IEEE, 2005)) qui communiquent via des modèles transactionnels (Cai et Gajski, 2003) ou fonctionnels de bus ou de FIFO. En effet, de tels systèmes embarqués ont alors une structure similaire à celle d'un RTPN (et d'un DEPN généré à partir de celui-ci). Si les processus ont un comportement et une performance déterministe et que les communications entre processus ont lieu exclusivement au travers de canaux de communication tels que des bus et des FIFO, alors il suffit que chaque communication au travers

de ces canaux de communications ait un délai déterministe supérieur à 0 pour que le système dans son ensemble ait un comportement et une performance déterministe. En d'autres termes, à chaque fois qu'on exécute ce système embarqué avec un stimulus identique (à la fois au niveau des valeurs et des temps d'arrivée), ce système embarqué a un comportement et une performance identique.

Ce résultat s'applique directement aux modèles temporisés et la simulation sera alors déterministe dès lors que les modèles de bus modélisent approximativement ou précisément les délais de communication. Ce résultat ne s'applique théoriquement pas aux modèles non temporisés étant donné que ces modèles n'ont alors aucune notion de delta-causalité. Cependant, on peut en pratique l'appliquer à ces modèles en les transformant en des modèles très approximativement temporisés, dans lequel le délai induit par les canaux de communications est d'une valeur arbitraire supérieure à 0. Dans le cas d'un modèle SystemC, le canal de communication doit donc effectuer un `wait()` avec un délai supérieur à 0 lors d'une communication. Concrètement, cela signifie qu'un processus ne peut pas effectuer un nombre infini de lectures ou d'écritures non-bloquantes en un temps fini et qu'on évite ainsi une situation qui mènerait à la famine des autres processus lors de la simulation ou de l'exécution.

ANNEXE B

PROCÉDURE DE CARACTÉRISATION DU RTOS ET DE L'API LOGICIELLE

Cet annexe vient compléter la section 7.2.3.4 sur la caractérisation du RTOS et de l'API logicielle en décrivant leurs paramètres de performance et présentant une méthode automatisée pour l'extraction de paramètres de performance par le profilage d'une application synthétique qui exerce les différents cas d'utilisation du RTOS et de l'API logicielle.

B.1 Définition des paramètres de performance

On définit d'abord la politique d'ordonnancement des tâches ainsi que la largeur l_p du processeur, qui désigne le nombre d'octets avec lequel le processeur fonctionne et communique nativement. Le RTOS μC/OS-II (Labrosse, 2002) utilise une politique d'ordonnancement préemptif avec priorités statiques et héritage de priorité (on suppose qu'un ordre de priorité est déjà associé aux modules de la spécification exécutable). Le processeur MicroBlaze (Xilinx Inc., 2005) a une largeur de 4 octets.

Il y a ensuite le paramètre t_{ctx}, qui désigne le temps nécessaire à un changement de contexte. La principale routine de traitement d'interruption de l'API logicielle s'exécute lorsque le processeur reçoit un paquet d'un module externe au processeur (soit un module implémenté en matériel ou une tâche logicielle qui se trouve sur un autre processeur), cette routine copie alors le contenu du paquet dans une plage de mémoire locale à la tâche destinataire. Cette routine prend t_{isr0} puis t_{isr+} pour chaque tranche de l_p octets. De plus, si la réception de cette écriture débloque une tâche logicielle qui attendait cette écriture, la routine aura un délai supplémentaire de $t_{isrunblk}$. Cette routine prendra donc un temps de $t_{isr0} + \lceil n/l_p \rceil t_{isr+} + t_{isrunblk} b$ pour recevoir un paquet de n octets, où b est une variable booléenne qui indique si la tâche destinataire a dû être débloquée. L'autre routine de traitement d'interruption est celle qui traite l'acquittement d'une écriture bloquante vers un module externe au processeur, cette routine de traitement d'interruption prend un temps de t_{israck}.

L'envoi d'une requête en lecture ou en écriture à un périphérique de largeur l_d prend un temps t_{per0}, puis t_{per+} pour chaque tranche de $min(l_p, l_d)$ octets. Une requête de n octets à un périphérique demandera donc un temps de $t_{per0} + \lceil n/min(l_p, l_d) \rceil t_{per+}$.

L'écriture d'un paquet de données vers un module externe au processeur demande un temps de t_{hw0}, puis t_{hw+} pour chaque tranche de l_p octets. Il y a un délai supplémentaire de

t_{hwblk} si il s'agit d'une écriture bloquante et un autre délai supplémentaire de $t_{hwackblk}$ si la tâche logicielle doit s'interrompre parce que l'acquittement de l'écriture n'est pas arrivé au moment où la tâche vérifie s'il est arrivé. Ce délai exclut le temps pendant lequel la tâche est suspendue en attente de l'acquittement. Le temps nécessaire à l'écriture d'un paquet de n octets à un module externe au processeur est de $t_{hw0} + \lceil n/l_p \rceil t_{hw+} + b_1(t_{hwblk} + t_{hwackblk} b_2)$, où b_1 et b_2 sont respectivement des variables booléennes qui indique si l'écriture est bloquante et si elle bloque pour attendre l'acquittement.

L'écriture d'un paquet de données vers un module se trouvant sur le même processeur prend un temps de t_{sw0}, puis t_{sw+} pour chaque tranche de l_p octets. De plus, si une autre tâche logicielle avait cessé son exécution en attendant cette écriture, alors $t_{swunblk}$ cycles seront dépensés pour débloquer cette tâche (même si un changement de contexte n'a pas lieu immédiatement). Si l'écriture est bloquante, alors il y a un délai supplémentaire de t_{swblk} et encore un autre délai supplémentaire de $t_{swackblk}$ si la tâche logicielle qui écrit doit attendre l'arrivée de l'acquittement. Ce délai exclut le temps pendant lequel la tâche est suspendue en attente de l'acquittement. Le temps nécessaire à l'écriture d'un paquet de n octets à un autre module sur le même processeur est de $t_{sw0} + \lceil n/l_p \rceil t_{sw+} + b_1(t_{swblk} + t_{swackblk} b_2) + t_{swunblk} b_3$, où b_1, b_2 et b_3 sont respectivement des variables booléennes qui indique si l'écriture est bloquante, si elle bloque pour attendre l'acquittement et si elle débloque l'autre module.

Finalement, la lecture d'un paquet de données en provenance d'un autre module prend un temps de $t_{rdempty}$ si il s'agit d'une lecture non-bloquante sur un canal vide. Dans les autres cas, le temps est de t_{rd0}, puis de t_{rd+} pour chaque tranche de l_p. Si la lecture bloque car les données ne sont pas encore arrivées, alors il faut ajouter un temps d'exécution supplémentaire de t_{rdblk}. Ce délai exclut le temps pendant lequel la tâche est suspendue en attente de l'écriture. Si la tâche logicielle doit renvoyer un acquittement, alors il faut ajouter un délai supplémentaire de t_{swack} si le module se trouve sur le même processeur ou de t_{hwack} si le module est externe au processeur. Si cet acquittement débloque une tâche logicielle sur le même processeur, alors il y a un délai supplémentaire de $t_{swackunblk}$. Le temps nécessaire à la lecture d'un paquet de n octets en provenance d'un autre module est donc de $t_{rd0} + \lceil n/l_p \rceil t_{rd+} + b_1(b_2(t_{swack} + t_{swackunblk} b_3) + (1 - b_2) t_{hwack}) + t_{rdblk} b_4$ où b_1, b_2, b_3 et b_4 sont respectivement des variables booléennes qui indiquent si l'écriture lue était bloquante, si le module transmetteur se trouve sur le même processeur, si il est débloqué par cette lecture et si la lecture bloque.

B.2 Définition de l'application synthétique

Les RTOS sont généralement conçus de manière à ce que le temps requis pour un changement de contexte ou pour démarrer le traitement d'une interruption soit prévisible. Par

exemple, le temps requis par l'ordonnanceur de μC/OS-II pour sélectionner la prochaine tâche à s'exécuter ne dépend pas du nombre de tâches sur le processeur (Labrosse, 2002). On peut donc caractériser les paramètres de performance en exécutant et profilant une application synthétique qui exerce les différents cas d'utilisation du RTOS et de l'API logicielle.

Une telle application synthétique a été créée pour SPACE. Elle est composée d'un périphérique (une mémoire RAM) et de cinq modules (un contrôleur, un répondeur, deux producteurs et un consommateur). Cette application exerce les différents cas d'utilisation de l'API de communications de SPACE : lectures et écritures bloquantes ou non-bloquantes vers des modules ou des périphériques avec différentes tailles de paquet. Selon l'architecture qui l'implémente, l'application synthétique exerce aussi (indirectement) les fonctions de changement de contexte et les ISR du RTOS. Ainsi, les rôles des modules de l'application synthétique sont les suivants :

- Le contrôleur démarre successivement chacune des trois phases de l'application synthétique en communiquant d'abord avec le répondeur puis avec les deux producteurs. Le contrôleur récupère également les résultats de chacune de ces phases de test et donne un rapport global sur le résultat de l'application synthétique.
- Le répondeur teste les différents cas d'écriture et de lecture bloquante ou non-bloquante en communiquant avec le contrôleur lors de la première phase de l'application synthétique.
- Le producteur #1 teste des écritures de différentes tailles vers un module en envoyant des paquets au consommateur lors de la deuxième phase de l'application synthétique.
- Le consommateur teste des lectures de différentes tailles en recevant les paquets du producteur #1 lors de la deuxième phase de l'application synthétique.
- Le producteur #2 teste des lectures et écritures de différentes tailles vers un périphérique en envoyant et recevant des paquets de la mémoire RAM.

La spécification exécutable de cette application synthétique peut également être raffinée vers une implémentation en remplaçant certains modules par leur implémentation logicielle ou matérielle.

B.3 Extraction des paramètres de performance

La méthode automatique d'extraction des paramètres de performance du RTOS et de l'API logicielle utilise l'application synthétique pour tester les différents cas d'utilisation du RTOS et de l'API logicielle. Ainsi, cette méthode génère, simule et profile automatiquement 32 architectures différentes de l'application synthétique. Les 31 premières architectures sont composées d'un bus et d'un processeur et chacune des 31 architectures se distingue par

un partitionnement logiciel/matériel différent des 5 modules de l'application synthétique (la configuration où les 5 modules sont en matériel n'est pas simulée). La 32e architecture utilise deux processeurs pour tester les cas de communication entre processeurs avec le producteur #1 et le consommateur : le contrôleur et le consommateur sont assignés au premier processeur alors que le producteur #1 est assigné au deuxième processeur. Le but de ces simulations est d'obtenir suffisamment de données sur la performance du RTOS et de l'API logicielle pour pouvoir en extraire les paramètres de performance. Ces simulations sont donc profilées à l'aide du profilage exhaustif présenté à la section 6.1.4 pour pouvoir obtenir les temps de chaque entrée et de chaque sortie de chaque fonction logicielle ainsi que les valeurs de ses arguments.

Ainsi, cette méthode automatique examine chacune des 32 ensembles d'enregistrements générées par le profilage exhaustif et en extrait des informations de performance sur les changements de contexte, les interruptions et les communications qui s'y produisent. Par exemple, dans le cas d'un changement de contexte fait par le RTOS μC/OS II, on cherche tous les appels à la fonction d'ordonnancement `OS_Sched` qui résultent en un appel à la fonction de changement de contexte `OSCtxSw` et on en extrait le temps d'exécution de chacune de ces occurrences.

De la même manière, on trouve les routines de service d'interruption en recherchant les appels à la fonction `OS_CPU_ISR`, qui détermine quelle routine de service d'interruption est appelée et on extrait le temps de traitement de l'interruption. En examinant quelle routine est ensuite appelée pour chaque interruption, on peut déterminer séparément le temps d'exécution de la routine qui traite les acquittements et celle qui traite la réception des paquets de données. Ces temps d'exécution sont classés selon la taille du paquet du reçu et selon le déblocage ou non d'une tâche.

Le même principe s'applique également lors de l'examen des appels aux fonctions de communication de SPACE en examinant les valeurs des arguments passés en paramètre et en vérifiant si des appels sont faits à des fonctions de blocage, de déblocage d'une tâche ou d'envoi d'acquittement. Cela permet de classer les temps d'exécution des appels aux fonctions de communication selon la taille des paquets et selon les différents cas de figure. Étant donné qu'on sait à quelle architecture correspond chaque ensemble d'enregistrements, on peut de plus classer les occurrences de ces appels aux fonctions de communication selon que cet appel implique deux modules sur un même processeur ou bien un autre module externe au processeur.

Une fois que les temps d'exécution des différentes occurrences des changements de contexte, des traitements d'interruption et des communications sont classés selon les différents cas de figure, on fait une moyenne arithmétique de ces temps d'exécution pour chaque cas de figure.

Un paramètre de performance comme t_{ctx} peut alors directement en être extrait, mais il faut procéder par déduction pour les autres paramètres. Par exemple, si la seule différence entre deux cas de figure d'appel à une fonction d'écriture vers un périphérique est que le second cas écrit un paquet dont taille est supérieure de l_p à celui écrit dans le premier cas, alors la différence entre le temps d'exécution moyen des deux cas de figures donne la valeur du paramètre t_{per+}. Pour trouver la valeur du paramètre t_{per0}, il faut ensuite trouver le temps d'exécution moyen du cas de figure où on écrit vers un périphérique un paquet de taille l_p et en soustraire la valeur t_{per+}. On procède de manière similaire pour extraire les valeurs de l'ensemble des paramètres de performance. C'est ainsi qu'ont été obtenues, pour le RTOS μCOS II et l'API logicielle de SPACE, les valeurs présentées au tableau 7.6.

ANNEXE C

COMPLEXITÉ COMBINATOIRE DE L'EXPLORATION ARCHITECTURALE

Cette section évalue la complexité combinatoire des problèmes d'exploration architecturale définis à la section 8.2, soit le nombre total d'architectures qui constituent l'espace de recherche. Celui-ci dépend de la fonction f de définition d'espace de recherche et on évalue donc la complexité combinatoire des fonctions f_1 et f_2 présentées à la section 8.1.

C.1 Analyse de la complexité combinatoire de f_1

Tel que présenté à la section 8.1, la fonction de définition d'espace de recherche $f_1 : S \times P_{1,1} \rightarrow \wp(A)$ cible l'ensemble $P_{1,1} \in P$ des plates-formes ayant un seul type de processeur et un seul type de bus. De plus, cette fonction génère l'espace de recherche en suivant les trois étapes ou sous-problèmes suivants :

(a) Le partitionnement logiciel/matériel ;
(b) L'allocation des processeurs et l'assignation des tâches aux processeurs ;
(c) Le choix d'une topologie de communications.

L'analyse combinatoire du sous-problème (a) (Arato *et al.*, 2005) et du sous-problème (b) (Baghdadi *et al.*, 2002) est simple et a déjà été réalisée. L'analyse combinatoire du sous-problème (c) est similaire à celle de (b) pour f_1. Finalement, l'analyse combinatoire des combinaisons de ces sous-problèmes, soit la complexité combinatoire des problèmes d'exploration architecturale définis à la section 8.2, ne semble pas avoir été considérée dans des travaux antérieurs. On constate que ces problèmes d'exploration architecturale ont une complexité combinatoire bien plus importante que chacun des problèmes (a), (b) ou (c) pris séparément. On débute d'abord par une revue de l'analyse combinatoire des problèmes (a) et (b).

C.1.1 Analyse combinatoire du partitionnement logiciel/matériel

Soit une application avec n modules. Le partitionnement logiciel/matériel consiste alors à partitionner les n modules de l'application en deux sous-ensembles disjoints, le premier contenant les modules à implémenter en logiciel et le second les modules à implémenter en matériel. Il faut donc décider, pour chacun des n modules, si celui-ci sera implémenté en logiciel ou en matériel. Comme il y a 2 choix pour chacun des n modules et que ces choix sont indépendants entre eux d'un point de vue combinatoire, le nombre de solutions possibles

est égal à 2^n (Arato *et al.*, 2005). En termes combinatoires, ce résultat est égal au nombre de manières de répartir n balles distinctes dans 2 boîtes distinctes (Riordan, 1958, p. 90).

C.1.2 Analyse combinatoire de l'allocation des processeurs et de l'assignation des tâches aux processeurs

Soit une application avec n tâches logicielles. L'allocation des processeurs consiste à choisir le nombre k de processeurs à allouer, alors que l'assignation des tâches aux processeurs consiste à assigner chacune des n tâches logicielles du système à un des k processeurs alloués de manière à ce que chaque processeur exécute au moins une tâche logicielle.

Supposons que k processeurs ont été alloués tel que $0 \le k \le n$. Étant donné qu'on cible une plate-forme $p \in P_{1,1}$, tous les processeurs sont identiques. En termes combinatoires, l'assignation des tâches aux processeurs revient donc à partitionner un ensemble de n objets distincts en k sous-ensembles non vides. Le nombre de solutions possibles est alors égal à $S(n,k)$, le nombre de Stirling de seconde espèce (Comtet, 1974, p. 204) :

$$S(n,k) = \frac{1}{k!}\sum_{j=0}^{k}(-1)^j\binom{k}{j}(k-j)^n \tag{C.1}$$

Dans cette équation, $\binom{k}{j} = \frac{k!}{j!(k-j)!}$ désigne un coefficient binomial. Un tableau des valeurs des nombres de Stirling de seconde espèce est disponible dans (Sloane, 2010, série A008277). En particulier, on a $S(n,0) = 0$ si $n \neq 0$ et, par convention, $S(0,0) = 1$. Le nombre total de solutions possibles à l'allocation des processeurs suivie de l'assignation des tâches est donné par la formule suivante :

$$\sum_{k=0}^{n} S(n,k) = B(n) \tag{C.2}$$

On obtient alors le nombre de Bell $B(n)$, qui désigne le nombre de partitions d'un ensemble de taille n (Comtet, 1974, p. 210). En termes combinatoires, c'est le nombre de manières de placer n balles distinctes dans des boîtes identiques. Les nombres de Bell forment la série suivante : 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, ... (Sloane, 2010, série A000110). Le résultat obtenu ici est équivalent au résultat obtenu dans (Baghdadi *et al.*, 2002) quand on suppose que tous les processeurs sont identiques.

On peut effectuer simultanément le partitionnement logiciel/matériel, l'allocation des processeurs et l'assignation des tâches logicielles aux processeurs. Soit une application avec n modules et supposons qu'on décide d'implémenter k de ces modules en logiciel tel que $0 \le k \le n$. Le nombre de manières de choisir ces k modules logiciels parmi les n modules de l'application est égal au coefficient binomial $\binom{n}{k}$. Ensuite, le nombre de manières d'allouer des processeurs

et d'assigner ces k modules logiciels aux processeurs est égal à $B(k)$ selon l'équation C.2. Le nombre de solutions possibles au problème combiné de partitionnement logiciel/matériel, d'allocation des processeurs et d'assignation des tâches logicielles aux processeurs est donc égal à :

$$\sum_{k=0}^{n} \binom{n}{k} B(k) = B(n+1) \tag{C.3}$$

En effet, l'équation C.3 est une relation de récurrence des nombres de Bell (Comtet, 1974, p. 210). Une autre manière d'arriver à ce résultat est d'ajouter un module virtuel aux n modules et de résoudre le problème d'allocation des processeurs et d'assignation des tâches aux processeurs avec ces $n+1$ modules. Ce problème a alors aussi $B(n+1)$ solutions et on interprète une solution de ce problème de telle façon que le « processeur » sur lequel se trouve le module virtuel contient en fait l'ensemble des modules matériels.

C.1.3 Analyse combinatoire du choix de topologie de communication

À première vue, ce problème est équivalent au problème de l'allocation des processeurs et de l'assignation des tâches aux processeurs. Ainsi, on pourrait simplement considérer que, pour n composants matériels, on alloue $0 \leq k \leq n$ bus, que le nombre de manières d'assigner les n composants aux k bus est $S(n,k)$ et que le nombre de solutions possibles est donc égal à $B(n)$. Cependant, il y a une différence majeure entre ces deux problèmes. Bien qu'on puisse allouer un processeur pour y exécuter une seule tâche logicielle, il est généralement inutile d'allouer un bus pour y assigner un seul composant matériel. Il faut donc qu'au moins deux composants matériels soient assignés à chaque bus pour qu'une topologie de communications soit considérée valide.

Supposons que nous avons déjà alloué k bus et que nous avons n composants matériels à assigner à ces k bus de telle sorte que chaque composant est assigné à exactement un bus et qu'au moins deux composants sont assignés à chaque bus. Cela revient donc à placer n balles distinctes dans k boites identiques de telle sorte que chaque boite contienne au moins 2 balles et le nombre de solutions possibles à ce problème est égal à $S_2(n,k)$, le nombre de Stirling 2-associé de seconde espèce (Sinha et Bhattacharya, 1985; Charalambides, 2005, p. 136), qu'on obtient par la formule suivante (Charalambides, 2005, p. 126) :

$$S_2(n,k) = \sum_{j=0}^{k} (-1)^j \binom{n}{j} S(n-j, k-j) \tag{C.4}$$

On trouve dans (Sloane, 2010, série A008299) un tableau des valeurs des nombres de Stirling 2-associés de seconde espèce. On obtient le nombre de choix possibles d'une topologie

de communication pour f_1 en considérant qu'on peut allouer $0 \le k \le \lfloor n/2 \rfloor$ bus et que, pour chaque valeur de k, il y a $S_2(n,k)$ solutions possibles :

$$\sum_{k=0}^{\lfloor n/2 \rfloor} S_2(n,k) = B_2(n) \tag{C.5}$$

Ce résultat est égal au nombre de Bell 2-associé $B_2(n)$, qui désigne le nombre de partitions d'un ensemble de n éléments tel que chaque partition est composée de sous-ensembles ayant chacun au moins 2 éléments (Sinha et Bhattacharya, 1985). En termes combinatoires, il s'agit du nombre de manières de placer n balles distinctes dans des boites identiques de telle sorte que chaque boite contienne au moins 2 balles. Les premiers termes de la série des nombres de Bell 2-associés sont les suivants : 1, 0, 1, 1, 4, 11, 41, 162, 715, 3425, 17722, ... (Sloane, 2010, série A000296).

C.1.4 Analyse combinatoire de l'exploration architecturale

Nous disposons maintenant des outils nécessaires pour trouver le nombre de solutions possibles aux problèmes d'exploration architecturale dont l'espace de recherche est généré par f_1, soit une composition du partitionnement logiciel/matériel, de l'allocation des processeurs et de l'allocation des tâches logicielles aux processeurs ainsi que du choix d'une topologie de communications.

Soit une application avec n modules. Si on décide d'implémenter $0 \le i \le n$ modules en tant que tâches logicielles, alors il y a $\binom{n}{i}$ manières de choisir ces i modules, alors que les $n-i$ autres modules sont implémentés en tant que composants matériels. Ensuite, on peut décider d'allouer $0 \le j \le i$ processeurs et il y a $S(i,j)$ manières d'assigner ces i tâches logicielles aux j processeurs de manière à ce que chaque processeur ait au moins une tâche et que chaque tâche soit assignée à un et un seul processeur.

La prochaine étape est l'allocation des bus. Les processeurs, une fois que les tâches logicielles leur ont été assignées, cessent d'être des objets identiques et deviennent des objets distincts étant donné que les tâches sont distinctes. Ils peuvent donc être considérés comme des composants matériels distincts pour le choix d'une topologie de communications. Tel que décrit à la section 8.1, la fonction de définition d'espace de recherche f_1 autorise un processeur à être assigné seul sur un bus. Il faut donc considérer que $0 \le k \le j$ processeurs peuvent se trouver chacun seul sur un total de k bus. Il y a $\binom{j}{k}$ manières de choisir ces k processeurs. Il reste alors à allouer d'autres bus et à leur assigner les $n-i+j-k$ composants matériels restants de manière à ce que chacun de ces autres bus ait au moins 2 composants matériels qui lui soient assignés. Le nombre de topologies de communication pour ces composants restants est de $B_2(n-i+j-k)$ selon l'équation C.5. On obtient donc que le nombre total de solutions

possibles aux problèmes d'exploration architecturale selon f_1 est :

$$E(n) = \sum_{i=0}^{n} \binom{n}{i} \sum_{j=0}^{i} S(i,j) \sum_{k=0}^{j} \binom{j}{k} B_2(n-i+j-k) \tag{C.6}$$

Les premiers termes de la série des valeurs de $E(n)$ pour $0 \leq n \leq 10$ sont 0, 1, 6, 31, 218, 1789, 16901, 179533, 2112247, 27202478, 379944116, ... La figure C.1 illustre la croissance de $E(n)$ selon une échelle logarithmique de base 10 pour $0 \leq n \leq 10$. On y constate sans surprise que $E(n)$ croit beaucoup plus vite que le nombre de solutions au problème du partitionnement logiciel/matériel (2^n) ou au problème d'allocation des processeurs et d'assignation des tâches aux processeurs ($B(n)$). Cela s'explique simplement par le fait que ces problèmes sont des sous-problèmes du problème général d'exploration architecturale. On constate également que, pour $0 \leq n \leq 10$, $E(n)$ croit plus vite que $n!$. On montre par contre ci-dessous que $E(n)$ devient borné par $n!$ lorsque n devient suffisamment grand. La série des valeurs de $E(n)$ semble être une nouvelle série qui n'est pas répertoriée dans (Sloane, 2010) et qui, au meilleur de notre connaissance, n'est pas mentionnée dans des travaux antérieurs. On présente donc ici les principales propriétés de cette nouvelle fonction $E(n)$.

C.1.4.1 Relation de récurrence

Si on doit réaliser l'exploration architecturale d'une application comprenant $n+1$ modules, alors le nombre de solutions est de $E(n+1)$. Supposons qu'on décide de placer le $(n+1)^{eme}$ module sur un premier bus. Si on décide que ce module sera seul sur ce bus, alors ce module doit être en logiciel sur un processeur (parce qu'un module matériel ne peut se trouver seul sur un bus) et il y a alors $E(n)$ manières de placer les n autres modules du système sur les autres bus. Si on décide de placer $0 \leq k \leq n-1$ modules sur les autres bus du système, et donc de placer les $n-k$ autres modules sur le même bus que le $(n+1)^{eme}$ module, alors il y a $\binom{n}{k}$ manières de choisir ces k modules et il y a $E(k)$ manières de placer ces k modules sur les autres bus du système. Il reste alors à placer les $n-k+1$ autres modules (incluant le $(n+1)^{eme}$ module) sur le premier bus. Or c'est un problème combiné de partitionnement logiciel/matériel, d'allocation des processeurs et d'assignation des tâches aux processeurs et on sait que le nombre de possibilités dans ce cas est égal à $B(n-k+2)$ selon l'équation C.3. Les valeurs de E suivent donc la relation de récurrence suivante :

$$E(n+1) = E(n) + \sum_{k=0}^{n-1} \binom{n}{k} B(n-k+2) E(k) \tag{C.7}$$

Cette relation de récurrence se vérifie pour les valeurs de $E(n)$ données ci-haut.

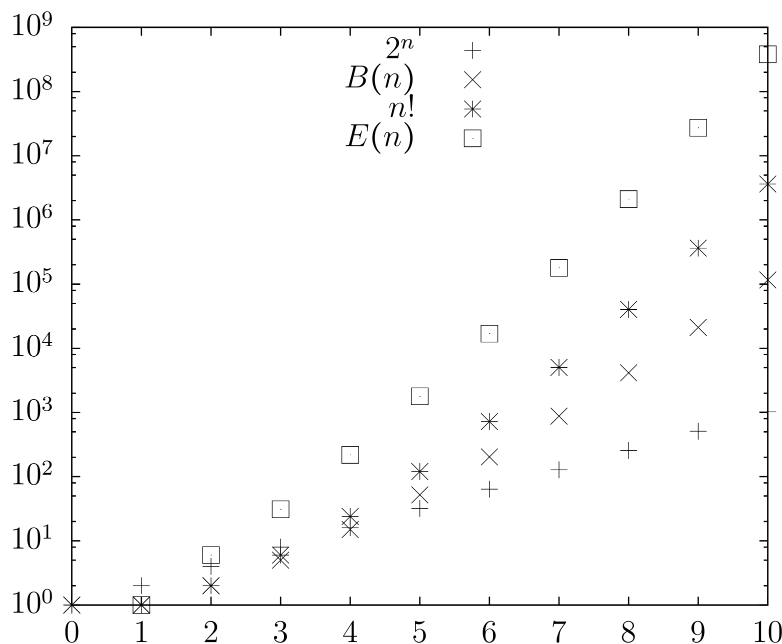

Figure C.1 Comparaison de la croissance de $E(n)$ avec 2^n, $B(n)$ et $n!$

C.1.4.2 Fonction génératrice

Définissons une séquence $e_0, e_1, e_2, \cdots = \{e_n\}_0^\infty$ telle que $e_n = E(n)$. La fonction génératrice exponentielle de e_n est alors une fonction f donnée par l'équation suivante (Wilf, 1994, p. 39) :

$$f(x) = \sum_{n=0}^{\infty} \frac{e_n}{n!} x^n \tag{C.8}$$

En utilisant la définition combinatoire de la fonction f_1 donnée à la figure C.2 pour la librairie `Combstruct` du logiciel mathématique Maple (Flajolet et Salvy, 1995), on trouve que la fonction génératrice exponentielle de e_n est :

$$f(x) = e^{-1-x+e^{-1+x+e^x}} \tag{C.9}$$

Si on a une séquence $a_0, a_1, a_2, \cdots = \{a_n\}_0^\infty$, on définit de manière analogue la fonction génératrice ordinaire de a_n par (Wilf, 1994, p. 33) :

$$f(x) = \sum_{n=0}^{\infty} a_n x^n \tag{C.10}$$

La fonction f donnée par l'équation $C.9$ est donc la fonction génératrice ordinaire de la séquence $\{a_n\}_0^\infty$ telle que $a_n = \frac{E(n)}{n!}$

```
[N, {N  = Set(Union(BS,Union(BM,BH))),
     BS = Set(Set(P,card>=1), card=1),
     BM = Set(Prod(Set(P, card>=1),Set(H, card>=1)), card = 1),
     BH = Set(Set(H,card >=2), card=1),
     P = Set(S, card >= 1), H=Atom, S=Atom}, labelled]
```

Figure C.2 Définition combinatoire de f_1 dans Maple

C.1.4.3 Analyse asymptotique

La fonction génératrice ordinaire de a_n définie par l'équation $C.9$ est analytique sur l'ensemble du plan complexe. En d'autres termes, elle n'a aucune singularité dans le plan complexe. On peut donc en déduire (Wilf, 1994, p. 49) que, pour tout $\epsilon > 0$ donné, il existe un N tel que $a_n < \epsilon^n$ pour tout $n > N$. Cela équivaut à dire que $a_n \in O(\epsilon^n)$, et donc $E(n) \in O(\epsilon^n n!)$ pour tout $\epsilon > 0$. Cela donne donc une borne supérieure asymptotique à la croissance de $E(n)$.

Inversement, il est trivial de montrer que $E(n) \geq B(n)$ pour $n > 0$ étant donné que $B(n)$ est le nombre de solutions au problème de l'allocation des processeurs et de l'assignation de n tâches logicielles sur un seul bus, alors que $E(n)$ énumère en plus des solutions contenant des modules matériels et plus d'un bus. $B(n)$ est donc une borne inférieure asymptotique à la croissance de $E(n)$ et, avec la borne supérieure asymptotique donnée au paragraphe précédent, cela signifie que $E(n)$ est une fonction quasi-factorielle selon la terminologie de (Bernardi *et al.*, 2007).

On sait que $B(n)$ tend asymptotiquement vers la fonction suivante (de Bruijn, 1981, p. 108) :

$$B(n) \sim \left(\frac{n}{\log n}\right)^n \tag{C.11}$$

Cela revient à dire que :

$$\lim_{n\to\infty} \frac{B(n)}{\left(\frac{n}{\log n}\right)^n} = 1 \tag{C.12}$$

On peut donc affirmer que, pour toute constante c, la fonction c^n est une borne inférieure asymptotique à $B(n)$ et donc à $E(n)$. On a donc $E(n) \in \Omega\left(\left(\frac{n}{\log n}\right)^n\right)$ ainsi que $E(n) \in \Omega(c^n)$ pour toute constante c.

On a donc réussi à établir que, pour toute constante c et pour tout $\epsilon > 0$, il existe un N tel que, pour tout $n > N$, on a :

$$c^n \leq \left(\frac{n}{\log n}\right)^n \leq E(n) \leq \epsilon^n n! \tag{C.13}$$

Cependant, on constate qu'en pratique la fonction $E(n)$ croit très vite pour les premières valeurs de n. Si on compare $E(n)$ à $n!$, on constate que, bien que $n!$ en vient éventuellement à dominer $E(n)$, cela ne se produit que pour les $n > 148$. On a donc $E(147) > 147! \approx 1.73 \times 10^{256}$. Pour mettre ce nombre en perspective, considérons que le nombre total d'opérations logiques élémentaires qui auraient pu être effectuées par l'ensemble de l'univers depuis qu'il existe est de l'ordre de 10^{120} (Lloyd, 2002).

C.1.5 Modules restreints au logiciel ou au matériel

Jusqu'ici, on a supposé que l'application était seulement composée de n modules qui pouvaient chacun être implémentés soit en logiciel, soit en matériel. On considère maintenant le cas où l'application contient également s tâches logicielles et h composants matériels. Aucune de ces s tâches logicielles ne peut être implémentée en matériel, tout comme aucun de ces h composants matériels ne peut être implémenté en logiciel.

Pour obtenir le nombre de solutions possibles selon f_1, il suffit de modifier l'équation C.6 en ajoutant s au nombre de tâches logicielles à assigner aux processeurs et h au nombre de composants matériels à assigner aux bus. On obtient alors l'équation suivante :

$$E(n,s,h) = \sum_{i=0}^{n} \binom{n}{i} \sum_{j=0}^{i+s} S(i+s,j) \sum_{k=0}^{j} \binom{j}{k} B_2(n-i+j-k+h) \tag{C.14}$$

On a bien sûr $E(n,0,0) = E(n)$. On obtient également $E(0,0,h) = B_2(h)$ et le problème se réduit alors à choisir une topologie de communications pour h composants matériels selon l'équation C.5.

Dans le cas où on a $E(0,s,0)$, on réduit l'équation C.14 à :

$$E(0,s,0) = \sum_{j=0}^{s} S(s,j) \sum_{k=0}^{j} \binom{j}{k} B_2(j-k) \tag{C.15}$$

Or les nombres de Bell 2-associés suivent la relation de récurrence suivante (Sinha et Bhattacharya, 1985) :

$$\sum_{k=1}^{j} \binom{j}{k} B_2(j-k) = B_2(j+1) \tag{C.16}$$

Si on utilise cette identité pour simplifier l'équation C.15, on obtient

$$E(0,s,0) = \sum_{j=0}^{s} S(s,j) \left[B_2(j) + B_2(j+1)\right] \tag{C.17}$$

Étant donné l'identité $B(n) = B_2(n) + B_2(n+1)$ (Bernhart, 1999), on obtient donc :

$$E(0,s,0) = \sum_{j=0}^{s} S(s,j)B(j) = B^2(s) \tag{C.18}$$

Le nombre de possibilités dans le cas où on a seulement s tâches logicielles est donc égal à $B^2(s)$, soit le nombre de Bell de deuxième ordre (à ne pas confondre avec le nombre de Bell 2-associé $B_2(s)$). Ce nombre représente le nombre de manières dont on peut partitionner les partitions d'un ensemble (Penson *et al.*, 2004). En effet, le problème consiste alors à partitionner les s tâches logicielles sur un certain nombre de processeurs, puis à partitionner ces processeurs sur un certain nombre de bus. Ce nombre représente également le nombre d'arbres de profondeur uniforme 3 ayant exactement s feuilles distinctes (Hogg et Huberman, 1985). On peut effectivement représenter chaque solution de $E(0,s,0)$ par un tel arbre : le niveau 3 est composé de s feuilles distinctes représentant les s tâches logicielles. Le niveau 2 de l'arbre est composé des processeurs sur lesquels sont assignés les s tâches et le niveau 1 est composé des bus sur lesquels sont assignés les processeurs. Par convention, le niveau 0 de l'arbre est sa racine. Les nombres de Bell de deuxième ordre forment la série suivante : $1, 1, 3, 12, 60, 358, 2471, 19302, 167894, \ldots$ (Sloane, 2010, Série A000258).

En résumé, les résultats de l'analyse combinatoire présentés dans cette section sont, à notre connaissance, inédits et généralisent de manière élégante des cas particuliers qui ont été étudiés dans des travaux antérieurs.

C.2 Analyse de la complexité combinatoire de f_2

Cette section présente une borne inférieure à la complexité combinatoire de f_2, où certains composants matériels sont double-port et peuvent être assignés simultanément à deux bus distincts. On commence par analyser un problème combinatoire préalable, qui permet d'obtenir ensuite une borne inférieure au nombre de topologie de communications selon f_2, puis à la fonction f_2 elle-même.

C.2.1 Analyse de $D_r(n,k)$

Supposons que nous avons n balles distinctes à placer dans $k + r$ boites, comprenant k boites identiques et r boites distinctes, de telle sorte que chacune des k boites identiques contienne au moins 2 balles. Combien y a-t-il de solutions possibles ? Nous désignons ce nombre de solutions par $D_r(n,k)$.

On note qu'il est possible de ne placer aucune balle dans une ou plusieurs des r boites distinctes. De manière réciproque, on peut donc décider de placer des balles dans $0 \leq i \leq r$ boites parmi les r boites distinctes. Supposons qu'on décide de placer des balles dans

seulement i boites parmi ces r boites. Il existe alors $\binom{r}{i}$ manières de choisir ces i boites. Ensuite, on peut choisir de placer une seule balle par boite pour $0 \leq j \leq i$ boites parmi ces i boites, et de placer au moins 2 balles par boite dans chacune des $i-j$ autres boites. Supposons qu'on décide de placer une seule balle par boite pour j boites. Il existe alors $\binom{i}{j}$ manières de choisir ces j boites et il y a $\binom{n}{j}$ manières de choisir les j balles qui iront dans ces j boites. De plus, étant donné que ces boites sont distinctes, il y a $j!$ manières de placer les j balles choisies dans ces j boites.

Il reste maintenant à placer les $n-j$ balles restantes dans les k boites identiques et les $i-j$ boites distinctes restantes de manière à ce qu'il y ait au moins 2 balles dans chacune de ces $k+i-j$ boites. Il existe $S_2(n-j, k+i-j)$ manières de partitionner ces $n-j$ balles en $k+i-j$ sous-ensembles d'au moins 2 balles. Ensuite, il y a $\binom{k+i-j}{i-j}$ manières de sélectionner les $i-j$ sous-ensembles de balles qui seront placés dans les $i-j$ boites distinctes, alors que les k sous-ensembles restants seront placés dans les boites identiques. Comme ces $i-j$ boites sont distinctes, il y a $(i-j)!$ façons d'y placer ces $i-j$ sous-ensembles. On obtient donc le nombre de solutions possibles via la formule suivante :

$$D_r(n,k) = \sum_{i=0}^{r} \binom{r}{i} \sum_{j=0}^{i} \binom{i}{j} \binom{n}{j} j! \binom{k+i-j}{i-j} (i-j)! S_2(n-j, k+i-j) \tag{C.19}$$

On peut simplifier cette équation en utilisant l'identité $i! = j!(i-j)!\binom{i}{j}$ et on obtient alors :

$$D_r(n,k) = \sum_{i=0}^{r} i! \binom{r}{i} \sum_{j=0}^{i} \binom{n}{j} \binom{k+i-j}{i-j} S_2(n-j, k+i-j) \tag{C.20}$$

En effet, il y a $i!$ manières de placer i ensembles de balles dans i boites distinctes. Étant donné que j de ces ensembles ne contiennent qu'une seule balle et que les $j-i$ autres contiennent au moins 2 balles, le choix des j boites dans lesquels on ne place qu'une seule balle se fait implicitement lors du choix d'une permutation des i ensemble parmi les $i!$ possibilités.

Si on a $r=0$ dans l'équation C.20, on obtient $D_0(n,k) = S_2(n,k)$. Ce résultat est cohérent avec le fait que $D_0(n,k)$ désigne le nombre de manières de répartir n balles dans k boites identiques de manière à avoir au moins 2 balles dans chaque boite. De manière similaire, si on a $k=0$ dans l'équation C.20, on obtient alors :

$$D_r(n,0) = \sum_{i=0}^{r} i! \binom{r}{i} \sum_{j=0}^{i} \binom{n}{j} S_2(n-j, i-j) \tag{C.21}$$

Or on sait que l'identité suivante existe (Riordan, 1958, p. 77) :

$$S(n,i) = \sum_{j=0}^{i} \binom{n}{j} S_2(n-j, i-j) \tag{C.22}$$

Avec cette identité, on simplifie l'équation C.21 pour obtenir :

$$D_r(n,0) = \sum_{i=0}^{r} i!\binom{r}{i}S(n,i) = \sum_{i=0}^{n} i!\binom{r}{i}S(n,i) = r^n \tag{C.23}$$

En effet, on peut remplacer r par n dans la sommation étant donné qu'on a $\binom{r}{i} = 0$ pour chaque $i > r$ et $S(n,i) = 0$ pour chaque $i > n$. Finalement, la dernière égalité est une identité fondamentale des nombres de Stirling de seconde espèce (Riordan, 1958, p. 33). $D_r(n,0)$ représente le nombre de manières de placer n balles distinctes dans r boites distinctes et il est connu que ce problème combinatoire admet r^n solutions. Le résultat obtenu via l'équation C.23 est donc cohérent avec cette interprétation combinatoire.

C.2.2 Analyse combinatoire des topologies avec composants double-port

Généralement, lorsqu'un composant matériel est double-port, son deuxième port est optionnel. En d'autres termes, on peut soit choisir de connecter seulement le premier port du composant à un bus, soit connecter les deux ports à deux bus différents. Pour simplifier l'analyse combinatoire, nous supposerons en premier lieu qu'un tel composant doit obligatoirement être connecté à deux bus différents.

C.2.2.1 Composants double-port assignés à exactement deux bus

Supposons d'abord que nous avons n composants matériels qui doivent chacun être assignés à un seul bus, d composants matériels qui doivent chacun être assignés à deux bus différents et k bus auxquels doivent être assignés au moins 2 composants matériels. Désignons le nombre de manières différentes d'assigner ces composants aux bus par $K_d(n,k)$.

On peut d'abord résoudre un problème apparenté qui est de placer $n+2d$ balles distinctes dans k boites identiques de manière à ce que chaque boite contienne au moins 2 balles. De plus, les $2d$ balles représentent d paires de balles et on doit s'assurer que les deux balles d'une même paire ne se trouvent pas dans la même boîte. Le nombre de manières $K'_d(n,k)$ de placer ces balles est inférieur à $S_2(n+2d,k)$. Ce serait le nombre exact de solutions possibles du problème apparenté si les deux balles d'une même paire pouvaient se trouver dans une même boîte. Il faut donc retrancher à $S_2(n+2d,k)$ le nombre de configurations où deux balles d'une même paire se trouvent dans la même boite.

Supposons que, parmi les d paires, on sache que r paires déjà choisies sont placées de telle sorte que les deux balles d'une même paire se trouvent dans la même boite, sans exclure la possibilités que d'autres paires se trouvent dans la même situation, et notons le nombre de solutions remplissant ce critère par N_r. Ces r paires peuvent être placées dans $0 \leq i \leq r$ boites. Si elles sont placées dans i boites, il y a alors $S(r,i)$ manières de les y placer. Étant donné

que $2r$ balles sont déjà placées dans i boites, il reste $n+2d-2r$ balles à placer dans $k-i$ boites identiques et i boites distinctes en plaçant au moins 2 balles dans chacune des $k-i$ boites. Le nombre de manières d'y placer les balles restantes est alors égal à $D_i(n+2d-2r, k-i)$, tel que défini par l'équation C.20. On obtient donc :

$$N_r = \sum_{i=0}^{r} S(r,i) D_i(n+2d-2r, k-i) \tag{C.24}$$

On cherche le nombre de configurations $K'_d(n,k)$ telles qu'aucune paire de balles n'est placée avec ses deux balles dans la même boite. En utilisant le principe d'inclusion et d'exclusion (Charalambides, 2005, p. 30), on obtient :

$$K'_d(n,k) = \sum_{r=0}^{d} (-1)^r \binom{d}{r} N_r = \sum_{r=0}^{d} (-1)^r \binom{d}{r} \sum_{i=0}^{r} S(r,i) D_i(n+2d-2r, k-i) \tag{C.25}$$

Comme les deux ports d'un même composant double-port sont représentés comme deux objets distincts dans le calcul de $K'_d(n,k)$, alors la permutation des deux port d'un même composant double-port pourrait (mais pas nécessairement) y produire deux solutions distinctes. Cependant, ces deux ports sont considérés comme équivalents dans le calcul de $K_d(n,k)$ et cette permutation n'y représente donc qu'une seule et même solution. S'il y a d composants dual-port, il correspond donc au plus 2^d solutions dans $K'_d(n,k)$ pour chaque solution dans $K_d(n,k)$. On obtient donc que $2^d K_d(n,k) \geq K'_d(n,k)$ et, en appliquant l'équation C.25, on obtient la borne inférieure suivante pour $K_d(n,k)$:

$$K_d(n,k) \geq \frac{1}{2^d} \left[\sum_{r=0}^{d} (-1)^r \binom{d}{r} \sum_{i=0}^{r} S(r,i) D_i(n+2d-2r, k-i) \right] \tag{C.26}$$

Lorsque $d=0$, donc lorsqu'il n'y a pas de composants double-port, on obtient le résultat suivant via les équations C.26 et C.20 :

$$K_0(n,k) \geq D_0(n,k) = S_2(n,k) \tag{C.27}$$

Étant donné que le problème se réduit alors à répartir n balles distinctes dans k boites identiques de manière à ce que chaque boite ait au moins 2 balles, on a en fait une égalité dans ce cas précis et $K_0(n,k) = S_2(n,k)$.

C.2.2.2 Analyse combinatoire des topologies de communication de f_2

Supposons que nous avons n composants matériels à un port et un composant double-port qui peut être assigné soit à un seul bus, soit à deux bus différents, et ce de manière à ce qu'au moins deux composants soient assignés à chaque bus. Si nous voulons assigner ces

composants à exactement k bus, alors soit le composant double-port est assigné à un seul bus et le nombre de possibilité est alors de $K_0(n+1,k)$, soit il est assigné à deux bus et le nombre de possibilités est alors de $K_1(n,k)$. Le nombre total de possibilités est donc de $K_0(n+1,k)+K_1(n,k)$.

De manière plus générale, supposons que nous avons n composants matériels à un port et d composants double-port qui peuvent chacun être assigné soit à un seul bus, soit à deux bus différents, et ce de manière à ce qu'au moins deux composants soient assignés à chacun des k bus. Si on décide que r des d composants double-port seront chacun assignés à deux bus différents, et donc que $d-r$ composants double-port seront assignés chacun à un seul bus, il y a alors $\binom{d}{r}$ manières de choisir ces r composants et, pour un choix donné, il y a $K_r(n+d-r,k)$ manières de répartir ces composants sur les k bus. Étant donné qu'on peut choisir d'allouer $0 \leq k \leq \lfloor n/2 \rfloor + d$ bus, le nombre de manières différentes d'allouer des bus et d'assigner ces composants à ces bus est donc de :

$$T(n,d) = \sum_{k=0}^{\lfloor n/2 \rfloor + d} \sum_{r=0}^{d} \binom{d}{r} K_r(n+d-r,k) \tag{C.28}$$

Si $d=0$, cette équation se réduit à l'équation C.5 en utilisant l'identité $K_0(n,k)=S_2(n,k)$ et on obtient donc $T(n,0)=B_2(n)$, qui est le nombre de manières de partitionner un ensemble de n objets tel que chaque sous-ensemble contienne au moins 2 objets. De manière générale, on peut calculer une borne inférieure à $T(n,d)$ en utilisant l'équation C.26 qui donne borne inférieure à $K_r(n+d-r,k)$.

C.2.3 Analyse combinatoire de l'exploration architecturale selon f_2

Pour obtenir le nombre de solutions possibles au problème d'exploration architecturale selon f_2, on applique la même logique que pour l'exploration architecturale selon f_1 à la section C.1.4. La différence entre f_1 et f_2 apparaît une fois que sont fixés le partitionnement logiciel/matériel, l'allocation des processeurs, l'assignation des tâches aux processeurs et le choix des processeurs qui se trouvent chacun seul sur un bus. Avec f_1, on se trouvait alors avec $n-i+j-k$ composants matériels à un seul port et le nombre de topologies pour ces composants était de $B_2(n-i+j-k)$. Avec f_2, on se trouve en plus avec d autres composants double-port et le nombre de topologies pour les composants matériels est alors de $T(n-i+j-k,d)$. Le nombre de solutions possibles pour l'exploration architecturale selon f_2 est donc donné par :

$$E_2(n,d) = \sum_{i=0}^{n} \binom{n}{i} \sum_{j=0}^{i} S(i,j) \sum_{k=0}^{j} \binom{j}{k} T(n-i+j-k,d) \tag{C.29}$$

On a que $E_2(n,0)=E(n)$ étant donné que $T(n-i+j-k,0)=B_2(n-i+j-k)$. Il est

possible de trouver une borne inférieure à $E_2(n,d)$ en appliquant l'équation C.28 et la borne inférieure de l'équation C.26. Le tableau C.1 présente les valeurs des bornes inférieures de $E_2(n,d)$ pour $0 \leq n \leq 10$ et $0 \leq d \leq 4$. On constate que le fait d'augmenter le nombre d de composants double-port augmente grandement le nombre d'architectures possibles.

Tableau C.1 Bornes inférieures de $E_2(n,d)$ pour $0 \leq n \leq 10$ et $0 \leq d \leq 4$

n\d	0	1	2	3	4
0	0	0	1	8	84
1	1	2	9	75	866
2	6	13	81	758	9984
3	31	108	784	8537	126949
4	218	999	8538	106011	1760088
5	1789	10378	102660	1436039	26370756
6	16901	119329	1348547	21034003	423900722
7	179533	1502952	19181141	330761670	7267559729
8	2112247	20551583	293290020	5551162652	132234473202
9	27202478	302856160	4792245931	98940738871	2542747661071
10	379944116	4779935788	83255046761	1864885214156	51485909731982

ANNEXE D

COMPLEXITÉ ALGORITHMIQUE DE L'EXPLORATION ARCHITECTURALE

Même s'il existe un très grand nombre de solutions possibles aux problèmes d'exploration architecturale, ceux-ci pourraient être résolus rapidement s'il existait pour ces problèmes un algorithme dont le temps d'exécution est une fonction polynomiale du nombre de modules (soit un algorithme qui appartient à l'ensemble P dans la théorie de la complexité des algorithmes). Or, sauf si $P = NP$, un tel algorithme n'existe pas : des étapes de l'exploration architecturale sont NP-difficiles et l'exploration architecturale elle-même est donc NP-difficiles. Cette discussion sur la complexité algorithmique de l'exploration architecturale suppose que $P \neq NP$.

Il a été démontré que, lorsqu'on considère séparément le temps d'exécution et la quantité des ressources matérielles, le partitionnement logiciel/matériel est un problème fortement NP-difficile (Mann, 2004; Arato *et al.*, 2005). Cela signifie non seulement qu'il n'existe pas d'algorithme polynomial pour résoudre ce problème de manière exacte, mais également qu'il n'existe pas d'algorithme exact pseudo-polynomial. En d'autres termes, ce problème demeure NP-difficile même si toutes les métriques associées à une instance du problème (temps d'exécution et quantité de ressources matérielles des modules) sont une fonction polynomiale du nombre de modules. (Garey et Johnson, 1979) (Une autre conséquence de ce résultat est qu'il n'existe pas de schéma d'approximation en temps complètement polynomial pour ce problème.)

Un problème ouvert présenté dans (Mann, 2004) est de savoir s'il existe un algorithme d'approximation avec facteur constant $\rho > 1$ pour le partitionnement logiciel/matériel, c'est-à-dire un algorithme qui peut trouver en un temps polynomial, pour toute instance du problème de partitionnement logiciel/matériel, une solution dont la valeur est au plus ρ fois celle de la solution optimale. De manière plus générale, on peut se demander s'il existe un schéma d'approximation en temps polynomial (PTAS) pour le partitionnement logiciel/matériel, c'est-à-dire un algorithme qui, pour tout $\epsilon > 0$ et toute instance du problème de partitionnement logiciel/matériel, peut trouver une solution dont la valeur est au plus $1 + \epsilon$ fois celle de la solution optimale. Nous donnons ici une solution à ce problème ouvert : il n'existe pas de PTAS pour le problème du partitionnement logiciel/matériel.

En effet, on trouve dans (Mann, 2004; Arato *et al.*, 2005) une réduction polynomiale du problème de la bissection minimale d'un graphe au problème du partitionnement logi-

ciel/matériel ainsi qu'une réduction polynomiale d'une solution du partitionnement logiciel/ matériel et de sa valeur à une solution du problème de la bissection minimale d'un graphe. Supposons qu'il existe un PTAS pour le partitionnement logiciel/matériel, alors il existerait un PTAS pour la bissection minimale qu'on pourrait implémenter de la manière suivante : réduire l'instance de la bissection minimale à une instance du partitionnement logiciel/matériel, trouver la solution approximative au problème de partitionnement logiciel/matériel et sa valeur, réduire cette solution et sa valeur à une solution et valeur équivalentes dans le problème de la bissection minimale. Il est donc possible de faire une réduction PTAS du problème de la bissection minimale au partitionnement logiciel/matériel. Or il n'existe pas de PTAS pour le problème de la bissection minimale d'un graphe (Khot, 2006). Il n'existe donc pas de PTAS pour le problème du partitionnement logiciel/matériel.

La non-existence d'un algorithme exact en temps polynomial ou pseudo-polynomial et d'un PTAS pour le partitionnement logiciel/matériel implique la non-existence d'un algorithme exact en temps polynomial ou pseudo-polynomial et d'un PTAS pour l'exploration architecturale. Ce résultat est renforcé par le fait que deux formulations de l'allocation de processeurs identiques et de l'assignation des tâches aux processeurs ont été démontrées fortement NP-difficiles (Fernandez-Baca, 1989; Papadimitriou et Yannakakis, 1990) et qu'il a été démontré qu'une de ces formulations n'a pas de PTAS (Fernandez-Baca, 1989).

ANNEXE E

GÉNÉRATION ALÉATOIRE D'UNE ARCHITECTURE

On peut générer aléatoirement une solution dans l'espace de recherche défini par la fonction f_1 (présentée à la section 8.1) en suivant les étapes suivantes :

1. Choisir aléatoirement les i modules à implémenter en logiciel (et donc les $n-i$ modules à implémenter en matériel) ;
2. Choisir aléatoirement le nombre j de processeurs à allouer ;
3. Grouper aléatoirement les $i+s$ tâches logicielles en j ensembles non vides et assigner chacun de ces ensemble à un des j processeurs ;
4. Choisir aléatoirement le nombre k de processeurs qui seront assignés sur un seul bus, allouer k bus et y assigner un des k processeurs sur chacun de bus ;
5. Choisir aléatoirement le nombre l de bus restants à allouer ;
6. Grouper aléatoirement les $n-i+j-k+h$ composants matériels en l ensembles contenant chacun au moins deux composants et assigner chacun de ces ensemble à un des l bus.

L'algorithme de génération aléatoire implémenté vise à assurer une distribution uniforme entre les solutions possibles. En d'autres termes, chaque architecture a une chance égale d'être choisie. Pour ce faire, on donne à chaque choix une probabilité proportionnelle au nombre d'architectures qui peuvent être générées avec ce choix. Par exemple, si une application a n modules, s tâches logicielles et h composants matériels, alors le nombre de total de solutions possibles est de $E(n,s,h)$, tel que présenté à la section C.1.5. Si on décide que i modules seront implémentés en logiciel, il y a alors $\binom{n}{i}$ manières de choisir ces i modules et $E(0,i+s,n-i+h)$ solutions possibles après ce partitionnement logiciel/matériel. L'algorithme de génération aléatoire construira donc une solution avec i modules logiciels avec une probabilité p_i donnée par l'équation suivante :

$$p_i(i) = \frac{\binom{n}{i} E(0, i+s, n-i+h)}{E(n,s,h)} \tag{E.1}$$

On vérifie aisément que $\sum p_i = 1$. Une fois qu'on a choisi le nombre i de modules à implémenter en logiciel, on peut choisir aléatoirement ces i modules. En utilisant le même raisonnement que pour l'équation précédente, l'algorithme alloue un nombre j processeurs avec une probabilité p_j :

$$p_j(j) = \frac{S(i+s,j)\sum_{k=0}^{j}\binom{j}{k}B_2(n-i+j-k+h)}{E(0,i+s,n-i+h)} \tag{E.2}$$

L'algorithme choisit ensuite aléatoirement une des $S(i+s,j)$ assignations possibles des tâches aux processeurs. La probabilité p_k que k processeurs se trouvent chacun seul sur k bus est donnée par :

$$p_k(k) = \frac{\binom{j}{k}B_2(n-i+j-k+h)}{\sum_{p=0}^{j}\binom{j}{p}B_2(n-i+j-p+h)} \tag{E.3}$$

Ensuite, l'algorithme alloue les l autres bus selon une probabilité p_l :

$$p_l(l) = \frac{S_2(n-i+j-k+h,l)}{B_2(n-i+j-k+h)} \tag{E.4}$$

L'algorithme termine alors par choisir une des $S_2(n-i+j-k+h,l)$ assignations possibles des composants matériels aux bus. Le résultat final est une architecture générée aléatoirement selon une distribution uniforme.

ANNEXE F

DÉTAILS DES ÉTUDES DE CAS

F.1 Décodeur JPEG avec détection de la peau

Dans l'application du décodeur JPEG avec détection de la peau présentée à la section 9.1.2 et à la figure 9.2, la mémoire JPEG contient une ou plusieurs images à traiter qui sont encodées selon le standard JPEG (Pennebaker et Mitchell, 1993) du *Joint Photographic Experts Group*. Lors de leur encodage, l'espace colorimétrique de chacune de ces images a d'abord été transformé vers l'espace colorimétrique YCbCr, dont la composante Y représente la luminance alors que les composantes Cb et Cr représentent respectivement les chrominances en bleu et en rouge. Chaque image a ensuite été divisée en macroblocs de 16x16 pixels. Six blocs de 8x8 ont été extraits à partir de chaque macrobloc : 4 blocs de luminance (Y) pour les 4 blocs de 8x8 pixels du macrobloc ainsi que 2 blocs de chrominance (Cb et Cr) pour l'ensemble du macrobloc. Les chrominances sont sous-échantillonnées étant donné que l'oeil humain est moins sensible à la chrominance qu'à la luminance. Une transformée en cosinus discrète (DCT) a ensuite amené chacun de ces blocs de 8x8 vers un domaine fréquentiel, puis une quantification a retiré de ces blocs les informations de haute fréquence. Finalement, un encodage RLE (*run-length encoding*) suivi d'un encodage Huffman (Huffman, 1952) a terminé la compression de l'image : le résultat de cette compression, auquel on a ajouté les tables utilisées pour la quantification et le codage Huffman, est alors l'image dans un format JPEG.

Ainsi, parmi les modules du décodeur JPEG avec détection de la peau, l'extracteur agit comme contrôleur pour l'ensemble de l'application. Pour chaque image JPEG, il lit les marqueurs dans l'image JPEG afin de déterminer à quelles adresses se trouvent les tables de quantification, les tables de codage Huffman, puis les macroblocs. Il envoie l'adresse de chaque table de quantification au quantificateur inverse afin qu'il puisse se configurer en lisant ces tables dans la mémoire JPEG. De la même manière, l'extracteur envoie l'adresse des tables de codage Huffman au décodeur Huffman et celui-ci les lit dans la mémoire JPEG. L'extracteur envoie ensuite l'adresse des macroblocs au décodeur Huffman. Celui-ci lit successivement chaque macrobloc dans la mémoire JPEG et les soumet à un décodage RLE et Huffman, selon ses tables de codage Huffman. Le décodeur Huffman envoie chaque macrobloc décompressé au quantificateur inverse et celui-ci, après une quantification inverse, envoie chaque macrobloc dé-quantifié à l'IDCT. Celle-ci applique une DCT inverse selon l'algorithme de (Loeffer *et al.*,

1989) sur chacun des blocs 8x8 du macrobloc et ceux-ci se trouvent alors au format YCbCr. L'IDCT envoie alors chaque macrobloc à Y2R pour être converti à l'espace colorimétrique RGB. Y2R restitue donc des macroblocs de 16x16 pixels au format RGB et il les écrit dans la mémoire RGB afin de restituer l'image RGB. La détection de peau lit alors chaque pixel de l'image RGB dans la mémoire RGB et détermine, à l'aide des règles de (Kovac *et al.*, 2003), si il s'agit d'un pixel de peau ou non. La détection de peau convertit chaque pixel dans un espace colorimétrique en tons de gris : un pixel de peau devient gris avec une luminance égale à celle du pixel RGB correspondant, alors que les autres pixels deviennent blancs. Le détecteur de peau écrit ces pixels dans la mémoire en tons de gris afin de produire l'image en tons de gris.

F.2 Encodeur/décodeur WIMAX

Le codeur/décodeur pour la couche physique du standard WiMAX comprend, tel que présenté à la section 9.1.3 et à la figure 9.3, une chaîne de codage et une chaîne de décodage WiMAX. Cette chaîne de codage comprend sept modules, en plus du paramétreur et du module transmetteur lui-même : le mélangeur, l'encodeur Reed-Solomon, le convolueur, le perforateur, l'entrelaceur, le modulateur et l'aiguilleur. Cette chaîne de codage introduit de la redondance aux données transmises afin de permettre la détection et la correction des erreurs de bits qui pourraient se produire lors de la transmission sans fil. Elle s'assure également que les données soient mises en forme pour être transmises comme un signal électromagnétique analogique modulé sur un ensemble d'ondes porteuses. Ainsi, le mélangeur applique d'abord un registre à décalage à rétroaction linéaire (LFSR) aux données fournies par le transmetteur afin d'équilibrer le nombre de bits égaux à '0' et à '1' pour les étapes subséquentes. L'encodeur Reed-Solomon applique ensuite un code Reed-Solomon (Reed et Solomon, 1960) aux données afin de leur ajouter de la redondance et permettre la correction des erreurs. Le convolueur ajoute également de la redondance sous la forme de bits de parité produits à partir d'un registre à décalage. Le perforateur supprime quant à lui une partie de la redondance en supprimant des bits selon le taux de transfert, ce qui permet de contrôler la vitesse et la fiabilité de la transmission. L'entrelaceur permute ensuite l'ordre des bits du paquet selon la modulation choisie pour que les bits de données adjacents ne soient pas transmis sur des fréquences porteuses adjacentes. Le modulateur génère ensuite, selon la modulation choisie, les symboles qui seront transmis sur le canal de communication, alors que l'aiguilleur répartit ces symboles sur les différentes fréquences d'ondes porteuses selon le nombre de sous-canaux utilisés. Les dernières étapes menant à la génération d'un signal électromagnétique, soit la transformée de Fourier inverse rapide et la conversion numérique-analogique, ne sont pas font

pas partie du modèle SPACE.

La chaîne de décodage WiMAX effectue essentiellement les opérations inverses de la chaîne de codage. Elle débute par deux étapes qui ne font pas partie du modèle SPACE, soit la conversion analogique-numérique et la transformée de Fourier rapide. Cette chaîne de décodage est composée de cinq modules, en plus du récepteur lui-même : le dé-modulateur, le dé-entrelaceur, le décodeur Viterbi, le décodeur Reed-Solomon et le dé-mélangeur. Le démodulateur commence par récupérer à partir des symboles du signal les données entrelacées selon la modulation et le nombre de sous-canaux. Le dé-entrelaceur permute alors les bits des données récupérées dans une opération inverse à celle de l'entrelaceur. Le décodeur Viterbi applique ensuite l'algorithme de Viterbi (Viterbi, 1967) aux données pour inverser la perforation et la convolution des données tout en corrigeant des erreurs de bits qui auraient pu se produire lors de la transmission du signal. Le décodeur Reed-Solomon inverse ensuite l'opération de l'encodeur Reed-Solomon tout en corrigeant d'autres erreurs qui n'auraient pas été corrigées par le décodeur Viterbi. Pour ce faire, le décodeur Reed-Solomon applique l'algorithme de Berlekamp-Massey (Massey, 1969) puis l'algorithme de recherche de Chien (Chien, 1964) pour localiser les erreurs qui sont ensuite corrigées par l'algorithme de Forney (Forney, 1965). La variante de l'algorithme de Berlekamp-Massey utilisé n'effectue pas de division de polynôme (Jeng et Truong, 1999). Finalement, le dé-mélangeur applique un LFSR pour inverser l'opération du mélangeur et restituer les données originales au récepteur. On peut trouver dans (Bah, 2010) plus de détails sur la modélisation de cette application avec SPACE.